Pontifícia Universidade Católica
do Rio de Janeiro

Alexander Chávez López

How does refactoring affect internal quality attributes? A multi-project study

Dissertação de Mestrado

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

Advisor: Prof. Alessandro Fabricio Garcia

Rio de Janeiro
September 2017

PUC-Rio - Certificação Digital Nº 1522494/CA

Pontifícia Universidade Católica
do Rio de Janeiro

Alexander Chávez López

How does refactoring affect internal quality attributes? A multi-project study

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

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

Prof. Arndt von Staa
Departamento de Informática – PUC-Rio

Prof. Gleison dos Santos Souza
UNIRIO

Prof. Márcio da Silveira Carvalho
Vice Dean of Graduate Studies
Centro Técnico Científico – PUC-Rio

Rio de Janeiro, September the 23rd, 2017

PUC-Rio - Certificação Digital Nº 1522494/CA

All rights reserved.

Alexander Chávez López

Alexander is a bachelor in Computer Engineering at Matanzas University "Camilo Cienfuegos" of Cuba (2012). He has worked on research projects in software engineering, information systems, and higher education in computing. Relevant venues have been accepting his work for publication, such as Conference on Foundations of Software Engineering (FSE'17), Brazilian Symposium on Software Engineerings (SBES'17), International Workshop on Semantic Evaluation (SemEval'14), and Joint Conference on Lexical and Computational Semantics (SEM'12 and '13). Alexander is a research scholar in software engineering for Opus Research Group at PUC-Rio.

PUC-Rio - Certificação Digital Nº 1522494/CA

Ficha Catalográfica

Chávez López, Alexander

How does refactoring affect internal quality attributes? A multi-project study / Alexander Chávez López; advisor: Alessandro Fabricio Garcia. 2017.

80 f. : il. color. ; 30 cm

Dissertação (mestrado) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática, 2017.

Inclui bibliografia

1. Informática – Teses. 2. Refatoração;. 3. Re-refatoração;. 4. Qualidade Estrutural de Código-Fonte;. 5. Atributos Internos de Qualidade;. 6. Métricas de Software.. I. Garcia, Alessandro Fabricio. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

PUC-Rio - Certificação Digital Nº 1522494/CA

To my dad. There is nothing I would not do,
to hear your voice again.

Acknowledgments

To my Lord for being so special and always guiding me. To my parents, for the support in both sunny and stormy days. To my mother for being my guide and co-author of my victories. To my dad for his support and love until his last days. To my wife for always being ready to help me in every moment. I love you now and forever.

To the advisor of this dissertation, for the support and for being an essential part of my professional progress. I have never seen such a qualified educator like him. To my friend Eduardo for all the help. I never met someone who radiates so much positive energy as Eduardo. To all my colleagues and friends from the OPUS research group, especially Diego for allowing me to be part of the “refactoring team.”

To my niece and nephew, Lindsay and Anthony, who filled me with inspiration. To my brothers and sister Ruben, Orlandito, and Elizabeth. To my friends Otoniel, Hector, Armando, Jalal, Alain, and Adriel.

Finally, to the Informatics Department (DI), as well as the Coordination for the Improvement of Higher Education Personnel (CAPES). I appreciate every opportunity that I had, including the financial support. Thank you very much.

PUC-Rio - Certificação Digital Nº 1522494/CA

Abstract

Chávez López, Alexander; Garcia, Alessandro Fabricio (Advisor). **How does refactoring affect internal quality attributes? A multi-project study**. Rio de Janeiro, 2017. 80p. Dissertação de mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Developers often apply code refactoring to improve the internal quality attributes of a program, such as coupling and size. Given the structural decay of certain program elements, developers may need to apply multiple refactorings to these elements to achieve quality attribute improvements. We call *re-refactoring* when developers refactor again a previously refactored element in a program, such as a method or a class. There is limited empirical knowledge on to what extent developers successfully improve internal quality attributes through (re-)refactoring in their actual software projects. This dissertation addresses this limitation by investigating the impact of (re-)refactoring on five well-known internal quality attributes: cohesion, complexity, coupling, inheritance, and size. We also rely on the version history of 23 open source projects, which have 29,303 refactoring operations and 49.55% of re-refactoring operations. Our analysis revealed relevant findings. First, developers apply more than 93.45% of refactoring and re-refactoring operations to code elements with at least one critical internal quality attribute, as oppositely found in previous work. Second, 65% of the operations actually improve the relevant attributes, i.e. those attributes that are actually related to the refactoring type being applied; the remaining 35% operations keep the relevant quality attributes unaffected. Third, whenever refactoring operations are applied without additional changes, which we call *root-canal refactoring*, the internal quality attributes are either frequently improved or at least not worsened. Contrarily, 55% of the refactoring operations with additional changes, such as bug fixes, surprisingly improve internal quality attributes, with only 10% of the quality decline. This finding is also valid for re-refactoring. Finally, we also summarize our findings as concrete recommendations for both practitioners and researchers.

Keywords

Refactoring; Re-refactoring; Code Structural Quality; Internal Quality Attributes; Software Metrics.

PUC-Rio - Certificação Digital Nº 1522494/CA

Resumo

Chávez López, Alexander; Garcia, Alessandro Fabricio. **Como a refatoração afeta os atributos de qualidade interna? Um estudo multi-projeto**. Rio de Janeiro, 2017. 80p. Dissertação de Mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Desenvolvedores frequentemente aplicam refatoração para melhorar os atributos internos de qualidade em projetos de software, tais como acoplamento e tamanho. Chamamos de rrefatoração quando desenvolvedores refatoram um elemento de código-fonte previamente refatorado. O conhecimento empírico é limitado acerca de até que ponto refatoração e rrefatoração de fato melhoram os atributos internos de qualidade. Nesta dissertação, nós investigamos a limitação supracitada com base em cinco atributos internos de qualidade conhecidos: acoplamento, coesão, complexidade, herança e tamanho. Também nos baseamos no histórico de versionamento de 23 projetos de software de código-fonte aberto, os quais possuem 29,303 operações de refatoração e 49.55% de rrefatorações. Nossa análise revelou descobertas interessantes apresentadas como segue. Primeiro, desenvolvedores aplicam mais de 93.45% de operações de refatoração e rrefatoração sobre elementos de código-fonte com ao menos um atributo interno de qualidade crítico, contrariando trabalhos anteriores. Segundo, para 65% das operações, os atributos internos de qualidade relacionados melhoram, enquanto que os demais 35% permanecem não-afetados. Terceiro, sempre que operações de refatoração são aplicadas sem mudanças adicionais no código-fonte, o que chamamos de operação de refatoração root-canal, os atributos internos de qualidade frequentemente melhoram, ou ao menos, não pioram. Ao contrário, 55% das operações de refatoração aplicadas com mudanças adicionais, tais como correção de bugs, surpreendentemente melhoram os atributos internos de qualidade, com somente 10% de piora, o que também é válido para rrefatoração. Nós sumarizamos nossas descobertas na forma de recomendações para desenvolvedores e pesquisadores.

Palavras-chave

Refatoração; Re-refatoração; Qualidade Estrutural de Código-Fonte; Atributos Internos de Qualidade; Métricas de Software.

PUC-Rio - Certificação Digital Nº 1522494/CA

Table of contents

PUC-Rio - Certificação Digital Nº 1522494/CA

PUC-Rio - Certificação Digital Nº 1522494/CA

List of figures

PUC-Rio - Certificação Digital Nº 1522494/CA

List of tables

PUC-Rio - Certificação Digital Nº 1522494/CA

PUC-Rio - Certificação Digital Nº 1522494/CA

You never fail until you stop trying.

Albert Einstein, *(1879-1955).*

1 Introduction

Developers apply refactoring operations to improve the code structural quality (1). These operations may affect multiple code elements together (2, 3). For instance, *Move Method* moves a method from one class to another aimed at improving the class *cohesion*. In turn, *Extract Superclass* (1) extracts a superclass and moves the functionalities shared by multiple classes to the superclass, which affects both *size* and *inheritance* of the classes converted into subclasses. Thus, one could expect that different refactoring types affect specific internal quality attributes. Consequently, by analyzing internal quality attributes, such as *cohesion* and *size*, could reveal the code structural improvements caused by refactoring operations. However, we have limited empirical evidence on the effects of refactoring operations on internal quality attributes.

Studies often assume that internal quality attributes help assess the code structural quality (4, 5). For instance, the *coupling* attribute is measured by the number of dependencies that one code element has with others (6). In fact, assuring code structural quality often concerns developers, who intend to foster the longevity of their software projects (7, 2). Thus, there is a need for measuring the code structural quality and drawing strategies for improving it. Software metrics serve as indicators of the degree of satisfaction for certain internal quality attributes. Several studies (8, 9, 10) propose and evaluate metrics for internal quality attributes like *coupling* (how much two code elements inter-depend), *size* (the length of the code elements), and *complexity* (how difficult is to read and understand a code element).

Developers usually apply two refactoring tactics (11): *root-canal refactoring* and *floss refactoring*. They apply root-canal refactoring when they aim at exclusively improving the code structural quality. In contrast, developers apply floss refactoring aimed at reaching a particular goal rather than improving the code structural quality, such as adding a new functionality or fixing a bug. During root-canal refactoring, developers improve the code structural quality without further changes in the source code. However, during floss refactoring, developer applies refactoring operations and additional changes to the source code. Because of the differences between refactoring tactics, one could analyze them separately. However, we lack studies that address this topic.

PUC-Rio - Certificação Digital Nº 1522494/CA

Previous studies (12, 13) assess refactoring operations from the viewpoint of developers, aimed at investigating how often developers apply these operations (11), or what motivates developers when refactoring code (13). Other studies (14, 15, 16) assess the effect of refactoring operations applied by developers on external quality attributes of software projects, such as the reuse of refactored elements (16), or the reduction of software maintenance effort (15). Recent studies (17, 18, 19) focus on the effects of refactoring operations on poor code structures, which are characterized by multiple internal quality attributes. They observe that developers are often concerned on refactoring to improve the code structural quality. They also assume that a single refactoring operation improves different internal quality attributes together. However, they do not assess such effects on each internal quality attribute in isolation.

We illustrate how relevant is assessing the effect of refactoring operations on each internal quality attribute as follows. Let us consider a class that centralizes most of the functionalities in a software project, which leads to a high *complexity*. In addition, the centralized functionalities address varied concerns of the project, which leads to a low class *cohesion*. Finally, the class has several dependencies with other classes, which characterizes high *coupling*. Previous work (17, 18) limits to observe whether refactoring operations affect a poor code structure only when such operations affect all three attributes together, namely *complexity*, *cohesion*, and *coupling*. Their results suggest that refactoring operations do not positively affect such poor code structures, though each operation aims at improving the code structural quality. However, it may be the case that each refactoring operation does not suffice to improve the whole poor code structure. In fact, they may improve certain internal quality attributes in isolation. Thus, by assessing the effect of refactoring operations on each attribute, we may reveal such effects.

Besides refactoring, the literature introduces *re-refactoring* (20, 21, 22) as a sequence of refactoring operations applied to a single code element. Similarly to refactoring, re-refactoring aims at improving the code structural quality (22). To investigate the effects of re-refactoring on internal quality attributes is relevant due to the following reasons. First, since re-refactoring occurs when developers apply several refactoring operations to a single code element, it may suggest that the developers are mostly concerned about problems in the code structural quality which are difficult to eliminate. Second, by applying several refactoring operations to a single code element, developers may affect differently the code structural quality when compared to refactoring operations applied in isolation. Again, there is still limited empirical knowledge on the effects of re-refactoring on internal quality attributes.

PUC-Rio - Certificação Digital Nº 1522494/CA

In a previous work (23), we partially address the aforementioned limitations. Our goal was empirically assessing the effects of refactoring operations on internal quality attributes. We rely on the version history of 23 Java open source software projects of GitHub with 113,306 commits and 29,303 refactoring operations. We consider 11 of the refactoring types most applied by developers (2, 12). We also consider five internal quality attributes often mentioned in the well-known Fowler's book (1): *cohesion*, *complexity*, *coupling*, *inheritance*, and *size*. Surprisingly, we observe that developers very often apply refactoring operations on code elements with critical internal quality attributes. These operations mostly improve or keep unaffected the internal quality attributes, which contradicts previous work (17, 18).

This dissertation presents an empirical studies aimed at understanding the relationship of refactoring and re-refactoring operations with internal quality attributes. Our results were partially reported in a recent conference paper (23). We overview our studies as follows. First, we assess whether refactoring and re-refactoring operations often affects code elements with critical internal quality attributes. For this purpose, we assess both refactoring operations, which have been limitedly explored by previous work (17, 18), in addition to re-refactoring, which no previous work have explored. Second, we deeply assess the effects of refactoring and re-refactoring operations on internal quality attributes in general and per refactoring type. We rely the five aforementioned internal quality attributes. We also aim at understanding what internal quality attributes mostly improve when the developers apply certain refactoring types. Third, we present several recommendations for developers and researchers on the application of both refactoring and re-refactoring operations, based on the internal quality attribute which could improve, remain unaffected, or worsen depending on the applied refactoring type. We discuss our empirical study, findings, and implications as follows.

PUC-Rio - Certificação Digital Nº 1522494/CA

1.1 (Re-)Refactoring and Critical Internal Quality Attributes

We conducted an empirical study aimed at assessing how often developers apply refactoring and re-refactoring operations on code elements with critical internal quality attributes. We define *critical internal quality attribute* as an attribute which has a critical value for at least one metric used to quantify it. We also define a *critical metric value* as a metric value which extrapolates a reference value (24). Our goal is revealing whether refactoring and re-refactoring operations often target on poor code structures indicated by the critical internal quality attributes. In case these operations actually affect critical attributes, developers may use both refactoring and re-refactoring operations as useful hints of parts in the source code which require improvements of the code structural quality.

Our results suggest that developers very often apply refactoring and re-refactoring operations on code elements with critical internal quality attributes. In fact, 94.64% and 99.87% of the refactoring and re-refactoring operations affect poor code structures, regardless the refactoring tactic. Moreover, 79.43% and 99.64% of the refactoring and re-refactoring operations affect code elements with multiple critical attributes. Overall, our findings have several implications. For instance, since most operations affect multiple internal quality attributes together, developers should carefully apply those operations. In addition, since each refactoring operation may contribute to the decay of the code structural quality, and 10% more re-refactoring operations affect poor code structures than refactoring operations, re-refactoring also requires a careful usage. Finally, regarding refactoring operations only, 72.82% are floss refactoring, i.e., mostly developers are not exactly concerned about improving the code structural quality. Thus, developers should carefully change the source after refactoring it, because these changes may negatively affect the code structural quality.

1.2 Effects of Refactoring on Internal Quality Attributes

Our empirical study also aimed at understanding the consequential effects of refactoring operations on internal quality attributes. In a first moment, we focus on refactoring operations only. We then investigate whether the internal quality attributes tend to improve, worsen, or remain unaffected as a consequence of refactoring operations applied by developers. Our goal is to confirm or refute whether developers improve at least one internal quality attribute after refactoring code elements, as previously speculated by the literature (1). For instance, we may surprisingly observe that certain refactoring types tend to worsen specific internal quality attributes.

The emphasis on code structural quality varies depending on the tactic used by developers to apply refactoring operations. Thus, in order to better understand the effects of refactoring operations on internal quality attributes, we consider two refactoring tactics: *root-canal refactoring*, in which developers are explicitly concerned with improving the code structural quality, and *floss refactoring*, in which developers use refactoring operations as means to reach other goals rather than only improving the code structural quality, such as adding new functionalities or fixing bugs. Overall, there is limited knowledge

PUC-Rio - Certificação Digital Nº 1522494/CA

regarding the differences of both tactics. Thus, we aim at observing possible varied effects for different refactoring tactics applied by developers, regarding the internal quality attributes.

As a result, we observe that refactoring operations often improve or keep unaffected the internal quality attributes, regardless the refactoring tactic (*root-canal refactoring* and *floss refactoring*). Our findings confirm the assumptions of previous work (25, 26) which state that internal quality attributes like *cohesion* and *coupling* actually provide hints of poor code structures. On the other hand, our findings contradict observations of previous work (17, 18), which state that refactoring operations often worsen the code structural quality. There are several implications of our findings. For instance, because several refactoring types often improve the code structural quality, developers should strongly consider applying them to improve the internal quality of their software projects. These refactoring types should also be recommended by tools aimed at supporting the quality improvement of software projects. However, specific refactoring types which tend to worsen the code structural quality should be carefully used by developers. In addition, tool aimed at supporting the quality improvement of projects should warn developers when applying such refactoring types.

PUC-Rio - Certificação Digital Nº 1522494/CA

1.3 Effects of Re-refactoring on Internal Quality attributes

Finally, our study aimed at understanding the effects of re-refactoring operations on the internal quality attributes. Similarly to the study with refactoring operations, we investigate whether re-refactoring operations tend to improve, worsen, or keep unaffected the internal quality attributes. Inspired by previous work (27), we refer to re-refactoring operation as any refactoring operation applied on a code element anytime after other refactoring operations being applied on the same code element. The re-refactoring operations may occur in two situations: exactly in the same commit affected by the previous refactoring operation; or in a different commit, which is either the next or a more distant commit in the evolution of the project.

Previous work (2) provide evidence that developers eventually apply re-refactoring operations on certain code elements, aimed at improving their code structural quality. However, there is still empirical knowledge on the possible effects of re-refactoring operations on the code structural quality. In fact, by relying on Fowler's definition of refactoring operation (1), one could expect that multiple refactoring operations applied to a single code element, one after the other, also improves the code structural quality. Thus, this dissertation addresses the aforementioned limitation by investigating how re-refactoring operations actually affect the five aforementioned internal quality attributes: *cohesion*, *complexity*, *coupling*, *inheritance*, and *size*.

Due to the lack of empirical evidence of the effects of re-refactoring operations on internal quality attributes, it could be interesting the study these operations, which previous work suggest to have a potential to support the maintenance of software projects (28). In fact, recent studies (29, 27) assume that re-refactoring could provide several benefits, such as the improve the code structural quality and prevent the decay of the project quality as a whole. Our goal is to confirm or refute these assumptions. We expect to provide findings with relevant implications for both industry and academia.

Similarly to the study with refactoring operations, for some refactoring types, the internal quality attributes tend to improve or remain unaffected when developers refactor code that was refactored in the past (re-refactoring). Finally, there is no difference between the results of refactorings and re-refactorings operations regarding the effect on the internal quality attributes.

PUC-Rio - Certificação Digital Nº 1522494/CA

1.4 Contributions

This dissertation summarizes several contributions on the investigation of refactoring and re-refactoring operations, which provide insights for both researchers and practitioners. In fact, the observed effects of refactoring and re-refactoring operations on internal quality attributes could support novel techniques for recommending refactoring operations to developers. By knowing the refactoring types which improve the code structural quality, these techniques could suggest operations which fit the expectation of developers regarding the code structural quality. Moreover, techniques could warn developers when applying operations which potentially cause the decay of the code structural quality. We discuss each contribution as follows.

– This dissertation investigates how refactoring and re-refactoring operations affect internal quality attributes. We observe that developers often apply these operations on code elements with critical internal quality attributes, regardless their purpose when refactoring code. Moreover, these operations mostly improve or keep unaffected the code structural quality. Our results suggest that developers should strongly consider applying refactoring and re-refactoring to improve the code structural quality. However, they should be careful when applying certain refactoring types, which could worsen the code structural quality.

– The dissertation also compares our empirical study with previous studies (17, 18). Surprisingly, our study findings partially contradict the findings of these studies. In fact, they suggest the lack of a clear relationship between refactoring operations and software metrics for internal quality attributes (17), or that refactoring operations rarely improve the code structural quality (18). However, our findings suggest that most refactoring operations are applied on poor code structures, by positively affecting it and improving the code structural quality.

– Moreover, the dissertation investigates the effects of re-refactoring operations on internal quality attributes, which is barely unexplored by previous work. Similarly to the analysis of refactoring operations, we observe that re-refactoring operations tend to improve or remain unaffected the internal quality attributes for some refactoring types. In addition, our results suggest that developers tend to apply refactoring operations to the same code elements in multiple times along the maintenance and evolution of software projects. This observations is reinforced by the fact that a half of the refactoring operations analyzed in this dissertation are also re-refactoring operations.

– By relying on our study with refactoring operations, we provide several recommendations for applying these operations aimed at improving the code structural quality. That is, we present what refactoring operations developers should apply when are concerned about improving certain internal quality attributes. These recommendations are directly related to the refactoring types which improve, remain unaffected, or worsen specific internal quality attributes. Thus, we could help developers in real development settings during the maintenance of software systems. These recommendations could support the proposal of refactoring recommendation tools based on the internal quality attributes which mostly concern developers and organizations.

– Finally, the dissertation provides a mapping of several refactoring types from the literature (1) with five well-known internal quality attributes. Additionally, the dissertation provides a mapping of these five internal quality attributes with software metrics which are commonly used to quantify them (30, 31). Although previous work (32) provide similar mappings, we cover a larger set of refactoring operations, internal quality attributes, and software metrics. Both mappings may support further investigations on the code structural quality.

PUC-Rio - Certificação Digital Nº 1522494/CA

1.5 Dissertation Outline

The remainder of this dissertation is organized as follows. Chapter 2 provides background information aimed at help understanding the dissertation and discusses related work. Chapter 3 describes the study design, including the study goal and research questions. The chapter also discusses some threats to the study validity. Chapter 4 presents the findings on how often refactoring and re-refactoring operations affect code elements with critical internal quality attributes. Chapter 5 discusses the findings on the effects of refactoring operations on internal quality attributes. Chapter 6 presents findings with respect to the effects of re-refactoring operations on internal quality attributes. Finally, Chapter 7 concludes the dissertation and outlines future directions.

PUC-Rio - Certificação Digital Nº 1522494/CA

2
Background and Related Work

This chapter provides background information. Section 2.1 discusses refactoring. Section 2.2 discusses internal quality attributes, and Section 2.3 discusses the quality metrics used to quantify each internal quality attribute. Additionally, we discuss previous work that conduct similar investigations. Section 2.4 discusses studies that assess the impact of refactoring operations on software quality from different perspectives. Section 2.5 discusses previous work that are closely related to ours. Finally, Section 2.6 discusses studies related to re-refactoring operations.

2.1 Refactoring

PUC-Rio - Certificação Digital Nº 1522494/CA

Code refactoring means changing the source code without changing the external behavior of a program but improving its internal code structure (1). Each refactoring operation is a micro-transformation that affects multiple elements in the source code. An example of refactoring operation is when the developer moves a method from one class to another, for example, to remove excessive dependencies between classes. This refactoring type is called *Move Method*. There are several other refactoring types with different purposes, such as extracting new code elements from excessively complex elements and managing class inheritances.

We selected 11 refactoring types from the Fowler's catalog (33). These refactoring types have been largely investigated in the literature (12). Table 2.1 summarizes the 11 refactoring types analyzed in our study. The first column presents the refactoring type. The second column informs the problem that each refactoring type addresses. The third column describes the solution aimed by applying each refactoring type. For instance, when a class provides a set of resources to other classes, one may apply a *Extract Interface* to provide only the resources of interest to the client classes.

As illustrated in Table 2.1, each refactoring operation is a potential solution for a structural problem in the source code. Therefore, before selecting and applying a refactoring type, developers need to identify what form of structural problem is occurring. In order to perform this identification,

Table 2.1: Analyzed refactoring types, extracted from (1)

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 itself, use this technique	Replace calls to the method with the method's content and delete the method itself
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 Method	The name of a method does not reveal its purpose	Change the name of the method
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

PUC-Rio - Certificação Digital Nº 1522494/CA

developers can rely on the analysis of code smells (1). Code smells are recurrent structures in the source code often considered key indicators of software quality degradation (1, 34, 35). Examples of code smell types vary from method-level smells, such as *Long Method* and *Feature Envy*, to class-level smells, such as *God Class* and *Shotgun Surgery* (36, 1).

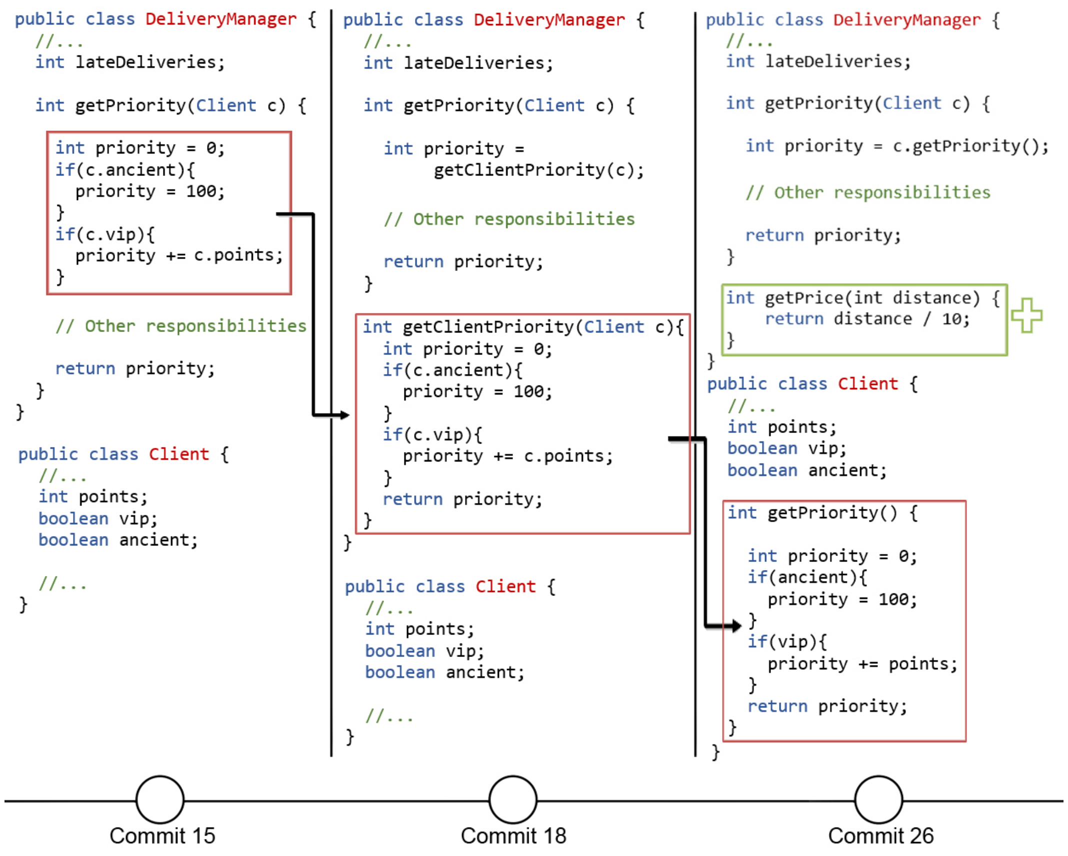
Refactored elements. As aforementioned, a single refactoring operation may affect multiple code elements. Thus, we consider as refactored elements the set of elements directly involved in the refactoring operation. For instance, let us consider the *Move Method* refactoring. In this refactoring type, a method m is moved from class A to class B. Hence, the set of refactored elements is $\{m, A, B\}$.

PUC-Rio - Certificação Digital Nº 1522494/CA

Refactoring tactics. According to Murphy-Hill (11), there are two ways of refactoring, namely *root-canal refactoring* and *floss refactoring.* In our study, we investigate these two refactoring tactics. Developers apply *root-canal refactoring* when they aim to exclusively improve the code structural quality. In contrast, developers apply *floss refactoring* as means to reach another particular goal rather than only improving the code structural quality, such as adding a new feature or fixing a bug. In other words, during *root-canal refactoring*, developers solely intend to improve the code structural quality without further alteration to the source code. In contrast, during *floss refactoring*, the programmer apply refactoring operations interleaved with other code editing activities, such as adding a feature or fixing a bug.

Figure 2.1 shows the history of changes of two classes through three project versions, represented by commits 15, 18 and 26. The *DeliveryManager* class is responsible for managing product deliveries and the *Client* class has information and responsibilities of each client. The *getPriority* method, located in *DeliveryManager* class, computes the priority of delivery based on customer data and other criteria related to deliveries, such as the number of late deliveries (represented by the comment *// Other responsibilities*). This method contains too many lines of code and responsibilities, making it difficult to understand and maintain it. To reduce the length of a method body, developers apply an *Extract Method* refactoring operation. In *Commit 18*, developers move the code responsible for computing the priority related to client data for a new method (*getClientPriority*). In summary, the changes made in *Commit 18* were: (i) creation of a new method (*getClientPriority*); (ii) moving part of the body of the long method (*getPriority*) for the new method; and (iii) introduction of a call to the new method from which its content was extracted (body of *getPriority*). In *Commit 18*, a *root-canal refactoring* tactic was applied, since all changes were related to the refactoring operation.

In *Commit 18* of Figure 2.1, we can see that class *DeliveryManager* is computing the priority based on the client data (method *getClientPriority*); this is a responsibility that corresponds to the class *Client.* In addition, the method *getClientPriority* used data from the *Client* class more than the class where it is implemented (class *DeliveryManager*). To solve this problem, developers applied in *Commit 26* a *Move Method* refactoring operation type. The method *getClientPriority* is moved from *DeliveryManager* class to the *Client* class. In addition, the developer added a new method (*getPrice*) in *DeliveryManager* that is not related to the refactoring operation. In other words, the developer used refactoring as a means to reach another specific end, such as adding a new method. This tactic for performing a refactoring operation (*Commit 26*) is called *floss refactoring*. The analysis of both refactoring tactics may help understanding if there is a relationship between the refactoring tactics and the behavior of internal quality attributes (Section 6.4).



```
public class DeliveryManager {
  //...
  int lateDeliveries;

  int getPriority(Client c) {

    int priority = 0;
    if(c.ancient){
      priority = 100;
    }
    if(c.vip){
      priority += c.points;
    }

    // Other responsibilities

    return priority;
  }
}

 public class Client {
  //...
  int points;
  boolean vip;
  boolean ancient;

  //...
 }
```

```
public class DeliveryManager {
  //...
  int lateDeliveries;

  int getPriority(Client c) {

    int priority =
         getClientPriority(c);

    // Other responsibilities

    return priority;
  }

  int getClientPriority(Client c){
    int priority = 0;
    if(c.ancient){
      priority = 100;
    }
    if(c.vip){
      priority += c.points;
    }
    return priority;
  }
}

public class Client {
  //...
  int points;
  boolean vip;
  boolean ancient;

  //...
}
```

```
public class DeliveryManager {
  //...
  int lateDeliveries;

  int getPriority(Client c) {

    int priority = c.getPriority();

    // Other responsibilities

    return priority;
  }

  int getPrice(int distance) {
      return distance / 10;
  }
}
public class Client {
  //...
  int points;
  boolean vip;
  boolean ancient;

  int getPriority() {

    int priority = 0;
    if(ancient){
      priority = 100;
    }
    if(vip){
      priority += points;
    }
    return priority;
  }
}
```

Figure 2.1: Root-canal refactoring and Floss refactoring

Re-refactoring. Developers often need to apply more than one refactoring to improve various internal quality attributes. For instance, a developer can apply a refactoring operation in order to improve the coupling attribute. In another subsequently refactoring operation, the cohesion attribute can be improved. That is, developers need to *re-refactor* the code element in order to improve both coupling and cohesion attributes.

We define re-refactoring as the action of applying another refactoring in a code element that was refactored before. For instance, in the Figure 2.1, the *getClientPriority* method was created by applying an *Extract Method* refactoring operation in *Commit 18*. Then, in *Commit 26*, the developers applied a *Move Method* refactoring operation, moving the *getClientPriority* method from *DeliveryManager* class to *Client* class. In other words, another refactoring operation was applied in the same method that was refactored in *Commit 18* (*getClientPriority*). The refactoring operation applied in *Commit*

PUC-Rio - Certificação Digital Nº 1522494/CA

26 is a *re-refactoring operation*, since it was applied to a code element that was refactored in the past.

2.2 Internal Quality Attributes

Each refactoring operations is often supposed to improve multiple structural attributes. Internal quality attributes are key indicators of code structural quality (37). Previous work apply different software quality metrics for computing internal quality attributes (3). In this dissertation, we analyze five internal quality attributes, namely *cohesion*, *coupling*, *complexity*, *inheritance*, and *size*. We selected these internal quality attributes because they are closely related to the 11 refactoring types that we aim to analyze (Section 3.4). We describe each internal quality attributes as follows.

- **Cohesion**: Refers to the degree to which class components belong together, i.e., the degree to which the internal elements of a class are related to each other (38). Existing techniques can measure cohesion from different levels of the source code. In the context of our study, we use techniques to measure cohesion in *classes* (5, 39) and are explained in detail in Section 3.5.
- **Coupling**: Measure of the degree of interdependence between classes (40). More recent, 30 defined coupling as: "*Two classes are coupled when methods declared in one class use methods or instance variables of the other classes.*". In other words, any evidence of a class component (method or field) using components of another class constitutes coupling. Since a change in a class component may involve changes in the classes coupled to it, code elements with low coupling are easier to maintain (41).
- **Complexity**: Is the measure of the complexity of a program's decision structure and is used to indicate the complexity of a method or module. In other words, complexity is the measure of the overload of responsibilities and decision of a code element. Cyclomatic complexity (42) is a typical measure of complexity in method or function. Methods must be developed with low complexity, since that simple code elements are easy to understand and maintain (41).
- **Inheritance**: Represents parent–child relationships and is measured in terms of number of subclasses, base classes, and depth of inheritance hierarchy. Inheritance enables software reusability, but large hierarchies may complicate software maintainance (43, 44).

PUC-Rio - Certificação Digital Nº 1522494/CA

– **Size**: Is measured in terms of number of Lines of Code (LOC), number of files, methods, classes, modules. In other words, *size* measures the length of a code element. According to 41, small classes are easy to maintain (41).

It is expected that each refactoring operation affects the internal quality attributes. The refactoring operations in Figure 2.1 show the relation between refactoring and internal quality attributes. The *getPriority* method, located in *DeliveryManager* class of *Commit 15*, contains too many lines of code and responsibilities, making it difficult to understand and maintain it. Developers applied an *Extract Method* refactoring operation. This refactoring operation has a direct impact on size. Because part of the code extracted from the method *getPriority* can be implemented in other places in the code. Therefore, the developer can replace duplicates with calls to the new method (*getClient-Priority*), reducing the size of the project.

After applying the *Extract Method* refactoring operation, we have the class *DeliveryManager* that compute the priority based on the client information (method *getClientPriority* in *Commit 18*). We can observe that *get-ClientPriority* method accesses more data of *Client* class than the own class *DeliveryManager*. This causes a high *coupling* between both classes. To reduce this *coupling*, the developer moves the method *getClientPriority* to the class that uses the method the most. We can observe a direct effect on the *coupling* internal quality attribute because was reduces the excessive external accesses to data of the *Client* class. In summary, we can perceive that factoring operations have a direct effect on internal quality attributes.

PUC-Rio - Certificação Digital Nº 1522494/CA

2.3 Software Quality Metrics

Existing literature has been using software quality metrics to quantify the internal quality attributes (3, 19). For instance, the *Lines of Code (LOC)* (31) metric quantifies the *size* of elements in the source code and *Cyclomatic Complexity (CC)* (42) quantify the *complexity* of a method. Each internal quality attribute must be quantified by several quality metrics in order to capture all its properties. For instance, coupling measure the degree of interdependence between classes. This degree of independence depends on two properties of the coupling attribute: (i) the number of times that a class X uses methods and fields of another class, and (ii) the number of times that other classes use methods and fields of the class X. In other words, input and output dependencies. Therefore, we selected a set composed by 25 quality metrics by applying the following criteria. First, we selected well-known metrics from the literature (6, 31, 39, 45, 46). Second, we selected metrics that assess different properties of each internal quality attribute (30, 37, 42, 47). Third, we selected metrics with evidence of accuracy in capturing and predicting problems in structural code quality. Section 3.5 describes in details the software quality metrics used in this study to quantify each internal quality attribute.

2.4 Previous Work on Refactoring and Software Quality

Bavota et al. (17) investigate three software systems aimed at understanding whether refactoring operations are applied on code elements with certain characteristics that suggest an opportunity of refactoring. These characteristics include quality metrics. According to their results, quality metrics do not have a clear relationship with refactoring. In contrast with their study, we investigate a much larger set of software systems. Also, we have considered 25 quality metrics, 10 metrics more than those analyzed by Bavota et al. (17). In addition, they did not explicitly analyze internal quality attributes as they focus on individual quality metrics. Our results contradict their findings as we observed that refactoring operations are frequently applied on code elements with at least one critical internal quality attribute.

Other studies (17, 18) classify refactoring operations according to the occurrence, addition or removal of poor code structures. However, the improvement of code structural quality may not be perceived whether only poor code structures are considered. Full removal of these poor code structures may require the simultaneous improvement of more than one internal attribute. Developers may need several refactoring operations to fully remove these structures. To address this literature gap, we investigate the effect of refactoring operations in a finer-grained level, i.e., we analyze the impact of refactoring operations directly on internal quality attributes.

Silva et al. (13) present an empirical study on the motivation of developers when applying refactoring operations. They conclude that refactoring operations are mostly driven by changes in software requirements rather than poor code structures. Even though the authors have not analyzed the relationship between refactoring and internal quality attributes, some of their findings suggest that there might be a relationship between certain motivations and internal quality attributes. For instance, they mention that developers are concerned about removing duplicated code, which is associated with two specific internal quality attributes: *size* and *inheritance*. The same reasoning applies to other types of motivation, such as maintainability and testability. Therefore, the impact of refactoring operations on internal quality attributes may

PUC-Rio - Certificação Digital Nº 1522494/CA

PUC-Rio - Certificação Digital Nº 1522494/CA

be related to different high-level goals of developers who apply refactoring.

Furthermore, the authors (13) mention that developers are seriously concerned about avoiding code duplication. For that, they apply the *Extract Method* refactoring type to remove it. This finding suggests that, despite of the aim of developers when refactoring code (not explicitly mentioned by the authors), developers are concerned on improving the code structural quality (i.e., they apply *root-canal refactoring*).

Sokol et al. (48) present a study focused in measuring the effect of refactoring only in terms of code complexity, which is only one attribute of code quality. To measure this attribute they used Cyclomatic Complexity metric (42). The authors randomly selected the fifty refactoring methods. Their studies show that refactoring does not necessarily decrease the cyclomatic complexity but increases the maintainability and readability of the program.

Several studies evaluated refactoring effects on external quality attributes, such as maintainability, flexibility, portability, reusability, readability, testability, and understandability. Geppert et al. (15) empirically studied the impact of refactoring on changeability. They studied three factors for changeability, such as effort, customer reported defect rates and scope of changes. Wilking et al. (49) investigated the effect of refactoring on modifiability and maintainability. None of these researches investigated the effects of refactoring operations on internal quality attributes.

On the other hand, Moser et al. (16) proposed a methodology to evaluate whether or not refactoring improves reusability and reuse in an XP-like development environment. They focused on quality metrics that are considered to be relevant to reusability. They conducted a case study on a software project developed using XP-like methodology and investigated the impact of refactoring on quality metrics. Their results indicated that refactoring has a positive effect on reusability and thus and reuse in XP-like environments. This work and its results are limited to software systems developed in the context of XP-like methodology.

Kim et al. (2) assessed the refactoring benefits and challenges at Microsoft through three complementary study methods: a survey, semi-structured interviews with professional software engineers, and quantitative analysis of version history data of Windows 7. Their survey finds that the refactoring definition in practice is not confined to a rigorous definition of behavior preserving code transformations and that developers perceive that refactoring involves substantial cost and risks. Their quantitative analysis of Windows 7 version history finds the top 5 percent of preferentially refactored modules experience higher reduction in the number of inter-module dependencies and several complexity measures but increase size more than the bottom 95 percent. This indicates that measuring the impact of refactoring requires multidimensional assessment. We decided to focus on open source projects due it is difficult to have access to the proprietary code for analysis.

2.5 Previous Work on Refactoring and Internal Quality

Previous work (41, 50, 51) investigate whether refactoring operations improve the code structural quality. Du Bois and Mens (50) measure the relationship between only five quality metrics and three refactoring types: *Extract Method*, *Encapsulate Field*, and *Pull up Method*. They proposed a formalism based on abstract syntax tree representation of the source code, extended with cross-references to describe the impact of refactoring on only five metrics. The results of their work showed both positive and negative impacts on the studied metrics.

On the other hand, Kataoka et al. (41) focused only on the *coupling* metrics to evaluate the refactoring effect, comparing the metrics before and after refactoring operations. They analyzed only a single C++ program for *Extract Method* and *Extract Class* refactoring types, which were performed by a single developer. All these limitations pose threats to the validity of their findings.

Stroggylos and Spinellis (51) analyzed three open source projects and studied the effect of refactorings in different quality metrics. They used source code version control system logs in order to detect the refactoring operations. The authors search through the log entries for mentions of words *refactor, refactorng, refactored* in order to detect a refactoring operation. They concluded that refactoring does not always improve the software quality. This study is limited in the technique used to detect refactoring. Because developers commonly do not mark commit logs with words derived of refactoring (52).

We have noticed that these studies were limited to only a few internal quality attributes, a few number of projects or/and a few number of refactoring types. In our study, we try to analyze more internal quality attributes to get a more precise indication of whether or not refactoring affects internal quality attributes. Also, we analyze more projects to capture different software development contexts, and we study 11 types of refactoring, which according to Murphy-Hill et al. (12), are the most common refactoring types. Finally, we analyze the impact of refactoring on internal attributes both before and after each refactoring operation.

PUC-Rio - Certificação Digital Nº 1522494/CA

2.6 Previous Work on Re-Refactoring and Software Quality

Several studies analyze refactorings in the source code that are executed in mass, in sequence, in repetition or in batch (20, 21, 22). Arcelli et al. (22) introduced refactoring actions among three EPSILON languages. They identified different alternative kinds of antipattern-based model refactoring support. The authors called *batch refactoring* to the sequentially execution of a pre-defined antipattern set detection rules and refactorings. The process can be repeated once (i.e., standard mode) or until no more antipattern occurrences are found (i.e, iterative mode). In the iterative mode, the process is repeated until no more antipattern occurrences have been found or until the specified maximum number of iterations has been reached (21). In summary, the author used *batch refactoring* to automatically remove the antipattern occurrences.

Jiau et al. (20) proposed a new methodology OBEY for optimal *batch refactoring* plan at three levels. The work revealed *Middle Man Refactoring* in three open source software systems JDTCore, HSQLDB and JEdit. OBEY analyzes a batched refactoring plan, identifies *Middle Man* symptoms that cause sub optimal execution, and renovates the plan for optimal execution. In addition, OBEY automatically executes a batched refactoring plan as an atomic code transformation and uses metric abstractions to identify *Middle Man* symptoms. As a result, the authors concluded that the use of OBEY eliminated the efforts required for fixing the sub optimal refactoring results.

The effects of a simple refactoring on the code structural quality have been a subject of several investigations. But the effects of re-refactorings on code structural quality and internal quality attributes is limitedly explored.

PUC-Rio - Certificação Digital Nº 1522494/CA

2.7 Final Remarks

This chapter presented the basic concepts of code refactoring and internal quality attributes. Our main goal with this chapter was to provide sufficient information to support the comprehension of the study proposed in this dissertation. First, we presented the concept of refactoring and the refactoring types analyzed in this dissertation. On the other hand, we show two tactics to perform refactoring (root-canal refactoring and floss refactoring), and we discuss the concept of re-refactoring. Second, we discuss the internal quality attributes used in this study.

On the other hand, this chapter discussed studies related to the scope of this dissertation. Our discussion was divided into two research topics as follows. First, we presented previous work that study the effects of refactoring on different aspects of structural quality, such as, code smells, poor code structures and external quality attributes. Second, we discussed studies aimed at investigating the effects of refactoring on internal quality attributes. Finally, we discuss works that address re-refactorings from different approaches. None of the works explore the impact of re-refactorings on internal quality attributes.

We noticed that discussed works were limited to only a few number of software projects, few number of refactoring types or few number of internal quality attributes. In this research, we try to consider more internal quality attributes, software project and refactoring types to get a more precise indication of refactoring effects on internal quality attributes.

In the next chapter, we present the study design. First, we introduce our research questions. Following, we present the steps for collecting the necessary data to answer the research questions. The steps include (i) selection of software projects, (ii) detect refactoring operations, (iii) map refactoring types with internal quality attributes, (iv) select the quality metrics to quantify each internal quality attribute, and (v) select the tools to compute the quality metrics.

PUC-Rio - Certificação Digital Nº 1522494/CA

3 Study Design

This chapter presents the study design. Figure 3.1 summarizes the study design phases, which correspond to the topics of the sections in this chapter. First, we present our goals and our research questions; they were designed in a way that enables us to understand the effect of refactoring and re-refactoring operations on internal quality attributes (Phase 1). Second, we select a set of 23 open source project of GitHub repositories (Phase 2). Third, we used the Refactoring Miner tool to detect refactoring operations and conducted a manual validation of the refactoring types identified by the Refactoring Miner tool with a median precision of 88.36% (Phase 3). Fourth, we mapped refactoring types with internal quality attributes in order to understand how refactoring operations affect each related internal quality attribute. Fifth, after mapping each refactoring type to its related internal quality attributes, we investigated the quality metrics for quantifying each internal quality attribute. Sixth, we investigate the necessary tools to compute the selected metrics. Finally, we analyzed the data and answered our research questions.

Thus, the remainder of this chapter is organized according to Figure 3.1 as follows. Section 3.1 introduces the goals and research questions. Section 3.2 presents the selection of software projects. Section 3.3 shows how we detected refactoring operations and the procedure to classify refactoring operations in *root-canal refactoring* and *floss refactoring*. Section 3.4 presents the protocol for mapping refactoring operations to internal quality attributes. Section 3.5 illustrates how we measure internal quality attributes. Section 3.6 presents the tools used to collect quality metrics and to detect refactoring operations. Finally, Section 3.7 discusses threats to the study validity, with respective treatments.

3.1 Goal and Research Questions

Our study aims at assessing how refactoring affects internal quality attributes. We study the effect of both refactoring and re-refactoring operations. By achieving this aim, we expect our study results contribute to both researchers and practitioners for two main reasons. First, the knowledge that

PUC-Rio - Certificação Digital Nº 1522494/CA

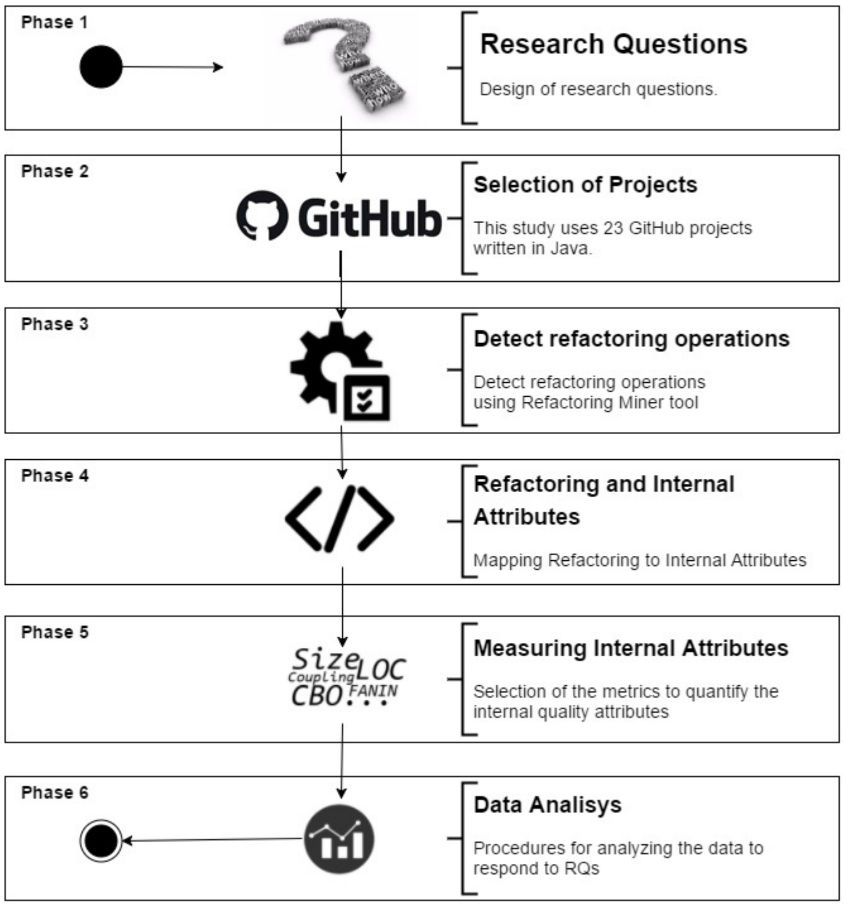

Phase 1
Research Questions
Design of research questions.
Phase 2
GitHub
Selection of Projects
This study uses 23 GitHub projects written in Java.
Phase 3
Detect refactoring operations
Detect refactoring operations using Refactoring Miner tool
Phase 4
Refactoring and Internal Attributes
Mapping Refactoring to Internal Attributes
Phase 5
Size LOC Coupling CBO FANIN ...
Measuring Internal Attributes
Selection of the metrics to quantify the internal quality attributes
Phase 6
Data Analisys
Procedures for analyzing the data to respond to RQs

Figure 3.1: Study Design

PUC-Rio - Certificação Digital Nº 1522494/CA

refactoring operations often target code elements with critical internal quality attributes (or otherwise) can help researchers and developers on improving the identification of program locations that need to be refactored. Developers often need guidance on how to prioritize refactorings in large systems. However, there is limited research effort in this field. Several studies have focused on investigating whether refactoring operations are commonly applied on code elements having bugs, code smells, design problems, and the like (17, 18, 53). However, it might be too late for a developer knowing afterwards that its refactoring operations introduced these problems in the source code. On the other hand, internal quality attributes can be measured as soon as there are preliminary versions of the source code. Thus, in this study, we analyzed whether developers tend to apply refactoring operations on code elements with at least one critical internal quality attribute. If this expectation is confirmed, our study results can: (i) guide researchers on identifying means to support early prioritization of code elements with critical internal quality attributes, and (ii) guide developers in early development stages on how to better concentrate effort on code elements with specific characteristics of structural quality decay.

Second, our analyses of the impact of refactoring (and re-refactoring)

PUC-Rio - Certificação Digital Nº 1522494/CA

operations on internal quality attributes can help researchers to produce refactoring recommendation systems that alert about refactoring operations that are potentially harmful, i.e. those operations that commonly degrade the internal quality attributes. Most refactoring recommendation systems are based on the fact that developers refactor with the sole aim of simply removing code smells (54, 55, 56, 57). As we explore the impact of refactoring (and re-refactoring) operations on structural quality, our results can better support developer with different aims. Many developers try to perform minor structural quality improvements through refactoring in order to achieve other aims, such as adding a new functionality or enhancing program testability. With our study results, we can better inform developers on incorporating rules into refactoring recommendation tools based on their different, hybrid aims, where improvement and/or worsening of internal quality attributes are desirable and/or expected. To achieve our goal, we will address the following research questions.

RQ1. Are refactoring operations often applied to code elements with critical internal attributes?

Boshnakoska and Misev (58) investigated the correlation between object-oriented software quality metrics and refactoring. They used an extended CK (30) metrics suite and concluded that object-oriented metrics can be used to identify classes that require immediate attention. In other words, object-oriented metrics are good indicators of degraded code elements, but in fact these elements are refactored in real software systems? We designed RQ1 to answer this question, i.e., investigate if developers tend to apply refactoring operations on code elements with critical internal quality attributes. Our goal is to understand if code elements with critical internal quality attributes are potential indicators of problems in the code structure which are commonly refactored. In the affirmative case, we can investigate the actual impact of refactoring operations on the internal quality attributes (RQ2).

To answer RQ1, we need to assess the criticality of internal quality attributes for each code element affected by a refactoring operation. For this purpose, we annotated each refactoring operation according to the number of critical internal attributes present in the code elements affected by the refactoring. Thus, refactored code elements can be affected by: (i) no critical attribute, i.e., elements that do not have any critical attribute, (ii) single critical attribute, i.e., elements that have only one bad attribute, and (iii) multiple critical attributes, i.e., elements that have more than one critical attribute.

In our study, an attribute is critical whether at least one metric used to quantify the attribute is either below a lower threshold or above an upper threshold. Thresholds are computed for each project's commits. These threshold values are calculated based on quartiles (59) for the commit. The *first* and *third* quartiles define the lower and upper thresholds, respectively. For each commit, we collect the metrics related to each code element. Then, we define the lower and upper thresholds of each metric by calculating the *first* and *third* quartile of the metric values obtained from all the code elements. The first quartile (Q1) is the lower threshold, which comprises the 25% of the bottom data (metrics). The third quartile (Q3) is the upper threshold, which comprises the top 25% of the data above it. Thus, if a metric value of an element is above the Q3, then we say that metric got critical in that element. There are few metrics that are critical if their value is below the Q1, according to the nature of the metric. In other words, the values of quality metrics are desired to be lower or higher. For instance, the value of Lack of Cohesion of Method 2 (LCOM2) (30), Depth of Inheritance Tree (DIT) (30), Weighted Method per Class (WMC) (30), Lines of Code (LOC) (31) and Cyclomatic Complexity (CC) (42) are desired to be lower in a system whereas Tight Class Cohesion (TCC) (39) are desired to be higher (60). If an element has more than one critical metric, then we say that the internal quality attribute related to those metrics is also critical.

RQ2. What is the impact of refactoring on internal quality attributes?

Each internal quality attribute is measured using a set of quality metrics, where each metric captures a distinct property of each internal attribute. For instance, to measure coupling it is not enough to know the number of elements of a class that calls elements of another class. We also need to capture the number of elements of other classes that call elements of the class being measured. Thus, for each internal attribute, we selected a set of metrics (Section 3.5) that may have improved, worsened or may not be affected after applying a refactoring operation. The behavior of each metric is calculated by comparing the metric value of the code elements in the commits before and after each refactoring operation.

To answer our RQ2, we divide the analysis into two approaches, related to how to classify the behavior of internal quality attributes: (i) *At Least One Metric*, and (ii) *Most Metrics*. In the first approach, we assume that the internal quality attribute improves when at least one of the metrics that quantifies such attribute improves. In the second approach, we assume that an internal quality attribute improves when most of the metrics that capture the attribute also improve. Both approaches are explained in more detail below.

PUC-Rio - Certificação Digital Nº 1522494/CA

Most metrics. An internal quality attribute improves when most of the metrics that capture the attribute also improve. This approach tries to cover the scenarios in which several properties of each attribute are improved by a refactoring operation. This approach is strict because each refactoring operation has to improve several metrics at once. This strict approach does not consider an attribute improvement if only a few attribute's metrics (less than a half of the metrics) improve. The majority of the metrics has to improve in order to represent an improvement of the internal quality attribute. There might be several metrics for a single internal quality attribute. For instance, Size has eight metrics, and Complexity and Coupling have five attributes each. Because each metric captures a specific property of the attribute, an improvement represents an overall improvement of the attribute properties.

At least one metric. An internal quality attribute improves when at least one of the metrics that quantify such attribute also improves. This approach is interesting because the developer may improve only one of the properties of an internal quality attribute in each refactoring operation. On the other hand, we try to resolve other study problems (Section 2.4) that measure internal quality attributes using only one quality metric. This approach is less strict than the *most metrics* approach because, for each refactoring operation, only one metric has to improve in order to consider the improvement of the internal quality attribute. With this approach, we aim to understand the impact of refactoring operations on a smaller metrics subset (less than a half) that capture each internal attribute. This analysis as a relevant resource of data to identify the improvement of specific properties of each internal attributes.

RQ3. What is the impact of re-refactoring on internal quality attributes?

Developers may need to apply more than one refactoring to improve one or various internal quality attributes. In fact, a developer may often apply multiple refactoring operations in a single code element. Suppose a developer applies a refactoring to a code element in an X commit. Then, it applies another refactoring in the same code element in a Y commit, where Y is greater than X in terms of time. We assume that the refactoring applied to commit Y is a re-refactoring. In other words, we define re-refactoring as the action of applying another refactoring operation (not necessarily the same refactoring type) in a code element that was refactored before. We disregard the temporal dimension here, i.e., a new refactoring applied to the same previously refactored element is considered a re-refactoring operation regardless when it was applied

PUC-Rio - Certificação Digital Nº 1522494/CA

to the same code element (e.g., not necessarily in successive commits, weeks or months).

To answer our RQ3, we analyze the impact of re-refactoring operations on internal quality attributes. Similar to RQ2, we divide the analysis into two approaches: *Most metrics* and *At least one metric*. We also analyze the impact of re-refactoring operations on internal quality attributes by analyzing both refactoring tactics: *Root-canal refactoring* and *Floss refactoring*.

3.2 Selection of Software Projects

The software development process is managed using software repositories that include source code, documentation, archived communications and defect-tracking systems. The researchers and developers can use the information contained in these repositories for maintaining software systems, improving software quality, and empirical validation of data and techniques. Researchers can mine these repositories to understand software development, software evolution and how some concepts like refactoring impact on the improvement of the quality of the source code.

To conduct our study, we selected a set of software projects from GitHub repositories because we are concerned with the evolution of software projects regarding refactoring operations. We focused the data analysis on open source projects to support the study replication and extension. In this study, we analyze 23 software projects selected using the following quality criteria. First, Java software projects, a very popular programming language[1]. Second, open source projects, to allow the study replication. Third, highly popular projects, based on the number of stars received by the projects. In addition, we decided to focus on open source projects due to three main reasons. First, it is difficult to have access to the proprietary code for analysis, but there are several open source projects with many clients and contributors, e.g., Elasticsearch (851 contributors) and Junit (137 contributors). Second, these projects are often under maintenance. Thus, assuring the code structural quality is as important as in the case of proprietary systems (2, 7). Third, by analyzing open source projects, we can compare our results with previous work (17, 18) that also analyze open source projects.

Table 3.1 lists the selected software projects. The first column presents the partial repository path at GitHub. The second column provides the number of commits per project. The third column presents the number of refactoring operations per project. The fourth column shows the number of contributors

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

PUC-Rio - Certificação Digital Nº 1522494/CA

per project. The projects have, in total, 113,306 commits, 29,303 refactoring operations and 2,843 contributors.

Table 3.1: Software Projects Analyzed in this Study

Software Project	**# Commits**	**# Refactorings**	**# Contributors**
alibaba/dubbo	1,836	280	30
AndroidBootstrap/android-bootstrap	230	8	17
apache/ant	13,331	2,063	29
argouml	17,654	2,588	87
elastic/elasticsearch	23,597	9,507	879
facebook/facebook-android-sdk	601	341	45
facebook/fresco	744	161	98
google/iosched	129	73	40
google/j2objc	2,823	713	36
junit-team/junit4	2,113	309	138
Netflix/Hystrix	1,847	266	103
Netflix/SimianArmy	710	55	46
orhanobut/logger	68	20	9
PhilJay/MPAndroidChart	1,737	398	57
prestodb/presto	8,056	2,068	195
realm/realm-java	5,916	1,699	70
spring-projects/spring-boot	8,529	1,386	378
spring-projects/spring-framework	12,974	5,320	228
square/dagger	696	96	35
square/leakcanary	265	12	39
square/okhttp	2,645	855	144
square/retrofit	1,349	232	109
xerces	5,456	853	31
Sum	**113,306**	**29,303**	**2,843**

3.3 Refactoring Detection and Classification

Since we aim to analyze a large set of commits for different systems, we used the Refactoring Miner tool (version 0.2.0) (61) to detect refactoring operations. Previous work observed a precision of Refactoring Miner equals 96.4% with low rates of false positives (61), which we confirmed in our validation process. Refactoring Miner detects 11 of the most recurring refactoring types investigated in the literature (12).

Validation of Refactoring Types We conducted a manual validation of the refactoring types identified by the Refactoring Miner tool. Such validation covered a random set of refactoring operations. After applying a statistical test with a confidence level of 95%, we observed a high precision of the tool for each refactoring type, with a median of 88.36% (excluding the *Rename Method* refactoring type). By applying the Grubb outlier test (62) (alpha = 0.05), we could not find any outliers, indicating that no refactoring type strongly

PUC-Rio - Certificação Digital Nº 1522494/CA

influences the median precision found. Thus, the obtained results represent a key factor to provide reliability to the results reported in this work.

Refactoring Tactic Classification. In our study, we manually analyzed a randomly selected sample of refactoring operations to classify them as *root-canal refactoring* or *floss refactoring.* Our goal is to investigate whether the refactoring tactics influences in the frequency and impacts of refactoring operations on the internal quality attributes. For this purpose, we assess whether the changes performed by developers during the refactoring are exclusively refactoring operations. We classify a transformation as *floss refactoring* when we identify additional changes in the code, such as the addition of methods or changes in a method body unrelated to refactoring operations. Otherwise, we classify the refactoring as *root-canal refactoring.* It is important to note that we carefully assigned the classification task to experienced developers, and provided sufficient time for completing the task.

PUC-Rio - Certificação Digital Nº 1522494/CA

3.4 Mapping Refactoring to Internal Quality Attributes

One could expect that each refactoring type is likely to positively affect a subset of quality attributes. Thus, we mapped each refactoring type to one or more internal quality attributes by relying on previous work (1, 63). Essentially, we infer the internal quality attributes related to each refactoring type addressed in our study from the Fowler's catalog (1). By doing that, we aim to gain a better understanding of how refactoring operations affect each related internal quality attribute.

Table 3.2 presents the association of each refactoring type with internal quality attributes, which are expected to be improved by each refactoring type. The first column lists each refactoring type. The second column presents the related internal quality attributes. According to the table, we expect that *Extract Superclass* has a direct impact on *size* and *inheritance.* Thus, if there are two different classes that implement similar functionalities, we apply *Extract Superclass* to create a single superclass that implements such functionalities. Consequently, we remove the duplicate code of both subclasses, thereby reducing their *size*, and increasing the *inheritance* depth.

3.5 Measuring Internal Quality Attributes

In Section 3.4, we mapped each refactoring type to its related internal quality attributes. After, we investigated the quality metrics for quantifying

Table 3.2: Expected effect on internal quality attributes

Refactoring Type	Related Internal Attributes
Extract Interface	Size, Inheritance
Extract Method	Size, Coupling
Extract Superclass	Size, Inheritance
Inline Method	Size, Coupling
Move Field	Coupling, Cohesion, Inheritance
Move Method	Coupling, Cohesion, Complexity, Inheritance
Pull Up Field	Size
Pull Up Method	Size
Push Down Field	Inheritance
Push Down Method	Inheritance
Rename Method	Size

each internal quality attribute. We defined a set composed by 25 quality metrics based on previous work (6, 30, 31, 39, 42, 45, 46). To select appropriate metrics per internal quality attribute, we applied the following criteria. First, we selected well-known metrics from the literature (6, 31, 39, 45, 46), including the CK suite (30) and McCabe's cyclomatic complexity (42). Second, we selected metrics that assess different properties of each internal quality attribute (30, 37, 47, 42). For example, LOC measures the number of lines of code, CBO measures the number of classes to which a class is coupled, and CC measures the complexity of a module's decision structure. Third, we selected metrics with evidence of accuracy in capturing and predicting problems in structural code quality. For instance, LOC helps in identifying classes having a poor design from the viewpoint of software developers (17).

Additionally, we have discarded metrics criticized in previous work (64), such as Lack of Cohesion in Methods (LCOM) (30), a metric that is non-normalized and has artificial outliers, which makes diffcult to compare two LCOM values (65). Table 3.3 presents the set of 25 metrics obtained through the aforementioned criteria. The first column lists the internal quality attributes. The second column presents the software metrics related to each internal quality attribute. The third column describes each software metric. Finally, the fourth column presents the behavior of each software metrics, i.e., when improves and when gets worse each software metrics. For instance, the complexity of a system improves while the value of the Cyclomatic Complexity (CC) (42) metric is lower, because of that, Cyclomatic Complexity (CC) improves when its value *decreases*. To compute each metric, we used the non-commercial license of the Understand tool[2].

3.6

[2]http://www.scitools.com

PUC-Rio - Certificação Digital Nº 1522494/CA

Table 3.3: Quality metrics used in this study

Attribute	Metric	Metric description	Improve when
Coupling	Coupling Between Objects (CBO) (30)	The number of classes to which a class is coupled	Decreases
	Fan-in (FANIN) (66)	The number of other classes that reference a class	Decreases
	Fan-out (FANOUT) (66)	The number of other classes referenced by a class	Decreases
	Coupling Intensity (CINT) (6)	The number of distinct operations called by the measured operation	Decreases
	Coupling Dispersion (CDISP) (6)	The number of classes in wich the operations called from the measured operation are defined by	Decreases
Cohesion	Lack of Cohesion of Methods 2 (LCOM2) (30)	Number of pairs of methods that do not share attributes, minus the number of pairs of methods that share attributes	Decreases
	Lack of Cohesion of Methods 3 (LCOM3) (45)	Number of disjoint components in the graph that represents each method as a node and the sharing of at least one attribute as an edge	Decreases
	Tight Class Cohesion (TCC) (39)	Ratio of number of similar method pairs to total number of method pairs in the class	Increases
Complexity	Cyclomatic Complexity (CC) (42)	Measure of the complexity of a module's decision structure	Decreases
	Weighted Method Count (WMC) (30)	The sum of Cyclomatic Complexity (42) of all methods declared in the given class	Decreases
	Essential Complexity (Evg) (42)	Measure of the degree to which a module contains unstructured constructs	Decreases
	Paths (NPATH) (67)	Number of unique paths though a body of code, not counting abnormal exits or gotos	Decreases
	Nesting (MaxNest)	Maximum nesting level of control constructs	Decreases
Inheritance	Depth of Inheritance Tree (DIT) (30)	The depth of a class as the number of its ancestor classes	Increases
	Number Of Children (NOC) (30)	The number of direct descendants (subclasses) of a class	Increases
	Base Classes (IFANIN) (9)	Number of immediate base classes	Increases
	Override Ratio (OR) (6)	The number of methods of the measured class that override methods from the base class, divided by the total number of methods in the class	Decreases
Size	Lines of Code (LOC) (31)	The number of lines of code excluding white spaces and comments	Decreases
	Lines with Comments (CLOC) (31)	Number of lines containing comment	Increases
	Statements (STMTC) (31)	Number of statements	Decreases
	Classes (CDL) (31)	Number of classes	Decreases
	Instance Variables (NIV) (31)	Number of instance variables	Decreases
	Instance Methods (NIM) (31)	Number of instance methods	Decreases
	Weight of Class (WOC) (6)	The sum of funtional public methods divided by the total number of public members	Decreases
	Number of Public Attributes (NOPA) (6)	The number of publics attributes of a class	Decreases

Selected Tools

Software quality metric collection tools. Structural quality attributes are measured using a set of software quality metrics. As mentioned in Section 3.5, all metrics values were collected using the Understand software with non-commercial license. Understand is a proprietary and paid application developed by SciTools. It is a static code analysis software tool and is mainly employed for calculation of source code metrics for software projects with large size or code bases. Understand supports a large number of programming languages, including Ada, C, the style sheet language CSS, ANSI C, and C++, C, Cobol, JavaScript, PHP, Delphi, Fortran, Java, JOVIAL, Python, HTML, and the hardware description language VHDL. The calculated metrics include complexity metrics, size and volume metrics, and other OO metrics such as Depth of Inheritance Tree (DIT) and Coupling Between Object Classes (CBO).

Refactoring detection tools. Refactoring has become a well-known technique for the software engineering community and developers. There are different tools to automatically detect the presence of a refactoring operation in a

PUC-Rio - Certificação Digital Nº 1522494/CA

pair of versions of a program. Studies of software refactoring require the analysis of a high number of commits to search refactorings operations, thereby causing the need for automatic detection and classification of refactorings. One of the tools for this purpose is Ref-Finder [3]. This tool is an Eclipse plugin that identifies refactorings using a template-based reconstruction technique. It expresses each refactoring type in terms of template logic queries and uses a logic programming engine to infer concrete refactoring situations. Ref-Finder currently supports 63 refactoring types described in Fowler's catalog (33). Ref-Finder seemed to be, in principle, a good candidate to support refactoring detection. However, there are some practical problems: (i) the precision was as low as 15% for most of the refactoring types (as observed in validation phase in the study of Cedrim et al. (18)), causing a substantial number of false positives; and (ii) the detection algorithm was inefficient when executed in a large dataset.

Because of these problems, Ref-Finder was discarded, and another tool was selected in this phase: Refactoring Miner (61) (version 0.2.0)[4]. This tool is more efficient than Ref-Finder since it was designed to be executed in a large dataset. Besides, its precision is 96.4% (as observed in our validation phase), leading to a very low rate of false positives. The only disadvantage of this tool is the number of refactoring types detected. While Ref-Finder detects 63 types of refactoring, Refactoring Miner detects 11 types. Fortunately, these 11 types were amongst the ones reported by Murphy-Hill as the most common refactoring types (11).

PUC-Rio - Certificação Digital Nº 1522494/CA

3.7 Threats to validity

We discuss threats to the study validity (71), with respective treatments, as follows.

Construct and Internal Validity. The threats to internal validity of this work concern the data collection procedure. Regarding the refactoring detection, we used the Refactoring Miner tool. To mitigate this threat, we selected random samples by refactoring type and performed a manual validation of it. The idea was to check whether we could have confidence on Refactoring Miner precision. We observed an average precision of Refactoring Miner equals to 96.4% with low rates of false positives. The set of metrics used in this study also represents a threat to validity, because it may not capture all relevant

[3]Available at https://github.com/SEAL-UCLA/Ref-Finder
[4]Available at https://github.com/tsantalis/RefactoringMiner

properties of the internal quality attributes. To mitigate this threat, we did not choose a random set of metrics. We chose metrics that assess different properties of each internal quality attribute (30, 37, 42, 47). Another criteria was used to chose our set of metrics that are detailed in Section 3.5. On the other hand, we analyzed 11 refactoring types detected by the Refactoring Miner tool. This amount represents a threat to validity because they do not cover all refactoring types reported by Fowler (1). We mitigated this threat by considering the 11 refactoring types that represent the most common types according to the literature (12).

To conduct our study, we selected a set of 23 software projects. The selection of the software projects represents a threat to validity and results extensibility. To mitigate this threat, our choice of the software projects is not random. To select the appropriate software projects to conduct our study, we applied the following criteria. First, we selected a set of software projects from GitHub repositories because we are concerned with the evolution of software projects regarding refactoring operations. Second, we selected open source projects to support the study replication and extension. Third, we selected Java software projects because it is a very popular programming language[5]. Fourth, we chosen highly popular projects, based on the number of stars received by the projects. Fifth, we selected software project with many contributors (2,843 in total) to have different points of view related to the refactoring operations. In addition, we computed our results for each project separately, and the results were similar in comparison to the general results. In other words, the results of each software project were similar to the results using the 23 software projects, this evidence that there is no a software project influencing the results.

Conclusion and External Validity. In this dissertation, we analyzed different tactics to perform refactoring operations, namely *root-canal refactoring* and *floss refactoring*. Due the lack of effective tools for classifying refactoring operations by tactics (*root-canal refactoring* and *floss refactoring*), we decided to perform a manual classification of 2,119 refactoring operations (about 7% of all refactoring operations). This manual classification represents a threat to validity. Then, two researchers performed double-checking for a random subset of the classified refactoring operations and obtained a high precision (>95%) in the classification.

[5]https://www.tiobe.com/tiobe-index/

PUC-Rio - Certificação Digital Nº 1522494/CA

3.8
Final Remarks

This chapter presented the study design of the work. First, we discussed our goals and designed our research questions to understand the effect of refactoring and re-refactoring operation on internal quality attributes. For this purpose, we select a set of 23 open source project of GitHub repositories and we used the Refactoring Miner tool to detect refactoring operations over our project sample. Then, we mapped refactoring types with internal quality attributes in order to understanding of how refactoring operations affect each related internal quality attribute. After mapping each refactoring type to its related internal quality attributes, we investigated the quality metrics for quantifying each internal quality attribute. With the set of metrics selected, we investigate the necessary tools to collect the value of each metrics. Finally, we discuss threats to the study validity, with respective treatments. The next chapter presents and discusses the results of the first research question.

PUC-Rio - Certificação Digital Nº 1522494/CA

4
(Re-)Refactoring Affecting Critical Elements

In this chapter, we assess whether refactoring and re-refactoring operations are often applied to code elements with critical internal quality attributes. We aim to observe if developers target their refactoring effort mostly on code elements that require structural quality improvement.

4.1
Overall Results

To address the RQ1 (*Are refactoring operations often applied to code elements with critical internal quality attributes?*), we have analyzed the internal quality attributes in code elements affected by refactoring operations. In particular, we computed the number of critical internal quality attributes associated with code elements being refactored. Algorithm 1 illustrates this process. The algorithm iterates over the set R of refactoring operations (line 2). For each refactoring operation r, the code elements affected by r are iterated (line 4). For each code element e, we compute the number of critical internal quality attributes (line 5). Finally, we sum the total number of critical internal quality attributes for the current code elements affected by a refactoring operation (line 6). The total number of critical internal attributes is used to classify the refactoring operation as $None$ if the number equals zero, $Single$ if the number equals one, and $Multiple$ if the number is greater than one (line 7). We also compute the number of critical internal quality attributes associated with code elements being re-refactored. In other words, we run Algorithm 1 on the set of refactoring operations applied in a code element that was refactored before, called re-refactoring operations.

Algorithm 1 Classifying refactoring operations

```
0: Initilize $\mathcal{R}$ = refactoring operations
0: **for** $r \in \mathcal{R}$ **do**
0:     Initilize $t$ = total of critical internal attributes for $r$
0:     Initilize $\mathcal{E}$ = code elements affected by $r$
0:     **for** $e \in \mathcal{E}$ **do**
0:         Initilize $c$ = number of critical internal attributes of $e$
0:         $t$ += $c$
0: Classify $r$ as $None$, $Single$, or $Multiple$ =0
```

PUC-Rio - Certificação Digital Nº 1522494/CA

PUC-Rio - Certificação Digital Nº 1522494/CA

Table 4.1: (Re-)Refactoring and elements with critical attributes

Experimental Groups	**Refactoring Tactic**	**Total**	**Critical Internal Attribute**		
			None	**Some**	
				Single	**Multiple**
Sample	**Root-Canal**	576	7 (1,72%)	17 (2.95%)	552 (95,83%)
	Floss	1,543	4 (0.26%)	13 (0.84%)	1,526 (98.90%)
All-refactorings	**Any**	29,303	1,570 (5,36%)	4,458 (15,21%)	23,275 (79,43%)
Re-Refactoring	**Any**	14,521	20 (0.14%)	33 (0.23%)	14,468 (99.64%)

Overall Analysis Regardless the Refactoring Type. After computing the number of critical internal quality attributes present in code elements affected by refactoring, we are able to analyze the results. Thus, Table 4.1 presents the number of refactoring operations applied on code elements with critical internal quality attributes. The first column lists each experimental group: *Sample*, composed of the refactoring operations that we manually classified (Section 3.3), *Re-refactoring* composed by the re-refactoring operations, and *All-refactorings*, composed by all the refactoring operations collected from the target projects. *Sample* and *Re-refactoring* are subsets of *All-refactorings* set. The second column indicates the tactics for conducting refactoring operations, namely *Root-canal refactoring* and *Floss refactoring*. Note that the *Re-refactoring* and *All-refactorings* groups include re-refactoring and refactoring operations regardless the refactoring tactic applied by developers. The third column provides the total number of refactoring operations per refactoring tactic and experimental group. The fourth column named *None* presents the number and percentage of operations applied to code elements without critical internal quality attributes. The fifth and sixth columns present the number and percentage of operations applied to code elements with at least one critical internal quality attribute, i.e., the *Some* columns (see Section 3.1). We divided these operations into two, namely: *Single*, i.e., the ones with exactly one critical internal attribute, and *Multiple*, i.e., the ones with two or more critical internal attributes.

The data in Table 4.1 suggest that developers tend to apply refactoring operations most frequently to code elements with at least one critical internal quality attribute, as indicated in the *Some* columns. By summing the refactoring operations with a *Single* and *Multiple* critical internal quality attributes, the total frequency represents 94.64% (27,733) of the refactoring operations. This result takes into consideration the *All-refactorings* line regarding all analyzed refactoring operations. Moreover, most of these refactoring operations are applied to *Multiple* critical internal quality attributes, i.e., two or more attributes. It represents 79.43% (23,275) of the refactoring operations, against only 15.21% (4,458) for code elements with a *Single* critical internal quality attribute.

These observations seem to confirm that critical internal quality attributes tend to indicate code elements that require refactoring operations. Thus, developers seem to be, in principle, targeting these code elements to improve their structural quality through refactoring. Also, they contradict the finding of Bavota et al. (17), which suggest that there is no clear relationship between quality metrics and refactoring operations. One of the reasons for not having a clear relationship is that Bavota et al. (17) considers the improvement of an internal quality attribute if the same metric related to that attribute often improves along all refactoring operations. In contrast, our study analyzes multiple metrics for the same internal quality attribute, which may have increased the chances of capturing the positive impact of refactoring operations on internal quality attributes. Each type of refactoring may prove a different impact on the same internal quality attribute whenever it is instantiated to a particular context.

From data in Table 4.1, we can draw conclusions on the frequency of refactoring operations per refactoring tactic. By comparing *Root-canal refactoring* and *Floss refactoring*, we observe no relevant differences between the percentages of operations per number of critical attributes. As in the aforementioned overall analysis, most of the operations are applied to code elements with at least one critical attribute. The percentages are 98.78% and 99.74% for *root-canal refactoring* and *floss refactoring*, respectively. Thus, regardless the aim of developers on improving or not the structural program quality, they tend to apply refactoring operations in elements with many critical attributes.

On the other hand, the third row of Table 4.1 shows the number of re-refactoring operations applied or not on code elements with critical internal quality attributes. The total of refactoring operations affecting any code elements that had already been affected by an earlier refactoring operation, which we call as re-refactoring operations, is 14,521 (49.55% of the total number of refactoring operations). This means that almost a half of the code elements that are refactored anytime are refactored again in the future. This observation confirms the literature assumption which suggests that multiple refactoring operations should be applied to gradually improve the code structural quality. In addition, similarly to the analysis of refactoring operations, we ob-

PUC-Rio - Certificação Digital Nº 1522494/CA

serve that most re-refactoring operations are applied to code elements with *Multiple* critical internal quality attributes. It represents 99.63% (14,468) of the re-refactoring operations, against only 0.37% (53) for code elements with a *Single* or *None* critical internal quality attribute.

We can observe that most of the refactoring operations are applied in code elements with more than one critical attribute (93.45% in average). In other words, in the vast majority of cases, when developers apply a refactoring operation, the refactored code has critical internal attributes, regardless of refactoring tactic and whether the code element was refactored in the past (re-refactoring). Moreover, the percentage of re-refactoring operations applied in code elements without critical internal quality attributes (None, 0.14%) and with only one critical internal quality attribute (Single, 0.23%), being the least percentages of the samples. This means that when a refactoring operation is applied to a code element that has been refactored in the past, in a few cases, those code elements have no critical internal quality attributes or have only one critical internal quality attribute.

PUC-Rio - Certificação Digital Nº 1522494/CA

Analysis per Refactoring Type. Table 4.2 presents the number of refactoring operations applied on code elements with critical internal quality attributes per refactoring type. The first column lists the 11 refactoring types under analysis. The next columns of this table follow a similar structure of the columns in Table 4.1. The second column provides the total number of refactoring operations per refactoring type. The third column presents the number and percentage of operations applied to code elements without critical internal attributes. The fourth and fifth columns present the number and percentage of operations applied to code elements with at least one critical internal attribute. Similarly to Table 4.1, we divided these operations into two, namely: *Single*, i.e., the ones with exactly one critical internal attribute, and *Multiple*, i.e., the ones with two or more critical internal attributes.

Data of Table 4.2 lead to some relevant observations discussed as follows. First, 8 out of the 11 refactoring types rarely affect code elements without critical internal quality attributes, as indicated by the *None* column. In fact, only *Extract Interface* and *Rename Method* affect more than approximately 10% of code elements without critical attributes. This result is expected for *Rename Method* refactorings since it does not alter the structure of the source code. Also, we find that refactorings of type *Rename Method* are applied as a consequence of other refactoring operations. For instance, a developer moves a method from one class to another. Consequently, they change the name of the method to represent the new responsibility in the target class. Overall, we

Table 4.2: Refactoring types and elements with critical attributes

Refactoring Type	Total	Critical Internal Attribute		
		None	Some	
			Single	Multiple
Push down Field	78	0 (0%)	0 (0%)	78 (100%)
Push down Method	114	0 (0%)	1 (0.88%)	113 (99.12%)
Inline Method	1,525	1 (0.07%)	1 (0.07%)	1,523 (99.86%)
Pull up Field	465	2 (0.43%)	0 (0%)	463 (99.57%)
Move Field	4,355	0 (0%)	6 (0.14%)	4,349 (99.86%)
Pull up Method	629	2 (0.32%)	5 (0.79%)	622 (98.89%)
Move Method	1,404	4 (0.28%)	10 (0.71%)	1,390 (99.01%)
Extract Method	7,513	14 (0.19%)	14 (0.19%)	7,485 (99.62%)
Extract Superclass	341	16 (4.69%)	10 (2.93%)	315 (92.38%)
Extract Interface	133	13 (9.78%)	12 (9.02%)	108 (81.20%)
Rename Method	12,746	1,518 (11.91%)	4,399 (34.51%)	6,829 (53.58%)

conclude that developers should carefully apply refactoring operations, since most refactoring types may negatively affect multiple critical internal quality attributes together.

Finding 1. *Most of the refactoring operations (93.45% in average) are very often applied to code elements with at least one critical internal quality attribute. This observation is independent of the refactoring tactic. It also applies to re-refactoring operations.*

4.2 Contradicting Findings and Implications: Discussion

In Section 4.1 we analyze the frequency of refactoring and re-refactoring operations applied to code elements with critical internal quality attributes. This analysis aims at revealing whether refactoring operations often target structurally critical elements in the source code. If so, this means that refactoring indeed has the potential to improve internal quality attributes. As a result, we find that developers apply more than 93% of the refactoring and re-refactoring operations to code elements with at least one critical internal quality attribute. In addition, more than 79% of the refactoring operations are applied in code elements with more than one critical internal quality attribute.

PUC-Rio - Certificação Digital Nº 1522494/CA

PUC-Rio - Certificação Digital Nº 1522494/CA

We then conclude that refactoring operations mostly target on code elements that indeed require structural quality improvement.

Our results have different implications. First, developers should carefully apply those operations, even if a code element was refactored before (re-refactoring), since each refactoring operation may negatively affect the code structural quality in several ways. Second, after manually classifying the refactoring operations per tactic, we have found that most operations are floss refactoring (72.82%), i.e., the major concern of developers when refactoring is not strictly or explicitly improving the code structural quality alone. Thus, developers should pay special attention when changing the source code to add new functionalities or fixing bugs after refactoring the source code, for instance, since these changes may have a negative impact on the code structural quality.

Our finding partially contradicts the results of a recent study (17), which suggest that there is no clear relationship between quality metrics and refactoring operations. In order to understand the causes for both study findings to differ, we compared our study with the previous one. Table 4.3 compares the design of both studies, which may lead to the different findings. The first column presents the studies. The second column shows the refactoring detection tool used by each study. The third column presents the releases that were used to detect refactoring operations. The fourth and fifth columns show the number of internal quality attributes and quality metrics analyzed in each study, respectively. The sixth column presents the exclusive metrics, i.e., the number of quality metrics used in each study that was not used in the other. Finally, the seventh column shows the number of software projects analyzed in each study.

Table 4.3: Comparison with the study design (17)

Study	**Refactoring Detection Tool**	**Refactoring Computation**	**Internal Attributes**	**Quality Metrics**	**Exclusive Metrics**	**Projects**
Previous (17)	Ref-Finder	Between major releases	5	11	6	3
Ours	Refactoring Miner	Between commits	5	25	20	23

By relying on Table 4.3, we observed several differences between both studies that may produce different results. For instance, to detect each refactoring operation, we used the Refactoring Miner tool (version 0.2.0) (61). Previous work observed a precision of Refactoring Miner equals 96.4% with low rates of false positives (61). Indeed, to detect each refactoring operation, bavota2015experimental used the Ref-Finder tool. The precision of Ref-Finder tool was as low as 15% for most of the refactoring types, as observed in validation phase in the study of Cedrim et al. (18), causing a substantial num-

Table 4.4: Quality metrics grouping of Bavota et al. (17) study

Internal Attribute	**Quality Metrics**
Cohesion	Lack of COhesion of Methods (LCOM) (30)
	Conceptual Cohesion of Classes (C3) (68)
Coupling	Response for a Class (RFC) (30)
	Coupling Between Object (CBO) (30)
	Conceptual Coupling Between Classes (CCBC) (69)
Complexity	Weighted Methods per Class (WMC) (30)
Inheritance	Depth of Inheritance Tree (DIT) (30)
	Number Of Children (NOC) (30)
	Number of Operations Added by a subclass (NOA) (31)
	Number of Operations Overridden by a subclass (NOO) (31)
Size	Lines of Code (LOC) (31)

ber of false positives. Furthermore, Bavota et al. (17) analyze the occurrence of refactoring operations only in major versions, being able to leave relevant information in intermediate commits. To the contrary, we analyze all the commits to detect and investigate the refactoring operations.

The fourth column in Table 4.3 presents the number of internal quality attributes analyzed by both studies. Since Bavota et al. (17) do not explicitly study internal quality attributes, we grouping the quality metrics used by them into internal quality attributes. The Table 4.4 presents this grouping. The first column shows the internal quality attributes. The second columns presents the quality metrics associated to each internal quality attributes. To archive this information, we used the same mapping criteria of Section 3.5. After grouping the quality metrics, we observe that both studies used the same internal quality attributes.

The fifth column presents the number of quality metrics analyzed in each study and the sixth column shows the exclusive metrics. We used a set of 25 quality metrics chosen using several criteria (See Section 3.5), against only 11 quality metrics used by Bavota et al. (17). On the other hand, Bavota et al. (17) uses the Lack of Cohesion of Methods (LCOM) metric (30), which was discarded in our study. There are evidences that LCOM is not an appropriate metric to capture and measure cohesion 64,65. First, LCOM is not normalized and has outliers, so, it is difficult to compare two values of LCOM. Second, LCOM equals 0 may be meaningless, since it does not exactly indicate “no lack of cohesion" (65). These aspects represent another difference that may have led to different results.

Finally, as shown in the last column of Table 4.3, we analyzed 23 software projects, against only 3 projects analyzed by Bavota et al. (17). We analyze the 3 projects analyzed by Bavota et al. (17) plus 20 more projects. The software projects selection may have led to different results.

PUC-Rio - Certificação Digital Nº 1522494/CA

We observed several differences related to the software project under analysis. First, our study has projects with additional domains. Second, both studies have different developers communities with diverse views about the application of refactoring operations and code structural quality. All these aspects can lead to different insights of generalization and results of the studies (70). Thus, another implication of our study is that more empirical studies, performed by independent researchers, need to be conducted in this field. The analyzes present in the next chapter also aims to reinforce either our findings or Bavota et al.'s findings.

4.3 Final Remarks

In this chapter, we analyze if refactoring and re-refactoring operations are often applied to code elements with critical internal quality attributes. We aim to observe if developers target their refactoring effort mostly on code elements that require structural quality improvement. We assume that an internal quality attribute is critical if at least one metric used to quantify the internal attribute has a critical value (Section 3.1). As a result, we observe that most of the refactoring operations (93.45% in average) are very often applied to code elements with at least one critical internal quality attribute. This observation is also valid for root-canal refactoring (98.78%), floss refactoring (99.74%) and re-refactoring (99.86%). This finding led us to investigate the effects of refactoring operations on internal quality attributes. The next chapter presents and discusses the results of RQ2.

PUC-Rio - Certificação Digital Nº 1522494/CA

5
Effects of Refactorings on Internal Quality Attributes

In this chapter, we assess the impact of refactoring operations on different internal quality attributes. While answering this research question, we aim to understand if developers improve at least one internal quality attribute after refactoring code elements. We are also interested in analyzing if developers affect most of the internal attributes when applying a refactoring operation. Through this assessment, we answer our RQ2 (*What is the impact of refactoring operations on internal quality attributes?*).

5.1
Answering RQ2 with Two Approaches

PUC-Rio - Certificação Digital Nº 1522494/CA

After applying a refactoring operation to a code element, each of the metrics that capture each internal quality attribute may present three different behaviors, which are captured in Table 5.1. The value of a given metric may increase, decrease, or remain unaffected. Also, multiple metrics help to capture a single internal quality attribute. Thus, to understand the impact of refactoring operations on internal quality attributes, we defined a notation for the metric behavior. Table 5.1 presents the notation used to categorize the metric behavior. This notation aims to support the discussion of study results in Sections 5.2 and 5.3.

Table 5.1: Notation of Software Metric Behavior

Symbol	**Description**
↑	The metric improves
↓	The metric worsens
_	The metric remains unaffected

We address RQ2 by analyzing the behavior of internal quality attributes in two different approaches: *Most Metrics* and *At Least One Metric*. The *Most Metrics* approach tries to identify whether refactoring operations affect several properties (quality metric) of an internal quality attribute, i.e., when the refactoring operation aims to positively affect the entire internal quality attribute. On the other hand, the *At Least One Metric* approach aims at assessing if refactoring positively affects at least one property (quality metric) of an internal quality attribute, i.e., when a simple refactoring operation aims to improve a specific property of the attribute, possibly as a step for future improvement of the entire internal quality attribute. Sections 5.2 and 5.3 describe each approach.

5.2 The Most Metrics Approach

Table 5.2 presents the impact of refactoring operations per type using the *Most Metrics* approach. The first column lists the 11 refactoring types under analysis. The second column provides the total number of refactoring operations per refactoring type. The remaining columns concern the five internal quality attributes investigated in this study. For each column, the table presents the predominant behavior for the respective refactoring type. For this purpose, we use the notation described in Table 5.1. These columns also provide the percentage of refactoring operations categorized in the predominant behavior for the respective internal quality attribute. Each cell highlighted in gray indicates that a given internal quality attribute (captured by the corresponding column of the cell) is expected to improve after applying a given refactoring type (captured by the corresponding row of the cell). The gray highlight on each cell was assigned by the mapping of refactoring to internal quality attributes described in Table 3.2. These gray cells represent the cases of internal quality attributes expected to be improved by the corresponding refactoring types.

PUC-Rio - Certificação Digital Nº 1522494/CA

Table 5.2: Refactoring Effect Using the *Most Metrics* Approach

Refactoring Type	**Total**	**Cohesion**	**Coupling**	**Complexity**	**Inheritance**	**Size**
Extract Superclass	341	↑ 51.32%	— 53.67%	— 60.12%	— 54.25%	↑ 73.02%
Extract Interface	133	— 83.46%	— 69.92%	— 94.74%	↑ 64.66%	— 42.11%
Move Field	4,355	↑ 63.31%	— 55.57%	— 88.4%	— 90.13%	— 50.95%
Inline Method	1,525	↑ 58.3%	↓ 39.87%	— 48.92%	— 92.92%	↑ 56.59%
Push down Field	78	↓ 47.44%	↑ 41.03%	— 87.18%	— 97.44%	↑ 46.15%
Extract Method	7,513	↓ 59.03%	↓ 45.77%	↑ 44.95%	— 93.81%	↓ 58.53%
Move Method	1,404	↓ 46.44%	↓ 41.17%	— 68.87%	— 81.05%	↑ 43.73%
Push down Method	114	↓ 42.11%	↑ 44.74%	— 59.65%	— 89.47%	↓ 58.77%
Pull up Method	629	↓ 43.88%	↓ 67.89%	— 69.16%	— 85.06%	↓ 52.94%
Pull up Field	465	↓ 59.35%	↓ 66.45%	— 68.6%	— 81.94%	↓ 63.87%
Rename Method	12,746	— 100%	— 82.93%	— 97.98%	— 100%	— 86.11%

PUC-Rio - Certificação Digital Nº 1522494/CA

Based on Table 5.2, we observe that, for *Extract Superclass*, *Extract Interface*, *Move Field*, *Inline Method*, and *Push down Field*, one or more internal quality attributes improved. That is, these refactoring types have internal quality attributes improving more than worsening. Furthermore, exactly one related internal quality attribute to a given refactoring type has improved for each type. Overall, five out of 11 (45.45%) of the refactoring types have improved for one or more internal quality attributes. On the other hand, for the *Extract Method*, *Move Method*, *Push down Method*, *Pull up Method*, and *Pull up Field*, one or more internal quality attributes worsens. That is, these refactoring types have internal quality attributes worsening more than improving. Overall, five out of 11 (45.45%) of the refactoring types have worsens for one or more internal quality attributes. Finally, *Rename Method* unalters all the internal quality attributes.

Finding 2. *For some refactoring types, the internal quality attributes tend to improve or remain unaffected. This observation is valid when considering most metrics per internal quality attribute.*

Based on Table 5.2, we compute the total internal quality attributes that improve, worsen and remain unaffected for each approach. This computation is performed over the related internal quality attributes (gray cells). Concerning the *Most Metrics* approach, the amount of improved internal quality attributes is equal to four (20%); thus, we observed some positive impact of refactoring operations on certain internal quality attributes. However, the amount of worsening internal attributes is equal to seven (35%), and nine (45%) remain unaffected. In fact, the number of worsened internal quality attributes surpasses the number of improved attributes. However, it represents only a half of the total number of improved attributes summed with unaffected internal quality attributes. These results suggest we should study if less strict approaches (e.g., the *At Least One Metric* approach) may better capture the positive impact of refactoring operations. Section 5.3 aims at analyzing the internal quality attributes using the *At Least One Metric* approach.

The study of Bavota et al. (17) suggests there is no relationship between refactoring operations and metrics that capture key quality attributes, such as *cohesion*. Table 5.2 shows that *cohesion* is expected to be improved by *Move Method* and *Move Field* refactoring types. We observed that *Move Field* improves cohesion while *Move Method* has an opposite effect, using the *Most Metrics* approach. One of the possible reasons for this diverging results (with respect to Bavota et al. (17)) is that our study used a different set of metrics to capture *cohesion*. In their study, they only analyze the LCOM (30) metric, that according to Henderson-Sellers (64) is not an appropriate metric to capture and quantify *cohesion*. According to Henderson-Sellers (64), the LCOM is not a non-normalized metric and has outliers. Consequently, it is difficult to compare two values of LCOM. On the other hand, LCOM equal to 0 is sometimes meaningless, that is, is does not exactly indicates "no lack of cohesion", as observed by some studies (64, 65).

5.3 The At Least One Metric Approach

Table 5.3 presents the impact of refactoring operations per type using the *At Least One Metric* approach. The structure of this table is similar to Table 5.2. Using this approach, when a particular metric used to quantify the internal quality attribute improves, it means that the internal quality attribute also improves. An analysis of Table 5.3 reveals that, in most cases, refactoring operations tend to improve the related internal quality attributes. For instance, the *Extract Superclass* refactoring type is expected to improve two internal quality attributes, namely *inheritance* and *size* (see Table 3.2). Table 5.3 shows that this refactoring type improves *inheritance* in 92.67% of the cases, and *size* in 81.52%. Besides that, it also improves *cohesion* and *coupling*.

Table 5.3: Refactoring Effect Using the *At Least One Metric* Approach

Refactoring Type	Total	Cohesion	Coupling	Complexity	Inheritance	Size
Extract Superclass	341	↑ 51.32%	↑ 48.68%	_ 60.12%	↑ 92.67%	↑ 81.52%
Inline Method	1,525	↑ 58.3%	↑ 77.64%	_ 48.07%	_ 91.48%	↑ 88.98%
Extract Method	7,513	↓ 59.03%	↑ 71.4%	↑ 47.03%	_ 93.09%	↑ 85.81%
Move Method	1,404	↓ 46.44%	↑ 48.01%	_ 68.87%	_ 74.29%	↑ 82.55%
Push down Field	78	↓ 47.44%	↑ 53.85%	_ 87.18%	_ 94.87%	↑ 79.49%
Push down Method	114	↓ 42.11%	↑ 77.19%	_ 59.65%	_ 85.09%	↑ 78.07%
Extract Interface	133	_ 83.46%	_ 63.16%	_ 94.74%	↑ 68.42%	_ 39.85%
Move Field	4,355	↑ 63.31%	_ 49%	_ 88.4%	_ 87.83%	_ 48.45%
Pull up Field	465	↓ 59.35%	↓ 66.45%	_ 68.6%	_ 70.97%	↑ 81.29%
Pull up Method	629	↓ 43.88%	↓ 67.89%	_ 69.16%	_ 83.47%	↑ 82.99%
Rename Method	12,746	_ 100%	_ 80.63%	_ 97.98%	_ 100%	_ 85.74%

The eight (72.73%) first refactoring types in Table 5.3 show a predominant behavior of improvements. This means that using the *At Least One Metric* approach, most refactoring types tend to improve rather than worsen the in-

PUC-Rio - Certificação Digital Nº 1522494/CA

ternal quality attributes. Similar to the *Most Metrics* approach, the *Rename Method* refactoring type does not alter internal quality attributes. Finally, *Pull up Field* and *Pull up Method* tend to worsen rather than improve the internal quality attributes. This observation is also valid for *Most Metrics* approach. Consequently, we should take some care by applying this refactoring types since it negatively affects most internal quality attributes.

Based on Table 5.3, we compute the total internal quality attributes that improve, worsen and remain unaffected for each approach. With respect to *At Least One Metric* approach, the amount of related internal quality attributes (grey cells) is equals to 11 (55%). The amount of worsen internal attributes is equals to one (5%), while eight (40%) attributes remain unaffected. As expected, the number of improved internal quality attributes when using a less strict approach has increased in 35% when compared to the *Most Metrics* approach. In turn, the number of worsened attributes has declined in 30% when compared to the more strict approach.

Finding 3. *The refactoring operations tend to improve or unalter the internal quality attributes. This observation is valid when considering at least one metric per internal quality attribute.*

The study of Bavota et al. (17) reports a low relationship between refactoring operations and metrics that capture the *coupling* attribute. In contrast, our study shows that refactoring operations in most of the cases improve the *coupling* attribute. For instance, in our work, we assume that the *coupling* attribute is related to the following refactoring types: *Inline Method*, *Extract Method*, *Move Method*, and *Move Field*. Using the *At Least One Metric* approach, we confirmed that *Inline Method*, *Extract Method* and *Move Method* refactorings often improve *coupling*, while *Move Field* refactorings often does not affect *coupling*. One of the possible reasons for the difference with Bavota's study is the different set of metrics employed to capture *coupling*. Bavota et al. (17) analyze three metrics that capture *coupling*, while we used five metrics (see Table 3.3).

5.4 Refactoring Recommendations

In Chapter 4 we assess whether refactoring operations are often applied to code elements with critical internal quality attributes. As a result, we observe that most of the refactoring operations (93.45% in average) are applied to code elements with at least one critical internal quality attribute. In this chapter we assess the impact of refactoring operations on different internal

PUC-Rio - Certificação Digital Nº 1522494/CA

quality attributes, aimed at providing recommendations about the use of each refactoring type. These recommendations can be tailored to particular developers' concerns with respect to internal quality attributes.

Table 5.4 presents recommendations for developers who are applying each refactoring type. The first column lists the 11 refactoring types under analysis. The second column presents whether each refactoring type is often applied to code elements with critical internal quality attributes. In other words, the second column summarizes the results of the RQ1 (Section 4). The third and fourth columns show the effect of each refactoring type on the internal attributes using the *Most Metrics* and *At Least One Metrics* approaches respectively. That is the results of the RQ2 (Section 5.1). Finally, the fifth column presents the recommendations for applying each refactoring type.

The data in Table 5.4 provide recommendations for applying refactoring operations when developers attempt to improve the structural quality by improving internal attributes. We can observe several interesting recommendations. For instance, the *cohesion* internal quality attribute can be improved by three refactoring types, (i) *Extract Superclass*, (ii) *Move Field*, and (iii) *Inline Method*. This means that developers should first consider the application of these refactoring types, when they want to improve *cohesion*. On the other hand, although *cohesion* is related to *Move Method* refactoring type, we can observe that the *cohesion* is often worsened by *Move Method*. We believe that the result can be produced because the moved method could be very related to the attributes and other methods of the source class. This recommendation serves as a warning for developers applying the *Move Method* refactoring. Developers should be more careful when performing this refactoring; otherwise, they have a high chance of reducing the *cohesion* of either the source of the target class.

We found that *Push down Field* and *Push down Method* refactoring types are opportunities to improve the *coupling* internal quality attribute. On the other hand, the *complexity* and *inheritance* internal attributes only were improved by *Extract Method* and *Extract Interface*, respectively. This means that when there is interest in ameliorating *complexity*, a refactoring operation of type *Extract Method* should be considered. Furthermore, the application of *Extract Interface* is likely to lead to improvements in *inheritance*. We can note that *Pull up Method* refactoring type does not improve any internal quality attribute and negatively affects *cohesion*, *coupling* and some properties of *size*. Therefore, care must be taken when applying this type of refactoring.

In summary, we assess whether refactoring operations are often applied

PUC-Rio - Certificação Digital Nº 1522494/CA

PUC-Rio - Certificação Digital Nº 1522494/CA

Table 5.4: Refactoring recommendations based on attributes.

Refactoring Type	**Occurrence with internal attributes (RQ1)**	**internal attributes (Most Metrics) (RQ2)**	**ng effects on internal attributes (At Least One) (RQ2)**	**Recommendation**
Extract Method	It rarely affects code elements without critical internal quality attributes. In most cases, Extract Method affects code elements with multiple critical internal quality attributes.	It unalters inheritance. On the other hand, it tends to improve complexity and worsens cohesion, coupling and size.	It unalters the inheritance. On the other hand, it tends to improve coupling, complexity and size, and worsen cohesion.	It can be freely used, without major concerns about inheritance. In addition, should be considered to improve the complexity internal quality attributes. Care should be taken to the cohesion attribute since this refactoring type tends to worsen cohesion.
Extract Interface	It affects a considerable number of code elements without critical attributes, with respect to the other refactoring types.	It unalters all attributes except inheritance, which tends to improve.	It unalters all attributes except inheritance, which tends to improve it.	It can be freely used, without major concerns about the code structural quality. In addition, should be considered to improve the inheritance internal quality attribute.
Extract Superclass	It affects a minimum but also considerable number code elements without critical attributes, with respect to the other refactoring types.	It unalters all attributes except cohesion and size, which tends to improve.	It unalters all attributes except cohesion and size, which tends to improve it.	It can be freely used, without major concerns about the code structural quality. In addition, should be considered to improve the cohesion and size internal quality attributes.
Inline Method	It rarely affects code elements without critical internal quality attributes. In most cases, Inline Method affects code elements with multiple critical internal quality attributes.	It unalters all attributes except cohesion and size, which tends to improve.	It improve cohesion, coupling and size. On the other hand, it unalters for complexity and inharitance.	It can be freely used to improve the cohesion and size internal quality attribute. It also potentially improves some properties of coupling.
Move Field	It never affects code elements without critical internal quality attributes. In most cases, Move Field affects code elements with multiple critical internal quality attributes.	It unalters all attributes except cohesion, which tends to improve.	It unalters all attributes except cohesion, which tends to improve it.	It can be freely used, without major concerns about the improvement of all internal attributes except cohesion. This refactoring type should be considered to improve the cohesion internal quality attributes.
Move Method	It rarely affects code elements without critical internal quality attributes. In most cases, Move Method affects code elements with multiple critical internal quality attributes.	It improves size and remains unaffected,both inheritance and complexity. However, it worsens cohesion and coupling.	It improves size and coupling. In addition, it unalters both inheritance and complexity. However, it worsens cohesion.	It can be freely used to improve the size internal quality attribute. Care should be taken to the cohesion attribute since this refactoring type,tends to worsen cohesion.
Rename Method	It affects a considerable number of elements without critical attributes, with respect to the other refactoring types.	It unalters all attributes.	It unalters all attributes.	It can be freely used, without major concerns about the code structural quality
Pull up Field	It rarely affects code elements without critical internal quality attributes. In most cases, Pull up Field affects code elements with multiple critical internal quality attributes.	It worsens cohesion, coupling, and size. However, it remains unaffected the other internal attributes.	It improves size only. In addition, it unalters the other internal attributes. However, it worsen cohesion and coupling.	It can be freely used without oncerns about complexity and inheritance. We should take some care by applying this refactoring type since it negatively cohesion and coupling, in addition to the negative impact of some properties of size.
Pull up Method	It rarely affects code elements without critical internal quality attributes. In most cases, Pull up Method affects code elements with multiple critical internal quality attributes.	It worsens cohesion, coupling, and size. However, it remains unaffected the other internal attributes.	It improves size only. In addition, it unalters the other internal attributes. However, it worsens cohesion and coupling.	It can be freely used without concerns about complexity and inheritance. We should take some care by applying this refactoring type since it negatively impacts cohesion and coupling, in addition to the negative impact of some properties of size.
Push down Field	It always affects code elements with multiple critical internal quality attributes.	It improves coupling and size, worsens cohesion, and unalters the other attributes	It improves coupling and size, worsens cohesion, and unalters the other attributes	It can be freely used, without major concerns about complexity and inheritance. In addition, it should be considered to improve coupling and size. We should take some care by applying this refactoring type when developers are concerned with improving cohesion.
Push down Method	It never affects code elements without critical internal quality attributes. In most cases, Push down Method affects code elements with multiple critical internal quality attributes.	It improves coupling and worsens both cohesion and size. In addition, it unalters both complexity and inheritance	It improves both coupling and size, worsens cohesion, and unalters complexity and inheritance	It can be freely used without concerns about complexity and inheritance. It also can be used to improve coupling and some properties of size. However, we should take some care by applying this refactoring type since it negatively affects cohesion

to code elements with critical internal quality attributes (Chapter 4). On the other hand, we assess the impact of refactoring operations on different internal quality attributes (Chapter 5). These analyses help us to produce recommendations about the use of each refactoring type, depending on the developers concerns about internal quality attributes (Table 5.4). These recommendations can help developers by suggesting refactoring types for improving specific internal quality attributes. Additionally, these recommendations can be used by tools to suggest or alert the application of refactoring operations. These suggestions or warnings can guide the improvement of code structural quality by inducing developers to positively impact internal quality attributes.

By analyzing Table 5.4, we observe that the *At Least One Metric* approach is not very less strict than the *Most Metrics* approach. That is, even by considering the improvement of an internal quality attribute by just one improve metric, we observe a slight difference of results when compared to the *Most Metrics* approach, which is our most strict approach. Indeed, for *Extract Interface*, *Extract Superclass*, *Move Field*, *Rename Method* and *Push down Field* nothing has changed. In addition, for *Inline Method* and *Pull up Field*, we observe that only one attribute has changed its behavior from one approach to another.

PUC-Rio - Certificação Digital Nº 1522494/CA

5.5 Root-canal versus Floss Refactoring

This section discusses the impact of *root-canal refactoring* and *floss refactoring* using both approaches. Table 5.5 presents the general results for the *Most Metrics* approach. The first column lists each metric behavior. The second and third columns present data regarding *root-canal refactoring*. Each cell presents the number and percentage of internal quality attributes in the case of related and all attributes, respectively. The fourth and fifth columns present data regarding *floss refactoring*. Similarly to the *root-canal refactoring*, each cell presents the number and percentage of internal quality attributes in the case of the related and all attributes, respectively.

Data in Table 5.5 point out interesting observations for the *Most Metrics* approach. For instance, for both *root-canal refactoring* and *floss refactoring*, the number of worsened attributes represents a half of the number of improved attributes summed with the number of unaffected attributes. These results suggest that, regardless the refactoring tactic, refactoring operations tend to improve most metric of the attribute.

Table 5.6 presents the general results for the *At Least One Metric* approach. The first column lists each metric behavior. The second and third

Table 5.5: *Most Metrics* approach by refactoring tactic

Behavior	**Root-canal Refactoring**		**Floss Refactoring**	
	Related Attributes	**All Attributes**	**Related Attributes**	**All Attributes**
↑	8 (40%)	14 (25.45%)	5 (25%)	10 (18.18%)
↓	4 (20%)	11 (20.00%)	7 (35%)	20 (36.36%)
—	8 (40%)	30 (54.55%)	8 (40%)	25 (45.45%)

columns present, for root-canal refactoring, the number and percentage of internal attributes per behavior, in the case of related (second column) and all attributes (third column). The fourth and fifth columns present, for *floss refactoring*, the number, and percentage of internal attributes per behavior, in the case of related (fourth column) and all attributes (fifth column). Considering the *At Least One Metric* approach, the number of improved and unaffected attributes are both higher than the number of worsened attributes. This observation applies to both refactoring tactics. As expected, by using a less strict approach, these results evidence that refactoring operations have mostly at least a slightly positive impact on internal quality attributes.

Finding 4. *Root-canal refactoring tends to improve internal quality attributes when considering at least one metric per attribute. Moreover, floss refactoring tends to improve at least one metric per attribute, even if developers may not be explicitly concerned with this improvement.*

Table 5.6: *At Least One Metric* approach by refactoring tactic

Behavior	**Root-canal Refactoring**		**Floss Refactoring**	
	Related Attributes	**All Attributes**	**Related Attributes**	**All Attributes**
↑	13 (65%)	23 (41.82%)	11 (55%)	23 (41.82%)
↓	0 (0%)	5 (9.09%)	2 (10%)	11 (20%)
—	7 (35%)	27 (49.09%)	7 (35%)	21 (38.18%)

Comparing both approaches, we observed that the incidence of worsened internal attributes is higher in the *Most Metrics* approach than such incidence in the *At Least One Metric* approach. This observation applies to both refactoring tactics. This means that whenever refactoring operations worsen a internal attribute, those operations tend to considerably deteriorate the

PUC-Rio - Certificação Digital Nº 1522494/CA

attribute. In other words, negative refactoring operations more often decrease multiple (rather than a single) metric of each negatively affected attribute.

5.6 Example

To represent our finding in a real example, we will show a *Move Method* refactoring operation affecting the internal quality attributes. This refactoring type is often applied when a method is used more in another class than in its own class. To solve this issue, developers move the method for the class that most uses it. Listing 5.1 and 5.2 show the source code before and after applying the *Move Method* refactoring, respectively. This example is taken from Apache Ant[1] project from GitHub. The *SysProperties* class is a utility class for handling system properties and the *EnvironmentData* class is a wrapper for environment variables. Before applying the *Move Method* refactoring, the *size()* method is in the *SysProperties* class. This method returns the number of environment variables in the *environment* field. We can observe that the *EnvironmentData* class must provide this responsibility. Consequently, there is a coupling between the two classes.

PUC-Rio - Certificação Digital Nº 1522494/CA

Listing 5.1: Before applying Move Method

```
public class SysProperties
{
    private Properties m_system;
    private EnvironmentData environment;

    public int size()
    {
        return environment.m_variables.size();
    }

    //More methods and properties
}

public class EnvironmentData
{
    public ArrayList m_variables = new ArrayList();

    //More methods and properties
}
```

[1]Available at https://github.com/apache/ant

Listing 5.2: After applying Move Method

```
public class SysProperties
{
    private Properties m_system;

    //More methods and properties
}

public class EnvironmentData
{
    public ArrayList m_variables = new ArrayList();

    public int size()
    {
        return m_variables.size();
    }

    //More methods and properties
}
```

PUC-Rio - Certificação Digital Nº 1522494/CA

To understand the effects of *Move Method* refactoring, we quantify the internal quality attributes before and after the refactoring operation. Table 5.7 presents the effect in the internal quality attributes caused by *Move Method* in the example. Each column presents the effects of each internal quality attribute. We used the same notation of Table 5.1. This *Move Method* refactoring improves the *coupling* by eliminating the dependency between both classes. Additionally, it improves the *size* making the code more easy to maintain. The *cohesion*, *complexity*, and *inheritance* remained unaltered. This example illustrates our findings that suggest that for some refactoring types, such as *Move Method* refactoring, the internal quality attributes tend to improve or remain unaffected. However, there are several cases, including instances of the *Move Method* refactoring, where the effect is clearly negative: the refactoring operation, in addition to unalter some attributes, worsens others.

Table 5.7: Effects on internal attributes caused by *Move Method*

Cohesion	**Coupling**	**Complexity**	**Inheritance**	**Size**
_	↑	_	_	↑

Back to Finding 4, we observed that the internal quality attributes tend to improve or remain unaffected using the *At Least One Metric* approach, regardless the refactoring tactic. The aforementioned example illustrates a real scenario where there is an improvement in two internal attributes (*coupling* and *size*). In fact, our findings are significant since previous work reports that refactoring operations tend to worsen or do not affect the code quality. In contrast, as we illustrate with our example extracted from our empirical study conducted with real software projects, certain refactoring types have a potential to improve a single or multiple internal quality attributes together.

5.7 Final Remarks

In this chapter, we analyze the effects of refactoring operations on internal quality attributes. We divide the analysis into two approaches, related to how to classify the behavior of internal quality attributes: (i) *At Least One Metric*, and (ii) *Most Metrics*. In (i) *At Least One Metric* approach, we assume that the internal quality attribute improves when any of the metrics that measures such attribute improve. In *Most Metrics* approach, we assume that an internal quality attribute improves when most of the metrics that capture the attribute also improve. We also analyze the impact of *root-canal refactoring* and *floss refactoring* using both approaches.

As a result, we obtain interesting observations. First, in 65% of the cases, the related internal quality attributes are improved, and the remaining 35% operations keep the quality attributes unaffected. Second, whenever *root-canal refactoring* operations are applied, we confirm that internal quality attributes are either frequently improved or at least not worsened. Finally, while refactoring operations often reach other specific aims, called *floss refactoring*, 55% of these operations surprisingly improve internal quality attributes, with only 10% of the quality decline.

We provide recommendations for applying each refactoring type to aimed at improving the code structural quality. These recommendations could support the proposal of refactoring recommendation tools based, on the internal quality attributes which mostly concern developers and organizations. The next chapter presents and discusses the results of RQ3 (*What is the impact of re-refactoring on internal quality attributes?*).

PUC-Rio - Certificação Digital Nº 1522494/CA

6
Re-Refactoring Impact on Internal Attributes

In this chapter, we assess the impact of re-refactoring operations on internal quality attributes. We aim to understand if developers improve at least one internal quality attribute after refactoring code elements which was refactored in the past. On the other hand, we are interested in analyzing if developers affect most of the internal attributes when applying a re-refactoring operation. Thus, we answer RQ3 (*What is the impact of re-refactoring operations on internal quality attributes?*) as follows.

6.1
Answering RQ3 with Two Approaches

Similarly to Chapter 5, we divide the analysis into two approaches: *Most Metrics* and *At Least One Metric* approaches. Section 6.2 and Section 6.3 describe each approach, respectively. For the data analysis in this section, we use the same notation of Table 5.1. That is, we use an arrow up (↑) to express an improvement on the attribute, an arrow down (↓) to express negative effects on attributes and (_) to expose unaltered attributes.

PUC-Rio - Certificação Digital Nº 1522494/CA

6.2
The Most Metrics Approach

Table 6.1 presents the impact of re-refactoring operations per type using the *Most Metrics* approach. The first column lists the 11 refactoring types under analysis. The second column provides the total number of re-refactoring operations per each type. The remaining columns concern the five internal quality attributes investigated in this study. For each column, the table presents the predominant behavior for the respective refactoring type. For this purpose, we used the notation described in Table 5.1. These columns also provide the percentage of refactoring operations categorized in the predominant behavior for the respective internal quality attribute. Highlighted cells in the table indicate that a given internal quality attribute (i.e., a column of the table) is expected to improve after applying a given refactoring type (i.e., a line of the table). The gray highlight on each cell was assigned by the mapping of re-refactoring to internal quality attributes described in Table 3.2.

These gray cells represent the cases of internal quality attributes expected to be improved by the corresponding refactoring types.

Table 6.1: Re-refactoring impact using the *Most Metrics* approach

Refactoring Type	**Total**	**Cohesion**	**Coupling**	**Complexity**	**Inheritance**	**Size**
Extract Superclass	114	↑ 54.39%	— 61.40%	— 57.89%	— 50%	↑ 71.93%
Extract Interface	42	— 85.71%	— 76.19%	— 100%	↑ 57.14%	— 54.76%
Move Field	3,813	↑ 65.7%	— 59.74%	— 90.24%	— 91.37%	— 56.86%
Push down Field	35	↑ 40%	↑ 42.86%	— 85.71%	— 97.14%	↓ 54.29%
Inline Method	1,521	↑ 58.32%	↓ 39.91%	— 48.98%	— 92.9%	↑ 56.54%
Extract Method	7,478	↓ 59.11%	↓ 45.77%	↑ 44.93%	— 93.82%	↓ 58.53%
Move Method	888	↓ 47.52%	↓ 38.29%	— 71.17%	— 84.23%	↑ 44.26%
Push down Method	70	↓ 48.57%	↑ 42.86%	— 51.43%	— 94.29%	↓ 57.14%
Pull up Method	301	↓ 39.2%	↓ 72.43%	— 67.77%	— 91.36%	↓ 59.47%
Pull up Field	259	↓ 47.88%	↓ 59.07%	— 61%	— 77.22%	↓ 69.5%

From data in Table 6.1, we observed several interesting results related to the impact of re-refactoring operations on internal quality attributes. Our results show that for the *Extract Superclass*, *Extract Interface*, *Move Field*, *Push down field*, and *Inline method* one or more internal quality attributes tend to improve (it improves rather than worsens), while the others remained unaffected. Furthermore, one internal attribute related to a given refactoring type has improved. Overall, five out of 10 (50%) of the refactoring types have improved for one or more internal quality attributes. On the other hand, we observed that for the *Extract Method*, *Move Method*, *Push down Method*, *Pull up Method*, and *Pull up Field* the related internal attributes (gray cells) always worsen or remain unaffected. Overall, five out of 10 (50%) of the refactoring types have worsened for one or more related internal quality attributes (it worsens rather than improves). In addition, we observed that *Pull up Method* and *Pull up Field* never improve any of the internal quality attributes when developers re-refactor the affected code elements.

Based in Table 6.1, we compute the tally of internal quality attributes that improve, worsen and remain unaffected for each approach. Concerning the *Most Metrics* approach, the amount of related internal quality attributes (gray cells) is equal to four (21.05%); thus, we observed some positive impact of refactoring operations on certain internal quality attributes. However, the amount of worsened internal attributes is equal to seven (36.84%), while eight

PUC-Rio - Certificação Digital Nº 1522494/CA

Table 6.2: Re-refactoring impact with *At Least One Metric* approach

Refactoring Type	**Total**	**Cohesion**	**Coupling**	**Complexity**	**Inheritance**	**Size**
Extract superclass	114	↑ 54.39%	↑ 55.26%	— 57.89%	↑ 95.61%	↑ 86.84%
Push down field	35	↑ 40%	↑ 54.29%	— 85.71%	— 97.14%	↑ 82.86%
Inline method	1,521	↑ 58.32%	↑ 77.65%	— 48.13%	— 91.45%	↑ 89.02%
Move field	3,813	↑ 65.7%	— 54.34%	— 90.24%	— 89.46%	— 54.73%
Extract interface	42	— 85.71%	— 69.05%	— 100%	↑ 59.52%	— 50%
Extract method	7,478	↓ 59.11%	↑ 71.4%	↑ 46.99%	— 93.1%	↑ 85.85%
Move method	888	↓ 47.52%	↑ 51.01%	— 71.17%	— 77.59%	↑ 85.25%
Push down method	70	↓ 49.02%	↑ 80%	— 51.43%	— 94.29%	↑ 84.29%
Pull up method	301	↓ 40.32%	↓ 72.99%	— 67.77%	— 91.03%	↑ 83.72%
Pull up field	259	↓ 46.98%	↓ 59.07%	— 61%	— 72.2%	↑ 85.71%

(42.11%) remain unaffected. In fact, the number of worsened internal quality attributes surpasses the number of improved attributes. However, it represents only a half of the total number of improved attributes summed with unaffected ones. These results suggest we should study if less strict approaches (e.g., the *At Least One Metric* approach) may better capture the positive impact of re-refactoring operations.

6.3 The At Least One Metric Approach

Using the *At Least One Metric* approach, when any of the metrics used to quantify the internal quality attribute improve, it means that the internal quality attribute also improves. Table 6.2 presents the impact of re-refactoring operations per refactoring type using the *At Least One Metric* approach. The structure of this table is similar to Table 6.1. Table 6.2 show that for 9 out of 10 refactoring types, representing a 90%, the internal quality attributes improved and remained unaffected. Only in the case of the *Move Method* refactoring type, which worsened the cohesion attribute.

Based on the data of Table 6.2, we compute the tally of internal quality attributes that improve, worsen and remain unaffected. For *At Least One Metric* approach, the amount of related internal quality attributes (grey cells) is equal to 11 (57.89%). The amount of worsened internal attributes is equal to one (5.26%), while seven (36.84%) remain unaffected. As a result, we observed that using *At Least One Metric* approach, most re-refactoring operations

PUC-Rio - Certificação Digital Nº 1522494/CA

improve the related internal attributes, representing a 57.89%.

Finding 5. *Similarly to the analysis of individual refactorings, for some refactoring types, the internal quality attributes tend to improve or remain unaffected when developers refactor code that was also refactored in the past (re-refactoring). Additionally, the re-refactoring operations tend to improve or unalter the internal quality attributes when considering at least one metric per internal quality attribute.*

6.4 Root-canal versus Floss Re-Refactoring

This section discusses the impact of *root-canal re-refactoring* and *floss re-refactoring* on internal quality attributes using both approaches. The set of *root-canal re-refactoring* is composed by the *root-canal refactoring* affecting code elements that were refactored in the past. This means that the developers are applying a re-refactoring with the aim of exclusively improving the code structural quality. On the other hand, the set of *floss re-refactoring* is composed by the *floss refactoring* affecting code elements that were refactored in the past. This indicates that the developers are applying a re-refactoring with more specific goals, such as adding features or fixing bugs.

Table 6.3 presents the general results for the *Most Metrics* approach. The first column lists each metric behavior. The second and third columns present data regarding *root-canal re-refactoring*. Each cell presents the number and percentage of internal quality attributes in the case of related attributes and all attributes, respectively. The fourth and fifth columns present data regarding *floss re-refactoring*. Similarly to the *root-canal re-refactoring*, each cell presents the number and percentage of internal quality attributes in the case of the related internal quality attributes and all attributes, respectively.

Table 6.3: Refactoring tactics and *Most Metrics* approach

Behavior	**Root-canal Refactoring**		**Floss Refactoring**	
	Related Attributes	**All Attributes**	**Related Attributes**	**All Attributes**
↑	9 (47.37%)	16 (32%)	5 (26.32%)	9 (18%)
↓	4 (21.05%)	11 (22%)	7 (36.84%)	19 (38%)
—	6 (31,58%)	23 (46%)	7 (36.84%)	22 (44%)

Data in Table 6.3 show results similar those observed in the analysis of refactorings operations for the *Most Metrics* approach. For instance, for both

PUC-Rio - Certificação Digital Nº 1522494/CA

root-canal and *floss refactoring*, the number of worsened attributes represents a half of the number of other (improved and unaffected) attributes. These results suggest that regardless the refactoring tactic, re-refactoring operations tend to improve most metrics of the attribute.

Table 6.4 presents the general results for the *At Least One Metric* approach. The first column lists each metric behavior. The second and third columns present, for *root-canal re-refactoring*, the number and percentage of internal quality attributes per behavior, in the case of related (second column) and all attributes (third column). The fourth and fifth columns present, for *floss re-refactoring*, the number, and percentage of internal quality attributes per behavior, in the case of related (fourth column) and all attributes (fifth column).

Table 6.4: Refactoring tactics and *At Least One Metric* approach

Behavior	**Root-canal Refactoring**		**Floss Refactoring**	
	Related Attributes	**All Attributes**	**Related Attributes**	**All Attributes**
↑	13 (68.42%)	26 (52%)	12 (63.16%)	22 (44%)
↓	0 (0%)	5 (10%)	1 (5.26%)	11 (22%)
—	6 (31.58%)	19 (38%)	6 (31.58%)	17 (34%)

Considering the *At Least One Metric* approach, the number of improved and unaffected attributes are both higher than the number of worsened attributes. This observation applies to both refactoring tactics. As expected by using a less strict approach, these results evidence that similar to refactoring operations, the re-refactoring operations have mostly at least a slightly positive impact on internal quality attributes. On the other hand, it is interesting to note that using the At least One Metrics approach, the internal quality attributes that worsen are less than 22%. This observation is valid for both refactoring tactics. In the specific case of root-canal re-refactoring, we find that the percentages of internal quality attributes that worsen are less than 10%.

Finding 6. *There is no difference between the results of refactorings and re-refactorings operations regarding the effect on the internal quality attributes.*

PUC-Rio - Certificação Digital Nº 1522494/CA

6.5
Final Remarks

In this chapter, we analyze the effects of re-refactoring operations on internal quality attributes. We divide the analysis into two approaches, related to how to classify the behavior of internal quality attributes: (i) *At Least One Metric*, and (ii) *Most Metrics*. In (i) *At Least One Metric* approach, we assume that the internal quality attribute improves when any of the metrics that measures such attribute improve. In *Most Metrics* approach, we assume that an internal quality attribute improves when most of the metrics that capture the attribute also improve. We also analyze the impact of *root-canal re-refactoring* and *floss re-refactoring* using both approaches.

Similarly to the analysis of refactoring operations, we observe that re-refactoring operations tend to improve or remain unaffected the internal quality attributes for some refactoring types. That is, the internal quality attributes tend to improve or remain unaffected when developers refactor code that was refactored in the past (re-refactoring), for some refactoring types. However, for *Extract Method*, *Move Method*, *Pull up Method*, and *Pull up Field* refactoring types, the related internal quality attributes tend to worsen or unalter after re-refactoring. Additionally, the re-refactoring operations tend to improve or unalter the internal quality attributes when considering at least one metric per internal quality attribute.

PUC-Rio - Certificação Digital Nº 1522494/CA

7 Conclusion and Future Work

PUC-Rio - Certificação Digital Nº 1522494/CA

This dissertation investigates the impact of refactoring and re-refactoring operations on internal quality attributes. For this purpose, we analyze 23 Java open source projects with 29,303 refactoring operations, which includes 49.55% of re-refactoring operations. Our study focuses on 11 commonly studied refactoring types and five internal quality attributes. We split our study into three parts. First, we investigate whether refactoring operations are often applied to code elements with critical internal quality attributes. Second, we assess the effects of refactoring operations on internal quality attributes using two complementary approaches: *Most Metrics*, when most of the metrics that capture an internal quality attribute improve, and *At Least One Metric*, when at least one of the metrics improves. Third, we assess the effects of re-refactoring operations on internal quality attributes using two complementary approaches: *Most Metrics* and *At Least One Metric*. We define re-refactoring as the action of applying another refactoring in a code element that was refactored before. We also analyze two different refactoring tactics: *root-canal refactoring*, when developers are explicitly concerned on improving the code structure quality, and *floss refactoring*, when quality improvement is a means to reach other goals.

As a result, we have noticed that developers apply more than 93.45% of the refactoring operations to code elements with at least one critical internal quality attribute. Furthermore, in 65% of the cases, the related internal quality attributes are improved, and the remaining 35% operations keep the quality attributes unaffected. Also, whenever pure refactoring operations are applied (*root-canal refactoring*), we confirm that internal quality attributes are either frequently improved or at least not worsened. Finally, while refactoring operations often reach other specific aims, such as adding a new feature or fixing a bug, 55% of these operations improve internal quality attributes, with only 10% of the quality decline. We found similar results for re-refactoring operations regarding the effect on the internal quality attributes. This means that there is no difference between the results of refactorings and re-refactorings operations regarding the effect on the internal quality attributes.

This dissertation provides several contributions. First, our results suggest that developers should consider applying refactoring and re-refactoring to improve the code structural quality. However, they should be careful when applying certain refactoring types, which could worsen the code structural quality. Second, the paper compares our empirical study with previous studies (17, 18). Surprisingly, our study findings partially contradict the findings of these studies. Our findings suggest that most refactoring operations are applied on code elements with critical internal quality attributes, by positively affecting it and improving the code structural quality. Third, the paper investigates the effects of re-refactoring operations on internal quality attributes, which is barely unexplored by previous work. Fourth, we provide recommendations for applying refactoring operations when developers attempt to improve the structural quality by improving internal attributes. These recommendations can be used by tools to suggest or alert in the application of refactoring operations to improve the code structural quality by improving internal quality attributes. Finally, the paper provides a mapping of several refactoring types from the literature (1) with five well-known internal quality attributes. Additionally, the paper provides a mapping of these five internal quality attributes with software metrics which are commonly used to quantify them (30, 31). Although previous work (32) provide similar mappings, we cover a larger set of refactoring operations, internal quality attributes, and software metrics. Both mappings may support further investigations on the code structural quality.

As future work, we intend to reflect upon our findings in order to improve refactoring tools. For instance, one could think of a recommender system that ranks refactoring opportunities in terms of the structural criticality of the code elements in a program. Additionally, incorporate the recommendations derived from our findings to propose suggestions and opportunities for code quality improvements concerned with internal quality attributes. Another direction of future work consists of conducting other studies using software systems written in different programming languages, such as C++ or C#. Also it is interesting to understand the effect of refactoring operations in proprietary software systems. Our present study focused only on the analysis of popular open source projects, which may have different structural degradation behaviors as compared to proprietary software systems. Finally, as future work, we propose to study the relationship between refactoring and re-refactoring operation with the addition of bugs and bugs fixing, as well as other design problems.

PUC-Rio - Certificação Digital Nº 1522494/CA

Bibliography

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

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

[3] MENS, T.; TOURWÉ, T.. **A survey of software refactoring**. IEEE Trans. Softw. Eng., 30(2):126–139, 2004.

[4] HEITLAGER, I.; KUIPERS, T. ; VISSER, J.. **A practical model for measuring maintainability**. In: QUALITY OF INFORMATION AND COMMUNICATIONS TECHNOLOGY, 2007. QUATIC 2007. 6TH INTERNATIONAL CONFERENCE ON THE, p. 30–39. IEEE, 2007.

[5] KITCHENHAM, B.; PFLEEGER, S. L.. **Software quality: the elusive target [special issues section]**. IEEE software, 13(1):12–21, 1996.

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

[7] AOKI, A.; HAYASHI, K.; KISHIDA, K.; NAKAKOJI, K.; NISHINAKA, Y.; REEVES, B.; TAKASHIMA, A. ; YAMAMOTO, Y.. **A case study of the evolution of jun: an object-oriented open-source 3d multimedia library**. In: PROCEEDINGS OF THE 23RD INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, p. 524–533. IEEE Computer Society, 2001.

[8] AL DALLAL, J.. **The impact of inheritance on the internal quality attributes of java classes**. Kuwait Journal of Science and Engineering, 39(2A):131–154, 2012.

[9] DESTEFANIS, G.; COUNSELL, S.; CONCAS, G. ; TONELLI, R.. **Software metrics in agile software: An empirical study**. In: INTERNATIONAL CONFERENCE ON AGILE SOFTWARE DEVELOPMENT, p. 157–170. Springer, 2014.

PUC-Rio - Certificação Digital Nº 1522494/CA

[10] SIMON, F.; STEINBRUCKNER, F. ; LEWERENTZ, C.. **Metrics based refactoring**. In: SOFTWARE MAINTENANCE AND REENGINEERING, 2001. FIFTH EUROPEAN CONFERENCE ON, p. 30–38. IEEE, 2001.

[11] MURPHY-HILL, E.; PARNIN, C. ; BLACK, A. P.. **How we refactor, and how we know it**. In: PROCEEDINGS OF THE 31ST INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, ICSE '09, p. 287–297, Washington, DC, USA, 2009. IEEE Computer Society.

[12] MURPHY-HILL, E.; PARNIN, C. ; BLACK, A.. **How we refactor, and how we know it**. IEEE Trans. Softw. Eng., 38(1):5–18, 2012.

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

[14] ALSHAYEB, M.. **Empirical investigation of refactoring effect on software quality**. Information and software technology, 51(9):1319–1326, 2009.

[15] GEPPERT, B.; MOCKUS, A. ; ROBLER, F.. **Refactoring for changeability**. In: PROCEEDINGS OF THE 11TH INTERNATIONAL SYMPOSIUM ON SOFTWARE METRICS (METRICS), p. 10–pp, 2005.

[16] MOSER, R.; SILLITTI, A.; ABRAHAMSSON, P. ; SUCCI, G.. **Does refactoring improve reusability?** In: INTERNATIONAL CONFERENCE ON SOFTWARE REUSE, p. 287–297. Springer, 2006.

[17] 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., 107:1–14, 2015.

[18] CEDRIM, D.; SOUSA, L.; GARCIA, A. ; GHEYI, R.. **Does refactoring improve software structural quality?** In: PROCEEDINGS OF THE 30TH BRAZILIAN SYMPOSIUM ON SOFTWARE ENGINEERING (SBES), p. 73–82, 2016.

[19] DU BOIS, B.; DEMEYER, S. ; VERELST, J.. **Refactoring-improving coupling and cohesion of existing code**. In: PROCEEDINGS OF THE 11TH WORKING CONFERENCE ON REVERSE ENGINEERING (WCRE), p. 144–151, 2004.

PUC-Rio - Certificação Digital Nº 1522494/CA

[20] JIAU, H. C.; MAR, L. W. ; CHEN, J. C.. **Obey: Optimal batched refactoring plan execution for class responsibility redistribution**. IEEE Transactions on Software Engineering, 39(9):1245–1263, 2013.

[21] ARCELLI, D.; CORTELLESSA, V. ; DI POMPEO, D.. **Towards a unifying approach for performance-driven software model refactoring**. In: GEMOC+ MPM@ MODELS, p. 42–51, 2015.

[22] ARCELLI, D.; CORTELLESSA, V. ; DI POMPEO, D.. **Automated translation among epsilon languages for performance-driven uml software model refactoring**. In: PROCEEDINGS OF THE 1ST INTERNATIONAL WORKSHOP ON SOFTWARE REFACTORING, p. 25–32. ACM, 2016.

[23] CHÁVEZ, A.; FERREIRA, I.; FERNANDES, E.; CEDRIM, D. ; GARCIA, A.. **How does refactoring affect internal quality aributes? a multi-project study**. In: PROCEEDINGS OF THE 31ST BRAZILIAN SYMPOSIUM ON SOFTWARE ENGINEERING (SBES), p. 1–10, 2017.

[24] FERREIRA, K. A.; BIGONHA, M. A.; BIGONHA, R. S.; MENDES, L. F. ; ALMEIDA, H. C.. **Identifying thresholds for object-oriented software metrics**. Journal of Systems and Software, 85(2):244–257, 2012.

[25] NORMAN, F. E.. **Software metrics–a rigorous approach**, 1991.

[26] FENTON, N.; PFLEEGER, S.. **Software metrics: a rigorous and practical approach pws publishing co**. Boston, MA, USA, 1997.

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

[28] KERIEVSKY, J.. **Refactoring to patterns**. Pearson Deutschland GmbH, 2005.

[29] BAKOTA, T.; HEGEDUS, P.; LADÁNYI, G.; KORTVELYESI, P.; FERENC, R. ; GYIMÓTHY, T.. **A cost model based on software maintainability**. In: SOFTWARE MAINTENANCE (ICSM), 2012 28TH IEEE INTERNATIONAL CONFERENCE ON, p. 316–325. IEEE, 2012.

[30] CHIDAMBER, S.; KEMERER, C.. **A metrics suite for object oriented design**. IEEE Trans. Softw. Eng., 20(6):476–493, 1994.

PUC-Rio - Certificação Digital Nº 1522494/CA

[31] LORENZ, M.; KIDD, J.. **Object-oriented software metrics**. Prentice Hall, 1994.

[32] BANSIYA, J.; DAVIS, C. G.. **A hierarchical model for object-oriented design quality assessment**. IEEE Transactions on software engineering, 28(1):4–17, 2002.

[33] FOWLER, M.. **Catalog of refactorings**, 2013.

[34] YAMASHITA, A.; MOONEN, L.. **Do code smells reflect important maintainability aspects?** In: SOFTWARE MAINTENANCE (ICSM), 2012 28TH IEEE INTERNATIONAL CONFERENCE ON, p. 306–315. IEEE, 2012.

[35] YAMASHITA, A.; COUNSELL, S.. **Code smells as system-level indicators of maintainability: An empirical study**. Journal of Systems and Software, 86(10):2639–2653, 2013.

[36] BROWN, W.; MALVEAU, R.; MCCORMICK, H. ; MOWBRAY, T.. **AntiPatterns**. John Wiley & Sons, Inc., 1998.

[37] MALHOTRA, R.. **Empirical research in software engineering**. CRC Press, 2016.

[38] YOURDON, E.; CONSTANTINE, L. L.. **Structured design: Fundamentals of a discipline of computer program and systems design**. Prentice-Hall, Inc., 1979.

[39] BIEMAN, J.; KANG, B.-K.. **Cohesion and reuse in an object-oriented system**. In: ACM SIGSOFT SOFTWARE ENGINEERING NOTES, volumen 20, p. 259–262, 1995.

[40] PRESSMAN, R. S.. **Software engineering: a practitioner's approach**. New York, McGraw Hill, 1987.

[41] KATAOKA, Y.; IMAI, T.; ANDOU, H. ; FUKAYA, T.. **A quantitative evaluation of maintainability enhancement by refactoring**. In: PROCEEDINGS OF THE 18TH INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE (ICSM), p. 576–585, 2002.

[42] MCCABE, T.. **A complexity measure**. IEEE Trans. Softw. Eng., (4):308–320, 1976.

PUC-Rio - Certificação Digital Nº 1522494/CA

[43] BIEMAN, J. M.; ZHAO, J. X.. **Reuse through inheritance: A quantitative study of c++ software**. ACM SIGSOFT Software Engineering Notes, 20(SI):47–52, 1995.

[44] BOOCH, G.. **Object-oriented analysis and design with applications, 1994**. Redwood City, CA.

[45] LI, W.; HENRY, S.. **Object-oriented metrics that predict maintainability**. J. Syst. Softw., 23(2):111–122, 1993.

[46] LIU, Y.; POSHYVANYK, D.; FERENC, R.; GYIMÓTHY, T. ; CHRISOCHOIDES, N.. **Modeling class cohesion as mixtures of latent topics**. In: PROCEEDINGS OF THE 25TH INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE (ICSM), p. 233–242, 2009.

[47] BIEMAN, J.; OTT, L.. **Measuring functional cohesion**. IEEE Trans. Softw. Eng., 20(8):644–657, 1994.

[48] SOKOL, F. Z.; ANICHE, M. F. ; GEROSA, M. A.. **Does the act of refactoring really make code simpler? a preliminary study**. In: 4TH BRAZILIAN WORKSHOP ON AGILE METHODS, 2013.

[49] WILKING, D.; KAHN, U. ; KOWALEWSKI, S.. **An empirical evaluation of refactoring**. e-Informatica, 1(1):27–42, 2007.

[50] DU BOIS, B.; MENS, T.. **Describing the impact of refactoring on internal program quality**. In: PROCEEDINGS OF THE INTERNATIONAL WORKSHOP ON EVOLUTION OF LARGE-SCALE INDUSTRIAL SOFTWARE APPLICATIONS (ELISA), CO-LOCATED WITH 19TH ICSM, p. 37–48, 2003.

[51] STROGGYLOS, K.; SPINELLIS, D.. **Refactoring–does it improve software quality?** In: PROCEEDINGS OF THE 5TH INTERNATIONAL WORKSHOP ON SOFTWARE QUALITY, p. 10. IEEE Computer Society, 2007.

[52] MURPHY-HILL, E.; BLACK, A. P.; DIG, D. ; PARNIN, C.. **Gathering refactoring data: a comparison of four methods**. In: PROCEEDINGS OF THE 2ND WORKSHOP ON REFACTORING TOOLS, p. 7. ACM, 2008.

[53] DIAS, M.; BACCHELLI, A.; GOUSIOS, G.; CASSOU, D. ; DUCASSE, S.. **Untangling fine-grained code changes**. In: SOFTWARE ANALYSIS, EVOLUTION AND REENGINEERING (SANER), 2015 IEEE 22ND INTERNATIONAL CONFERENCE ON, p. 341–350. IEEE, 2015.

PUC-Rio - Certificação Digital Nº 1522494/CA

[54] TSANTALIS, N.; CHATZIGEORGIOU, A.. **Identification of extract method refactoring opportunities for the decomposition of methods**. Journal of Systems and Software, 84(10):1757–1782, 2011.

[55] TAIRAS, R.; GRAY, J.. **Increasing clone maintenance support by unifying clone detection and refactoring activities**. Information and Software Technology, 54(12):1297–1307, 2012.

[56] HUA, L.; KIM, M. ; MCKINLEY, K. S.. **Does automated refactoring obviate systematic editing?** In: SOFTWARE ENGINEERING (ICSE), 2015 IEEE/ACM 37TH IEEE INTERNATIONAL CONFERENCE ON, volumen 1, p. 392–402. IEEE, 2015.

[57] HOTTA, K.; HIGO, Y. ; KUSUMOTO, S.. **Identifying, tailoring, and suggesting form template method refactoring opportunities with program dependence graph**. In: SOFTWARE MAINTENANCE AND REENGINEERING (CSMR), 2012 16TH EUROPEAN CONFERENCE ON, p. 53–62. IEEE, 2012.

[58] BOSHNAKOSKA, D.; MIŠEV, A.. **Correlation between object-oriented metrics and refactoring**. In: INTERNATIONAL CONFERENCE ON ICT INNOVATIONS, p. 226–235. Springer, 2010.

[59] MOORE, D.; NOTZ, W. ; FLIGNER, M.. **The basic practice of statistics**. W. H. Freeman, 2015.

[60] RODRIGUEZ, D.; HARRISON, R.. **An overview of object-oriented design metrics**. 2001.

[61] TSANTALIS, N.; GUANA, V.; STROULIA, E. ; HINDLE, A.. **A multidimensional empirical study on refactoring activity**. In: PROCEEDINGS OF THE 23RD CONFERENCE OF THE CENTER FOR ADVANCED STUDIES ON COLLABORATIVE RESEARCH (CASCON), p. 132–146, 2013.

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

[63] JOSHUA, K.. **Refactoring to patterns**, 2005.

[64] HENDERSON-SELLERS, B.. **Object-oriented metrics**. Prentice-Hall, Inc., 1995.

PUC-Rio - Certificação Digital Nº 1522494/CA

[65] HITZ, M.; MONTAZERI, B.. **Chidamber and kemerer's metrics suite: a measurement theory perspective**. IEEE Transactions on software Engineering, 22(4):267–271, 1996.

[66] HENRY, S.; KAFURA, D.. **Software structure metrics based on information flow**. IEEE transactions on Software Engineering, (5):510–518, 1981.

[67] NEJMEH, B. A.. **Npath: a measure of execution path complexity and its applications**. Communications of the ACM, 31(2):188–200, 1988.

[68] MARCUS, A.; POSHYVANYK, D. ; FERENC, R.. **Using the conceptual cohesion of classes for fault prediction in object-oriented systems**. IEEE Transactions on Software Engineering, 34(2):287–300, 2008.

[69] POSHYVANYK, D.; MARCUS, A.; FERENC, R. ; GYIMÓTHY, T.. **Using information retrieval based coupling measures for impact analysis**. Empirical software engineering, 14(1):5–32, 2009.

[70] NAGAPPAN, M.; ZIMMERMANN, T. ; BIRD, C.. **Diversity in software engineering research**. In: PROCEEDINGS OF THE 2013 9TH JOINT MEETING ON FOUNDATIONS OF SOFTWARE ENGINEERING, p. 466–476. ACM, 2013.

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

PUC-Rio - Certificação Digital Nº 1522494/CA

A
Published paper

Chávez, A., Ferreira, I., Fernandes, E., Cedrim, D., and Garcia, A. (2017). How does refactoring affect internal quality attributes? a multi-project study. InProceedings of the 31st Brazilian Symposium on Software Engineering (SBES)

Cedrim, D., Garcia, A., Mongiovi, M., Gheyi, R., Sousa, L., de Mello, R., ... Chávez, A. (2017, August). Understanding the impact of refactoring on smells: a longitudinal study of 23 software projects. In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (FSE) (pp. 465-475). ACM.

PUC-Rio - Certificação Digital Nº 1522494/CA