Pontifícia Universidade Católica
do Rio de Janeiro

Vinícius Passos de Oliveira Soares

Aligning developer quality concerns, refactoring applications, and their effects

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 2021

Pontifícia Universidade Católica
do Rio de Janeiro

Vinícius Passos de Oliveira Soares

Aligning developer quality concerns, refactoring applications, and their effects

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 Examination Committee.

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

Prof. Alberto Barbosa Raposo
Departamento de Informática – PUC-Rio

Prof. Marcos Kalinowski
Departamento de Informática – PUC-Rio

Rio de Janeiro, September 21st, 2021

Vinícius Passos de Oliveira Soares
Sou estudante de Mestrado em Ciência da Computação pela Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio). Tenho bacharelado em Ciência da Computação pela Universidade do Estado do Rio de Janeiro (UERJ). Minha pesquisa atual tem como foco determinar a correlação entre preocupações de desenvolvedores com relação à qualidade do código sendo modificado, e tanto quais quanto como refatorações são aplicadas ao código nestas situações. Assim, pretende-se permitir que desenvolvedores tenham uma maneira mais acurada de definir quais refatorações devem ser aplicadas, e como devem ser aplicadas, dependendo da situação atual, e das preocupações do desenvolvedor no momento da mudança. Como resultado de minha dedicação à pesquisa, enviei trabalhos aceitos para veículos relevantes, como o Simpósio Brasileiro de Engenharia de Software (SBES).

Bibliographic data

Passos de Oliveira Soares, Vinícius

Aligning developer quality concerns, refactoring applications, and their effects / Vinícius Passos de Oliveira Soares; advisor: Alessandro Fabricio Garcia. – 2021.

96 f: il. color. ; 30 cm

Dissertação (mestrado) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática, 2021.

Inclui bibliografia

1. Refactoring – Teses. 2. Refactoring Complexity – Teses. 3. Self-Affirmed Refactorings – Teses. 4. Non-Functional Concerns – Teses. 5. Refatoração. 6. Atributos de Qualidade Interna. 7. Refatorações Auto-Afirmadas. 8. Requisitos Não-Funcionais. 9. Preocupações de Desenvolvedores. I. Fabricio Garcia, Alessandro. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

Acknowledgments

I would like to first thank my advisor, Prof. Alessandro Fabricio Garcia, for the guidance during this process, and the OPUS group, for all the help given through these two years. I also thank my graduation advisor, Prof. Marcelo Schots de Oliveira, for paving my path to the world of research. Special thanks to my parents, for being at my side through all that has happened, and to Daniel Coutinho, for all the help since my undergraduate years.

To CAPES and PUC-Rio, for the aids granted, without which this work does not could have been accomplished. To FAPERJ, for the aids granted for the projects I have participated.

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001.

Abstract

Passos de Oliveira Soares, Vinícius; Fabricio Garcia, Alessandro (Advisor). **Aligning developer quality concerns, refactoring applications, and their effects**. Rio de Janeiro, 2021. 96p. Dissertação de Mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Even though the refactoring process has been increasingly investigated in the last years, many of its characteristics remain poorly understood. Software refactoring is the process of improving the maintainability of a system through structural changes that do not alter its behaviour. Recent studies revealed that software projects frequently have to undergo composite refactorings. In such refactorings, developers perform a series of single transformations in conjunction and in a single commit, which are expected to have a larger and more positive impact than single refactorings. However, refactorings frequently cause changes that either keep the software quality the same, or cause it to worsen, which lead recent works to look for potential causes of this behavior. However, the complexity of these composite changes often affecting their outcomes in some positive or (unexpectedly) negative way remains not investigated, much like the developers' concerns while performing refactoring. For the latter, some previous work was performed around characterizing and detecting refactoring-related developer discussions. However, it is unknown whether and how developers' refactoring concerns made explicit in such discussions can influence the refactorings' effects on a system. Thus, this work reports two studies aimed at bridging some of those gaps in knowledge in which causes lead to the non-positive effects frequently found in refactoring, by understanding: (i) if more complex refactorings are indeed more effective than simple refactorings, as one would expect; (ii) in which situations developers tend to have explicit concerns while refactoring the code; and (iii) what is the impact of such concerns on the effectiveness of a refactoring to improve structural quality. We analyze these characteristics and reach the following results: First, as refactoring complexity increases, the effectiveness of such refactorings increases as well. Second, there is a relationship between refactoring effectiveness and explicit refactoring concerns, in which the possibility of negative effects is lower when developers are explicitly concerned about refactoring. Finally, developers tend to be more explicit about their concerns on the refactoring process when they are faced with more complex refactoring tasks.

Keywords

Refactoring; Internal Quality Attributes; Self-Affirmed Refactorings; Non-Functional Requirements; Developer Concern.

Resumo

Passos de Oliveira Soares, Vinícius; Fabricio Garcia, Alessandro. **Alinhando preocupações de qualidade de desenvolvedores a aplicações de refatorações e seus efeitos**. Rio de Janeiro, 2021. 96p. Dissertação de Mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Mesmo com o processo de refatoração sendo investigado cada vez mais nos últimos anos, muitas de suas características se mantém pouco compreendidas. Refatoração de software é o processo de melhorar a manutenibilidade de um sistema por meio de mudanças estruturais que não alteram seu comportamento. Estudos recentes revelaram que projetos de software frequentemente recebem refatorações compostas. Em tais refatorações, desenvolvedores aplicam uma série de transformações únicas em conjunção e em um único commit, e se espera que estas refatorações tenham um efeito maior e mais positivo do que refatorações singulares. Porém, refatorações frequentemente causam mudanças que ou mantém a qualidade do software da mesma forma, ou causam a piora do mesmo, levando trabalhos recentes a procurar causas em potencial para este comportamento. Porém, o porquê da complexidade destas mudanças compostas frequentemente afetarem seus resultados de alguma forma positiva ou (inesperadamente) negativa continua não investigado. O mesmo ocorre com o potencial efeito das preocupações dos desenvolvedores durante a aplicação de refatorações. Sobre estas preocupações, alguns trabalhos anteriores foram desenvolvidos em torno da caracterização e detecção de discussões de desenvolvedores relacionadas a refatorações. Porém, não se sabe se e como estas preocupações de desenvolvedores com refatorações, tornando-se explícitas em tais discussões, podem influenciar os efeitos de refatorações em um sistema. Portanto, este trabalho apresenta dois estudos com o objetivo de preencher a lacuna no conhecimento de que causas levam aos efeitos não-positivos frequentemente encontrados em refatorações, procurando entender: (i) se refatorações mais complexas realmente são mais efetivas do que refatorações simples, como esperado; (ii) em que situações desenvolvedores tendem a explicitar suas preocupações com refatoração do código; e (iii) qual é o impacto de tais preocupações na efetividade de uma refatoração em melhorar a qualidade estrutural do código. Nós analisamos estas características e atingimos os seguintes resultados: Primeiro, conforme a complexidade das refatorações aumenta, a efetividade das mesmas aumenta conjuntamente. Segundo, há uma relação entre a efetividade de refatorações e preocupações explícitas com refatorações, onde a possibilidade de efeitos negativos é menor quando desenvolvedores estão explicitamente preocupados com refatoração. Finalmente, desenvolvedores

tendem a explicitar mais frequentemente suas preocupações com o processo de refatoração quando deparados com tarefas de refatoração mais complexas.

Palavras-chave

Refatoração; Atributos de Qualidade Interna; Refatorações Auto-Afirmadas; Requisitos Não-Funcionais; Preocupações de Desenvolvedores.

Table of contents

List of figures

List of Abreviations

SAR – Self-Affirmed Refactoring

NFR – Non-Functional Requirement

NFC – Non-Functional Concern

Numbers have an important story to tell. They rely on you to give them a clear and convincing voice.

Stephen Few, *Lecture.*

1
Introduction

The activity of refactoring consists of applying one or more types of transformations. Thus, developers focus on improving code quality as a means to achieve a higher degree of maintainability [23]. These refactorings can vary in effectiveness, depending on whether or not it successfully improves internal code quality attributes [11, 23, 27, 40]. For example, they might enhance cohesion as well as reduce coupling, complexity and size. These internal quality attributes are the standard metrics for characterizing software maintainability [2].

Another important characteristic of refactorings is that developers often apply them in a combined effort, even though they were proposed as singular transformations. These grouped transformations are known as *composite* or *batch* refactorings [8]. The joint application of various transformation types – i.e., more complex refactorings – is expected to affect the refactoring's likelihood of effectively improving code quality. These composite refactorings comprise about half of the refactorings applied in software projects [8, 34]. Thus, research about them has been growing in popularity in the last few years [8, 10].

Along with this, recent studies also described the many kinds of composite refactoring patterns used in practice [50]. They range from a simple, repeated usage of one or two different types of refactorings, to more complex code refactorings spanning over 5 different types of transformations [9]. This means that refactorings might not be only comprised of simple transformations in reality. They might also require far more complex change sets in order to fulfill their goal.

Regardless of refactoring complexity, most transformations primarily aim at improving the code structure and its maintainability. However, changes in which well-known code refactorings were performed might contain transformations related to other facets of software quality, represented by non-functional requirements (NFRs) [30]. This connection between NFRs and refactorings is strengthened by findings of empirical studies. They show a relationship between internal quality attributes and NFRs beyond maintainability, such as security [18, 33, 44], performance [19, 26, 45, 46] and robustness [12, 13, 28].

Although developers often apply refactorings in practice, their quality-related concerns when doing so may not be explicitly mentioned along a change. Thus, it can be difficult to determine how these concerns affect the refactoring application. The detection of such concerns can be achieved through the collection of:

Explicit mentions of refactoring-related terms. We define *explicit mentions of refactoring-related terms in commit messages, pull requests and issue discussions related to a commit that contained a refactoring* as "refactoring explicitness". These explicit mentions of refactoring-related terms in commit messages are also popularly known as self-affirmed refactorings (SARs) [3, 42]. An example of a commit message containing a SAR is as follows: "Decouple DefaultChannelPipeline from AbstractChannel. Motivation: DefaultChannelPipeline was tightly coupled to AbstractChannel which is not really needed. Modifications: Move logic of calling handlerAdded(...) for handlers that were added before the Channel was registered to DefaultChannelPipeline by making it part of the head context. Result: Less coupling and so be able to use DefaultChannelPipeline also with other Channel implementations that not extend AbstractChannel"[1].

Explicit mentions of concerns associated with NFRs. We define *explicit mentions of NFR-related terms in commit messages, pull requests and issue discussions related to a commit* as non-functional concerns, i.e., NFCs. An example of a commit message containing an NFC – in this case, of Performance – is as follows: "Remove WeakOrderedQueue from WeakHashMap when FastThreadLocal value was removed if possible. Motivation: We should remove the WeakOrderedQueue from the WeakHashMap directly if possible and only depend on the semantics of the WeakHashMap if there is no other way for us to cleanup it. Modifications: Override onRemoval(...) to remove the WeakOrderedQueue if possible. Result: Less overhead and quicker collection of WeakOrderedQueue for some cases."[2].

Once able to classify and detect developer quality-related concerns, both developers and researchers can use this information in order to better understand their influence in refactoring effectiveness. This understanding allows us to determine if such concerns can be potentially related to the (non-)positive effects found in refactoring applications.

Current studies show that refactorings are not always effective in terms of improving structural quality attributes [8, 9, 50]. Even though recent studies

[1]Message example adapted from https://github.com/netty/netty/, in commit a729e0fcd94009905d219665bdd069eb31433b7c

[2]Message example adapted from https://github.com/netty/netty/, in commit 640a22df9efb41e3d29b79916938c1c315be2872

attempt to understand potential factors that influence refactoring effectiveness, they mainly focus on analyzing if a refactoring is effective or not. Thus, it is still not known whether developer concerns, and not only those related to the maintainability of the code, have any relationship with the effectiveness of the refactoring transformations. Also, it is unknown if, for example, a complex self-affirmed refactoring applied when the developer explicitly showed concern with maintainability (and other non-functional requirements) can (or not) have a more positive impact on the software's structural quality.

Thus, one could hypothesize that by combining multiple transformation types, developers can better address major structural problems in the code [8, 34, 50]. These combinations of transformations could be considered "complex refactorings". Going even further, one might wonder if quality-related concerns of the developers during refactoring application could be related to the characteristics of the refactorings they perform in the code. Another possible hypothesis is if refactoring effectiveness is related to developers explicitly manifesting their concerns with the code's quality when applying them. However, there is a lack of knowledge if well-known refactorings [23] are more effective when developers perform changes while concerned with NFRs.

1.1 Problem Statement and Limitations of Related Work

This section discusses related work and provides statements of our research problem. We decompose our general problem in three specific research problems.

Limited knowledge on the relationship between refactoring complexity and effectiveness – In recent years, there has been a number of studies correlating refactoring effectiveness to other factors. Their goal is to understand which factors might be causing refactoring to fail to reach positive effects. For example, one of these investigated factors is the set of refactoring types being used together in a refactoring application [8, 9]. However, other factors of a refactoring might have a relation to their effectiveness, and should also be investigated. One such factor is the refactorings' *complexity*, which can be defined in a variety of ways. For instance, one aspect of refactoring complexity is the number of transformations, including repeating ones, applied in a refactoring. This aspect was explored in previous works, though results say that it has little correlation to the actual effectiveness of the applied refactorings [9]. However, other aspects of refactoring complexity might affect refactoring effectiveness. Aspects such as the number of files affected by a refactoring, or the number of different refactoring types applied in the same

change set, are still mostly unexplored. Then, our first problem is defined below:

> **Problem 1**: There is a lack of studies that attempt to understand to what extent certain aspects of refactoring complexity correlate to their effectiveness.

Limited knowledge on to what extent developers having explicit concerns with a refactoring may affect its effectiveness – As previously stated, studies mainly aim at analyzing the relationship between refactoring types and their effectiveness. Effectiveness in these studies is the ability of refactorings to remove code smells and improve internal quality attributes. However, the understanding of whether or not refactorings applied with the explicit intent of refactoring have any difference in results is still unexplored. Such an intent could be one of the factors related to the (non-)positive effects of refactoring seen in practice.

Through the usage of *self-affirmed refactorings* [3, 20], it is possible to discern between refactorings in which the developers were explicitly concerned with refactoring, and those in which they were not. With this, we are able to determine in which refactorings developers performed refactorings with intent to the point of turning such intent explicit through a SAR [3]. However, there is also a lack of studies that attempt to correlate the presence of self-affirmed refactorings with the effectiveness of the refactorings applied in the same change set. We can then define our second research problem as follows.

> **Problem 2**: There is a lack of understanding on to what extent the explicit concerns that developers have with refactorings in a change set relate to their effectiveness in improving code quality.

Limited knowledge on to what extent developer non-functional concerns relate to which refactorings they apply – As previously described, many studies attempt to understand whether or not developer refactorings are effective at removing specific code smells. These studies focus on not only this general effectiveness, but also if they are effective at removing the code smells they are supposed to remove [8, 16, 22]. Developers tend to follow Fowler's recommendations to refactor their code, combining them to form larger changes [29]. However, some refactorings only rarely have any positive effect, regardless of them being used in the recommended refactoring mechanics [1, 8, 15, 16].

Despite existing studies already analyzing this issue in more depth, there is still a search for potential factors that might be causing this lack

of positive effects in refactoring usage. One factor comprises is the NFCs during the developers' change. Thus, one should relate those concerns to the characteristics of the refactorings they have applied. By understanding this, it would be possible to better recommend refactorings to developers. For instance, a developer would not only rely on code smells present in the code as hints, but also developer concerns. This also allows for improving recommendations to developers that might not use refactorings only for code smell removal [30]. Given all these observations, we can frame our final research problem as follows.

Problem 3: There is still little understanding to what extent developers' non-functional concerns affect refactoring usage and effectiveness.

1.2 Main Research Contributions

In this context, this work focuses on understanding to what extent developers having explicit concerns with either refactorings or NFRs correlate to the characteristics of the applied refactorings. Not only that, but also the relation of such concerns to the effectiveness of the applied refactorings. Alongside this, we also analyze their relationship to the relevant refactoring characteristics, such as their complexity. Finally, our measurement of effectiveness is whether and how the internal quality attributes of the affected elements were changed. To reach the main goals of this work, we have achieved the following contributions:

Contribution 1: We report an analysis on to what extent refactoring complexity correlates to their effectiveness as well as determining which aspect of refactoring complexity is the one that truly affects effectiveness.

Through this contribution, we expect to be able to better recommend refactorings to developers based on their complexity. We could also aid developers in knowing which refactoring complexity aspects could be more worrisome during refactoring application. With proper guidance, more effective means of refactoring might be able to yield better results than what is currently being considered in the academy and practice.

Contribution 2: We report an analysis on to what extent explicit refactoring concerns, characterized by the presence of SARs in developer discussions, affect refactoring effectiveness.

Through this, we expect to be able to better guide developers in deciding whether or not specific situations might need a "focused" refactoring, i.e.,

performing an attentive refactoring as a primary goal. Similarly, we also expect to better guide developers in deciding in which situations they can apply refactorings as a secondary goal, combined with other, more concerning changes. We might also be able to suggest whether developers should need to be more concerned with the process they follow when implementing complex refactorings.

Contribution 3: We report an analysis on to what extent the presence of NFCs in developer discussions affect the effectiveness of applied refactorings.

With this contribution, we expect to better guide developers in understanding which refactorings can be used to improve code quality in ways other than through code smell removal. Alongside this, we also expect to better understand if some concerns might need to be separated from refactoring concerns. Another potential result is a new approach to recommending refactorings to developers that work in conjunction with other quality-related changes. One such example of this would be a refactoring that could potentially solve problems created by a performance-enhancing change.

1.2.1 Other Contributions

We also consider the following contributions as relevant, and somehow related to problems listed in Section 1.1. First, we offer a unique data set of eight projects, comprised of information about (i) the internal quality attributes of each element in each commit of a project; (ii) refactorings performed in each commit of a project; and (iii) the corresponding developer statements and discussions described in commit messages, issues, and pull requests. We also offer an automatic, keyword-based SAR classifier adapted from Ratzinger [42]'s proposed SAR classification method. By applying preprocessing techniques to the developer discussions, and adapting the keyword set, our approach achieves an F1-Score of 78%. Finally, we also offer a manually validated data set comprised of 775 commits spread across the eight projects, classified based on the presence of SARs, and which NFRs are discussed in each of the commits or their related issues and pull requests.

1.2.2 Publications

At the time of the defense of this dissertation, this research has one published paper [47] in which the author here is also the first author there.

We also contributed to the work of other researchers and collaborators, e.g., Bibiano et al [9]. Thus, part of our Master's research and results are also described in these co-authored papers. Table 1.1 lists the related publications in which I have worked on during the Master's period. Respectively, each column has the publication title, the submission's venue, its status, and whether it was directly derived from this work, or just provided partial contributions. Regarding the first four papers presented, all of them are already published or were accepted in well-recognized international and national venues, such as ICPC, ICSME and SBES. These are leading conferences in Software Engineering subareas and all of them have QUALIS scores varying from A1 to A3.

Table 1.1: Publications worked on during this research

Title	**Conference**	**Status**	**Relation to Master's Research**
On the Relation between Complexity, Explicitness, Effectiveness of Refactorings and Non-Functional Concerns [47]	SBES 2020 (QUALIS A3)	Published	Directly Derived
How Does Incomplete Composite Refactoring Affect Internal Quality Attributes [9]	ICPC 2020 (QUALIS A3)	Published	Related
Revealing the Social Aspects of Design Decay [6]	SBES 2020 (QUALIS A3)	Published	Related
Look Ahead! Revealing Complete Composite Refactorings and their Smelliness Effects. [10]	ICSME 2021 (QUALIS A2)	Published	Related
On the Influential Interactive Factors on Degrees of Design Decay: A Multi-Project Study	SANER 2022 (QUALIS A2)	Submitted	Related
Relating Complexity, Explicitness, Effectiveness of Refactorings and Non-Functional Concerns: A Replication Study	To be Defined	Planned	Directly Derived

1.3 Dissertation Outline

The remainder of this work, which is mainly structured as a combination of two of our technical papers, namely one published and one to be submitted, is organized as follows:

Chapter 2 presents the background, which introduces concepts related to (i) refactoring and its mechanics; (ii) refactoring characteristics; (iii) self-affirmed refactorings and refactoring explicitness; and (iv) non-functional requirements and software quality. Moreover, we also discuss related work.

In **Chapter 3**, we present our first study. We performed a preliminary analysis on four open-source projects, in order to determine the potential relations between refactoring complexity, explicitness and effectiveness, as well as NFCs. We first propose a potential relationship between refactoring complexity and effectiveness, then later correlate such relationship to both

refactoring explicitness and the presence of NFCs. This study consists of the paper "On the Relation between Complexity, Explicitness, Effectiveness of Refactorings and Non-Functional Concerns" [47], which was accepted on the Brazilian Symposium on Software Engineering (SBES) in 2020.

Chapter 4 presents our second study, a replication of the first study done with an improved and stricter methodology. We have also added four new projects, summing a total of eight. This study improves upon the analysis reported in Chapter 3, presenting new results. It consists of the paper "Relating Complexity, Explicitness, Effectiveness of Refactorings and Non-Functional Concerns: A Replication Study", which is to be submitted to a journal.

Finally, **Chapter 5** summarizes the conclusions of our work, presenting the main contributions, implications, as well as the future work.

2
Background and Related Work

This chapter contains the background and related work of this Masters' dissertation. Section 2.1 clarifies the concept of refactoring, as well as its mechanics. Section 2.2 discusses the characteristics of refactorings, such as their complexity and their effectiveness. Section 2.3 presents the concept of self-affirmed refactorings, which we use as refactoring explicitness. Section 2.4 describes the NFRs analyzed in this work, as well as how they relate to internal software quality. Finally, Section 2.5 concludes this chapter.

2.1
Refactoring and its Mechanics

Refactoring is a software maintenance process that perform changes on the code without altering its behavior in order to improve the code's quality [23]. This refactoring process can be performed in a variety of ways, from a simple renaming of a variable (*Rename Variable*) to more complex procedures. One example of a complex procedure would be transforming a specific repeated block of code with method calls to an external method with such block of code (*Extract Method*) [23]. Thus, by applying these different refactoring types, developers can remove software quality problems named *code smells* [56].

Code smells are intrinsically related to refactoring, as the refactoring process is the main procedure employed by developers to either mitigate or remove these code smells [23]. Code smells represent problematic structures in the code. They do not directly lead to bugs or errors, but cause the code to become more difficult to maintain [56], possibly influencing the introduction of bugs in the future. One such example of code smell is the *God Class*, which represents a class that contains too many responsibilities – and thus should be split into two or more classes [23]. In the literature, it has been proven that these smells do cause negative effects for the developers maintaining the code [37], and that developers do wish to fix such problems in actual, real-life scenarios [39].

The refactoring process is proven to at least mitigate the negative effects of code smells in most cases [9]. However, misuse of refactoring can,

in fact, even introduce more code smells than what the code had before their application [16]. Thus, it is not only necessary for developers to apply refactorings when needed, they also have to be mindful of which refactoring type to apply in which situation, in order to avoid degrading the software quality even further.

On the application of these refactorings, a distinction can be made between *single refactorings*, i.e., the application of only one code transformation in a single method/class; and *composite refactorings*, i.e., the application of multiple refactorings in a single (or a small group of) method/classes, in order to perform more sweeping changes to the code [32]. These composite refactorings can also be split into two groups, depending on which heuristic was used to detect them. *Commit-based*, which means that the composite was a group of refactorings applied within a single commit, or change, regardless of which methods/classes they were applied on; and *range-based*, which means that the composite was a group of refactorings applied over a series of commits, or changes, consecutively performed over the same small set of methods/classes [50].

Still on the same context of refactorings, the following previous pieces of work analyzed refactorings and their mechanics, as well as composite refactorings:

Bibiano et al [9] performed a study on five different open-source projects, in order to analyze the impact of incomplete composite refactorings in their quality attributes. They did so by first identifying the most common forms of incomplete composites, as well as their effects on internal quality attributes. Then, they analyzed a set of incomplete composite refactorings in five software projects, and how they affected the *Feature Envy* and *God Class* smells, as well as four internal quality attributes. In their results, they described that most incomplete composite refactorings tended to at least maintain the internal structural quality of classes that contained code smells, even if they failed to achieve positive results. Thus, this means that incomplete composite refactorings have a mostly neutral effect on internal quality, which might mean that there are factors related to composite refactorings that might affect refactoring effectiveness.

Fernandes et al [21] evaluated how composites compare to single refactorings in improving cohesion, (code) complexity, coupling, inheritance, and size of affected elements. They analyzed refactoring operations on 23 software projects, on a set in which nearly 50% of it was comprised of re-refactorings. This analysis was performed by measuring the effectiveness of both refactorings and re-refactorings on the five internal quality attributes.

They revealed that most operations improve attributes that are presumably associated with the refactoring type that was applied, though many (35%) still keep those attributes unaffected. They also showed that regardless of the refactorings' state as root-canal or floss refactorings, they mostly improved attributes, or at least did not worsen their state, a result differing from previous works. Thus, this means that refactorings may be applied in ways that achieve improvements differently from their expected intention – which could be related to developer concerns.

2.2 Refactoring Characteristics

As previously explained, refactoring is not necessarily a simple, standardized process. There are many refactoring types to be mindful of, along with the notion of single and composite refactorings. Also, these changes can have a variety of different characteristics that can be quantified, thus making it possible to differentiate each refactoring application. One such characteristic, which is the most explored by previous studies, is *refactoring effectiveness*. Refactoring effectiveness is the quantifiable metric that determines if the refactoring was able to improve code quality and, if it did, how intense was the change. In most cases, researchers use two different metrics for determining refactoring effectiveness: code smell density and diversity, as well as internal quality attributes [9, 36, 43].

Code smell density and diversity are metrics that quantify how many code smells are present in a single class/method (density), and how many different types of smells are present in the same class/method (diversity) [36]. Through these metrics, it is possible to more clearly see the effects of refactoring – a refactoring could, for example, simply remove a smell, thus reducing density, but not completely remove all smells from that type, thus keeping diversity the same. However, while these metrics do allow for the measurement of effectiveness, there is still the matter of refactorings being able to mitigate, yet not entirely remove code smells altogether – which these metrics are unable to detect.

Thus, researchers also use internal quality attributes to quantify refactoring effectiveness. The most commonly-used internal quality attributes, which are also the ones developers are most worried about, are Complexity, Cohesion, and Coupling [2]. Within these internal quality attributes, there are individual metrics that can be collected from the code, and that are directly related to the presence of specific code smells – as even the common code smell detection tools use such metrics for their detection [35]. Thus, with this, it is possible to

see the more slight effects of refactoring, i.e., smaller changes that can affect individual attributes positively or negatively, even if the refactorings do not completely remove or add an entire code smell.

Even though refactoring effectiveness is the most investigated characteristic, there are other ways of characterizing refactoring as well. One such characteristic is *refactoring complexity*. This complexity can be measured in a variety of ways – through the number of refactorings applied in a composite [9], through the amount of classes/methods changed by a single refactoring/composite, or through the number of unique refactoring types applied in a composite. While all of these aspects of complexity exist, there is still little to no research on the relationship between refactoring complexity and effectiveness, i.e., if more complex refactorings have higher effectiveness or not.

The following previous studies analyzed refactoring effectiveness:

Alshayeb [5] performed an empirical analysis of three Java systems, in order to evaluate claims that refactoring improves software quality. They evaluated possible correlations of the internal quality attributes of cohesion, coupling, (code) complexity, inheritance and size with the external quality attributes of adaptability, maintainability, understandability, reusability and testability. They performed this analysis by correlating nine metrics, split into internal quality attributes, with each external quality attribute. Then, they analyzed the trends of refactoring changes on each of these nine metrics, and were thus able to determine how refactorings affected each of the five external quality attributes. They concluded that there were no specific trends, as refactorings could just as commonly improve an attribute in some classes, while also worsen the same attribute in other classes of the same system. Thus, this means that refactorings may not always have positive effects on the code. This also means that analyzing specific measurable effects, instead of general trends, might be a better approach for analyzing the effect of refactorings.

Bavota et al [7] studied whether internal quality attributes or code smells relate with refactoring needs, based on 11 attributes (including size, coupling and cohesion) and 10 types of smells. To do so, they mined the history of three Java projects in order to investigate if refactorings do occur on code component in which indicators suggest they are needed. They defined these indicators as both critical values on quality-related metrics, as well as the presence of smells detected by tools. By performing this analysis, they were able to achieve results that indicate that quality metrics often do not show a clear relationship with refactoring. This means that refactoring operations are frequently applied in components that might not have any measurable factor of degradation. Even when considering code smells, only 42%

of refactorings were performed in classes that contained such smells. Thus, this might mean that developers' concerns during refactoring may not only regard the maintainability of the code, but also other potential concerns.

Chavez et al [17] investigated how root-canal and floss refactorings relate with internal quality attributes, including cohesion, coupling, (code) complexity, and inheritance. To do so, they performed an analysis of the version history of 23 projects, and collected refactoring information through the usage of RefactoringMiner. They classified refactoring effectiveness in two manners: if at least one metric of an attribute improved, or if most metrics of an attribute improved. With their results, they were able to notice that developers frequently apply refactoring operations to code elements with at least one internal quality attribute in a critically decayed state, which opposes previous works. They also described that only 65% of refactoring operations are able to improve their related internal quality attributes. They also described that root-canal refactorings more frequently improved internal quality attributes. Finally, they described that 55% refactorings performed together with other, non-refactoring aims (defined by the changes done in the code) had positive effects – however, only 10% had actual negative effects on the code. This difference means that the concerns that developers have during the refactoring process might be an important factor to consider in refactoring effectiveness.

The aforementioned pieces of work analyzed the relationship between a variety of external factors and refactoring effectiveness. However, neither of them had as one of their goals finding one (or more) aspects of refactoring complexity that might have any correlation to their effectiveness.

2.3 Self-Affirmed Refactorings and Refactoring Explicitness

Currently, researchers differentiate between refactorings with and without refactoring concern through the classification of *floss* and *root-canal* refactorings. They represent "a change set containing only refactoring applications" and "a change set containing both refactorings and non-refactoring changes", respectively [34]. While the differentiation between *floss* and *root* refactorings is effective at discerning refactoring intent, it is also very difficult to classify change sets as *floss* or *root canal*. Thus, usage of such classification is limited to manually-conducted studies, which have smaller, more concise data sets. On the other hand, another method was recently developed to attempt to classify changes as being primarily refactorings or not through the developers' eyes – the usage of *self-affirmed refactorings* [3, 20].

The presence of self-affirmed refactorings in software development has

been only recently explored. The term "self-affirmed refactoring" was coined in the last few years [3]. Self-affirmed refactorings are the phenomenon in which developers discuss about the refactoring process during, or soon after, the refactoring application. This is usually done by searching for specific keywords [42] or contexts [4] in commit messages in software repositories. Thus, the method of detecting refactoring concerns through self-affirmed refactorings only requires: (i) the extraction of developer discussions regarding the change set, and; (ii) a classification of these discussions based on whether or not refactorings are explicitly mentioned. Therefore, it is more viable for the purpose of extracting concerns than the classification between *floss* and *root* refactorings. By matching the presence of a refactoring in the code with refactoring discussion in the commit message, it is possible to determine whether or not the developer was concerned enough with the refactoring process in order to externalize it through the message explaining the changes performed in the commit.

The following previous studies analyzed refactoring explicitness:

Ratzinger [42], among many other contributions, proposed a phrase-based approach to detecting developer discussions related to refactorings. To do so, they performed an analysis of two out of the three projects analyzed in the work as a whole – as one of the projects did not apply enough refactorings to be considered. In this analysis, they manually classified refactorings as having either a single refactoring, multiple refactorings, or no refactorings. They then performed an iterative approach in attempting to identify refactorings using only commit messages as input. They first started by using the word "refactor", then expanded the keyword set every iteration. Using these keywords, they were able to label an average of 12% of changes as refactorings. This prediction of refactorings through commit message keywords also had a high accuracy, with 93% precision, and 98% recall. Thus, this means that self-affirmed refactorings are not common, comprising only 12% of changes. It also means that a keyword-based approach might be a valid way to classify commit messages as being self-affirmed refactorings or not.

AlOmar et al [2] performed an empirical study on self-affirmed refactorings in order to identify if frequently-used design metrics reflect what developers consider as quality. To do so, they extracted a group of design-related refactoring activities applied and documented by developers in 3795 Java projects. They also extracted a group of structural metrics and anti-pattern enhancement changes from these projects. With this, they were able to perform an analysis of the impact of those refactoring operations with regards to state-of-the-art metrics, in order to understand whether or not such metrics represent

developer concerns in practice. Results indicate that, for cohesion, coupling, (code) complexity, and inheritance, the academia-standard metrics do reflect developers' definition of quality. However, developers do not consider metrics such as encapsulation, abstraction, and design size often when determining software quality. Thus, this work is important as it proves that self-affirmed refactorings can be used as a factor for determining developer concerns. Alongside this, they also showed that developers mainly focus on cohesion, coupling, (code) complexity and inheritance when performing refactorings.

2.4 Non-Functional Requirements and Software Quality

The relationship between internal quality attributes and non-functional requirements was addressed by a number of empirical studies. In such studies, a strong relationship between four non-functional requirements – robustness, security, maintainability and performance – and specific internal quality attributes was described. Thus, we chose these four NFRs in order to be studied in this work. These non-functional requirements can be defined as follows:

Maintainability represents developer concerns with long-term maintenance of the code. It contains concepts such as code readability, good documentation and effective modularization of the code, among others. Refactoring is directly correlated to this NFR, as its main goal is improving maintainability. Some examples of commit messages with maintainability concerns, and which concerns they describe, can be seen as follows:

- "Change ImageDecodeOptions to match pattern from other options. Summary: Update ImageDecodeOptions to support extending this class (...)"[1]: In this example, developers describe changing a specific class in order to match the code pattern of other similar classes. They also describe changes to better support potential extensions in the future.
- "Renamed following the general naming convention used in Netty. Renamed 'delay' to 'checkInterval' Added some design ideas, TODOs, and FIXMEs."[2]: In this example, developers describe renaming a specific method to make its usage clearer. They also describe additions and improvements to code comments and documentation.
- "Rename PartitionFunction to BucketPartitionFunction. This commit renames 'PartitionFunction' to 'BucketPartitionFunction' and re-

[1]Message example adapted from https://github.com/facebook/fresco/, in commit 6c05bc6f3d69648a6a3129d866cbcac4b739f327

[2]Message example adapted from https://github.com/netty/netty/, in commit 48e258c8107bfd9baacf77f9815ebb781431a45e

introduces ‘PartitionFunction’ as an interface. Partitioning function is a general concept useful in different places (remote or local exchange partitioning or partitioning spiller for join’s spill) so it deserves simple expressive name.”[3]: In this example, developers describe a complex refactoring that starts by renaming an existing class to a more appropriate name. Afterwards, they create a new interface with the same name the class had before its renaming, since it could be useful in the future.

Robustness represents developer concerns with the project running properly, avoiding potential failures, as well as recovering from failures without issue. Thus, it contains concepts such as exception handling and error logging, alongside others. Some examples of commit messages with robustness concerns, and which concerns they describe, can be seen as follows:

– “Add information to TooManyBitmapsException for debugging (...)”[4]: This example represents developer concern with *exception logging*. This attribute of robustness encompasses all information regarding a specific exception type, which is especially useful in the case of frameworks, such as the system in this example.

– “Check if DnsCache is null in DnsNameResolver constructor. Motivation: We miss checking if DnsCache is null in DnsNameResolver constructor which will later then lead to a NPE. Better fail fast here. Modifications: Check for null and if so throw a NPE. Result: Fail fast.”[5]: This example represents developer concern with the robustness of the code, even if they do not directly fix the exception itself. However, by throwing the error earlier, other problems that could occur related to null-pointer exceptions at the previous point of error can now be more easily identified.

– “Use ChannelException when ChannelConfig operation fails in epoll. Motivation: In NIO and OIO we throw a ChannelException if a ChannelConfig operation fails. We should do the same with epoll to be consistent. Modifications: Use ChannelException Result: Consistent behaviour across different transport implementations.”[6]: In this example, developers describe a change from a more generic exception type to a more specific one. This, in turn, can allow for better exception handling and recovery

[3]Message example adapted from https://github.com/prestodb/presto/, in commit 1a4ac73ed993a1f818ff17c9196a67bdbb4f31dc

[4]Message example adapted from https://github.com/facebook/fresco/, in commit df4e4ca57e9febcd8ff69a95c7628edcc6437ed9

[5]Message example adapted from https://github.com/netty/netty/, in commit d7ff71a3d1d0ba16818ebe8ab44691197c2ffd48

[6]Message example adapted from https://github.com/netty/netty/, in commit 0c835420008bb1767ea1969cd8d63adf1c80e374

in the future, as well as allowing developers to better understand exactly what caused a specific failure.

Security represents developer concerns with limiting information access only to a specific subset of users, as well as blocking malicious access. Thus, it contains concepts such as information hiding, cryptography, and secure data transmission, among others. Some examples of commit messages with security concerns, and which concerns they describe, can be seen as follows:

- "JCBC-1203: Add CertAuthenticator and related checks. (...) This change brings in the CertAuthenticator as well as adds all kinds of sanity checks so that invalid auth combinations are rejected. (...) This changeset adds the CertAuthenticator (which can be used as a singleton) and then adds checks to the CouchbaseAsyncCluster so that invalid combinations depending on the environment settings are not allowed and error quickly. (...)"[7]: In this example, developers describe adding a new kind of authenticator to the system. They also describe the addition of a variety of new checks in order to determine which combinations of authorizations should be rejected.
- "Strip auth headers when redirected to another host. These are potentially private and we don't want to leak them to another host, regardless of whether they're created by the calling application or by the Authenticator."[8]: In this example, developers describe a change in which the system now removes authentication-related headers from requests when they are redirected to other hosts. This reduces the potential for information leaks, thus improving security.
- "Introduce Handshake as a value object. I needed a non-terrible way to provide the HTTPS handshake information to the async API. Previously we were passing the live socket around, which was leaky and gross. This creates a new value object that captures the relevant bits of the handshake. We can use it in the response, the connection, and also in the cache. It's plausible that in the future we can use it to allow the application to block requests if the handshake is insufficient."[9]: In this example, developers describe a change in which the system is now able to separate only the necessary information from a socket in order to perform

[7]Message example adapted from https://github.com/couchbase/couchbase-java-client/, in commit 88bc7f3003aee048ddd272fd850acaa457a13807

[8]Message example adapted from https://github.com/square/okhttp/, in commit ed70981925e64fd0cb593d09bdd401ea4ea19848

[9]Message example adapted from https://github.com/square/okhttp/, in commit e2bfa5dd6c0aee7d7e34b224a649500b9e5c267f

HTTPS handshakes. Thus, it eliminates potential leaking of information, improving security.

Performance represents developer concerns with both the runtime and resource usage of the code. Thus, it contains concepts such as memory usage, I/O frequency, and parallelization, alongside others. Some examples of commit messages with performance concerns, and which concerns they describe, can be seen as follows:

– "Remove WeakOrderedQueue from WeakHashMap when FastThreadLocal value was removed if possible. Motivation: We should remove the WeakOrderedQueue from the WeakHashMap directly if possible and only depend on the semantics of the WeakHashMap if there is no other way for us to cleanup it. Modifications: Override onRemoval(...) to remove the WeakOrderedQueue if possible. Result: Less overhead and quicker collection of WeakOrderedQueue for some cases."[10]: In this example, developers describe the removal of a specific ordered queue if it is not necessary. With this, they describe the results being a code that generates less overhead, quicker results, a clear sign of improved performance.

– "Give compiler hint about inline functions. Motivation: Some of the methods are frequently called and so should be inlined if possible. Modifications: Give the compiler a hint that we want to inline these methods. Result: Better performance if inlined."[11]: This example has developers describing a change in which they performed changes to meta-information of certain methods, in order to tell the compiler they should be inlined. As inlined methods have less runtime than delegated methods [26], this change improves code performance.

– "Lift Performance. Using 'f.lift()' directly instead of 'subscribe' improves ops/second on the included test from 5,907,721 ops/sec to 10,145,486 ops/sec."[12]: In this example, the developers describe a change in which, instead of subscribing to an event listener, the system now calls the function directly. And by doing so, improves code performance, as also described by the developers in the commit message.

The following previous studies analyzed non-functional requirements in relation with internal quality attributes. **Siegmund et al** [45] analyzed

[10]Message example adapted from https://github.com/netty/netty/, in commit 640a22df9efb41e3d29b79916938c1c315be2872

[11]Message example adapted from https://github.com/netty/netty/, in commit 7a3d91f43d12eb0b23a65662d26026c6cd451d76

[12]Message example adapted from https://github.com/ReactiveX/RxJava/, in commit 1ef689dd9200a915ba47ea5875bacf8a1ca8485d

whether and how refactorings can be used in order to optimize non-functional properties of software product lines. To do so, they performed a case study by fusing characteristics of refactorings with the feature-oriented programming, forming refactoring feature models. Then, they performed an analysis on the effects of refactoring applications on three non-functional properties: performance, footprint and coding styles. They then extended this analysis as an approximated influence of each refactoring on each non-functional property. As a result, specifically related to performance, they found that refactorings such as *Inline Method*, *Inline Class* and *Remove Middleman* can be used to improve performance. This is due to their removal of delegation, which can improve performance by up to 50%. However, they must still be applied with caution, as *Long Methods* can cause cache mismatches on the processors, which increases execution time.

Demeyer [19] hypothesized that the inclusion of polymorphism through refactoring did not reduce the performance of the refactored code. In order to prove this hypothesis, they performed an experiment by using a benchmark C++ code. This experiment contained a set of programs using conditional statements, and another program using polymorphism for the same task. They then performed a million executions on each program, using an array with 20 objects, and reported the average runtime. With their results, they discovered that, in fact, due to today's compilers and processors, the code that used polymorphism had higher performance than the code that did not. Thus, this means that refactorings are not to be avoided by developers that look to improve performance, but instead a tool they can use to improve it, depending on the situation.

Cacho et al [12, 13] hypothesized that mechanisms related to improving software maintainability could affect the improvement of software robustness. To prove such hypothesis, they conducted an empirical study to understand how changes in C# programs affected their robustness. They focused on changes in both normal and exceptional code, and how they affected exception handling faults. They applied a change impact and control flow analysis across many versions of 16 C# programs. Their results showed that most of the problems hindering software robustness in the analyzed programs stemmed from changes in the normal code. Alongside this, they also showed that, even when focusing only on changing exception handling, changes also introduced many potential faults. Finally, they showed that maintainability-driven changes on the exception handling mechanism introduces a flexibility that may lead to the introduction of faults. This means that maintainability-driven changes can cause effects on software robustness. However, this also means that developers

need to be more attentive when performing maintainability-driven changes, in order to avoid potential negative effects on robustness in the future.

Chowdhury et al [18] evaluated to what extent the metric of *coupling propagation* is effective at indicating security risks in software. To do so, they performed a case study on two Java projects, from two different security standards. They then collected metrics related not only to coupling propagation, but also other security-related aspects. Finally, they analyze the results of these metrics, and if they accurately show the security disparity these projects have. As a result, they found that coupling propagation is, in fact, related to actual software security. They theorize that this might stem from the fact that higher levels of coupling mean attackers can more easily access and compromise multiple sections of the code. Alongside this, high coupling would mean that messages that should be secure could go through classes that were more vulnerable. Through this, we can see that the ability of refactoring to reduce coupling can, in fact, be used to improve the security of a specific project.

While these works analyzed the applications of refactoring on the context of NFRs, all of them focused on how refactorings could be used in order to improve attributes specific to each NFR. Our work, on the other hand, focuses on how developer concerns with specific NFRs can affect the refactoring's effectiveness, both in general, and looking at each specific internal quality attribute.

2.5 Summary

This chapter provided the background to support the understanding of this dissertation. We presented basic concepts, used throughout the next dissertation chapters. We also discussed related work reporting studies on refactorings, its mechanics, and its characteristics, as well as studies on self-affirmed refactorings, and the relation between non-functional requirements and software quality. The next two chapters present the empirical studies that we conducted for addressing the problems listed on Section 1.1. For this purpose, we analyzed the potential relationships between refactoring complexity, refactoring effectiveness, refactoring explicitness, and the presence of non-functional concerns, to determine how these factors interact, and potentially discover ways to improve refactoring effectiveness.

3
On the Relation between Complexity, Explicitness, Effectiveness of Refactorings and Non-Functional Concerns

Previous studies have analyzed many different factors in order to determine which of them could influence refactoring effectiveness. Their goal is to understand the common non-positive effect of refactorings [8, 9, 50]. However, these works mainly focused on analyzing how the mechanics of the applied refactorings affect such effectiveness, rather than how the refactoring-related concerns of the developers performing the code transformations could do so. Thus, we conducted an analysis on four large software projects, collecting data related to refactoring complexity and effectiveness, as well as developer concerns related to refactoring, and four non-functional requirements. We then performed a manual validation of a sample of the data set, and analyzed the potential relations that could exist between refactoring complexity, and various developers' concerns, with regards to refactoring effectiveness. These concerns vary from explicit refactoring intents to typical non-functional requirements affecting refactoring-containing changes.

As such, in this chapter, the paper *On the Relation between Complexity, Explicitness, Effectiveness of Refactorings and Non-Functional Concerns* [47] will be presented in its entirety. The paper was accepted and published on the Brazilian Symposium on Software Engineering (SBES) in 2020. Due to this, Section 3.1 describes themes similar to those already presented in Chapter 1 of this dissertation, though with Chapter 1 describing it in more detail. Similarly, Section 3.2 covers the same related works as those discussed in Chapter 2 of this dissertation, with Chapter 2 once again being in more detail. The sections were kept in order to preserve the paper in its entirety, but they may be skipped due to the aforementioned repetition.

On the Relation between Complexity, Explicitness, Effectiveness of Refactorings and Non-Functional Concerns

Vinícius Soares,
Anderson Oliveira,
Juliana Alves Pereira
PUC-Rio, Rio de Janeiro, Brazil
[vsoares,aoliveira,juliana]@inf.puc-rio.br

Ana Carla Bibano,
Alessandro Garcia
PUC-Rio, Rio de Janeiro, Brazil
[abibiano,afgarcia]@inf.puc-rio.br

Paulo Roberto Farah[1,2]
Silvia Regina Vergilio[1]
[1]DInf-UFPR, Paraná, Brazil
[2]UDESC, Santa Catarina, Brazil
[prfarah,silvia]@inf.ufpr.br

Marcelo Schots
Universidade do Estado do Rio de Janeiro,
Rio de Janeiro, Brazil
schots@ime.uerj.br

Caio Silva,
Daniel Coutinho
PUC-Rio, Rio de Janeiro, Brazil
[csilva,dcoutinho]@inf.puc-rio.br

Daniel Oliveira,
Anderson Uchôa
PUC-Rio, Rio de Janeiro, Brazil
[doliveira,auchoa]@inf.puc-rio.br

3.1 Introduction

The refactoring activity consists of applying one or more refactoring transformations types, focusing on improving code quality as a means to achieve better maintainability [23]. Along software maintenance, developers perform improvements to non-functional requirements (NFRs) while also applying well-known code refactorings [30]. Even though developers often apply refactorings in practice, concerns about refactoring and NFRs are rarely explicitly mentioned along a change. Thus, one might wonder whether refactoring is more effective when an explicit manifestation of such concerns occurs. A refactoring is considered effective when it successfully improves internal code quality attributes [11, 23, 27, 40], such as enhancing cohesion or reducing coupling, complexity and size. Internal quality attributes are not only the academy-standard metrics for detecting problematic code [2]. Empirical studies (see Section 3.2) also show a relationship between these quality attributes and various NFRs beyond maintainability, such as security, performance and robustness. However, there is no knowledge if well-known refactorings [23] are more effective when developers perform changes with such NFR concerns in mind.

In addition, even though refactorings were proposed as singular transformations, developers often apply they in conjunction trough the so-called *batch* or *composite* refactorings. The joint application of various transformation types – i.e., more complex refactorings – might increase the likelihood of effectively improving code quality. Composite refactorings comprise about half of the refactorings applied in software projects [8, 34]. Thus, their research has been growing in popularity in the last few years [8]. Along with this, recent studies also described the many kinds of composite refactoring patterns used in practice [50]. They range from a simple, repeated usage of one or two different types of refactorings, to complex processes spanning over 5 different types of refactorings [9].

One could hypothesize that refactorings are expected to be more effective in improving code structural quality based on (i) the combination of multiple transformation types ("complex refactorings") in order to solve major structural problems in the code [8, 34, 50]; (ii) explicit mentions of concerns with refactoring (refactoring explicitness) – which is also popularly known as self-affirmed refactorings (SARs) [3, 42]; and (iii) explicit mentions of concerns associated with NFRs (these concerns are referred to in this work as NFCs).

Studies show that refactorings are not always effective in terms of improving structural quality attributes [8, 9, 50]. However, it is still not known whether the expectations above have any relationship with the effectiveness of refactoring. Existing studies only focus on analyzing if a refactoring is effective or not; they do not investigate to what extent their complexity, as well as their associated concerns, relates to its effectiveness. For example, a complex SAR with an explicit concern with maintainability (and other NFRs) might have a positive impact on the software's structural quality.

In this context, this study intends to conduct a preliminary investigation of whether and how the variation of refactoring complexity and its explicit concerns correlates to the improvement of internal quality attributes. To this end, we analyzed a total of 2,588 refactorings, obtaining information on both their effects on internal quality attributes and the amount of unique refactoring types they are composed of. We also developed and evaluated two keyword-based classifiers, one for SAR, and one for NFC detection in developer discussions – with the SAR detector reaching an F1-Score of over 80%. Through this analysis, we achieved the following findings:

Refactoring Complexity. We were able to determine that the complexity of a refactoring is impactful on their effects. As refactoring complexity increases, so does the chance of positive impacts; however, so does the risk of having a negative impact on the code. This reinforces the notion that refactoring complexity should be considered in studies and refactoring techniques. Thus, with proper guidance, a more effective means of refactoring can yield better results than what is currently being considered in the academy.

Refactoring Explicitness. An explicit concern with refactoring, surprisingly, more frequently affects their effectiveness negatively. Though many of the explicit refactorings were more complex, the developers did not always select refactoring type compositions that have shown themselves to be adequate to the improvement of structural quality. Less complex refactorings, composed of at most three types, had more positive effects – even if they were less effective overall. These results suggest that developers might need to be more concerned

with the process they follow when implementing complex refactorings, and not only the act of refactoring itself, as the improvement of code structure is not being achieved. These results could also be explained by the lack in tool support for complex refactorings, similar to what was found in other works [30, 51, 55].

Non-Functional Concerns. An explicit concern with NFRs may actually hinder the application of refactorings. However, when concerns with maintainability and robustness were present, refactorings generally had a more positive effect, while performance and security brought positive impacts on only one specific attribute each. This may also suggest a need for more optimization-based recommenders [38] to aid developers in making changes that positively affect necessary metrics without causing detriment to the others.

3.2 Related Work

This section classifies related works into three categories: *effectiveness*, *explicitness*, and *non-functional concerns*.

Effectiveness. Alshayeb [5] performed an empirical analysis of three Java systems, in order to evaluate claims that refactoring improves software quality. They evaluated possible correlations of the internal quality attributes of cohesion, coupling, (code) complexity, inheritance and size with the external quality attributes such as maintainability and testability. They concluded that refactorings rarely had positive effects on these attributes, being mostly neutral. Similarly, on the same context of internal quality attributes, Bavota et al [7] studied whether internal quality attributes or code smells relate with refactoring needs, based on 11 attributes (including size, coupling and cohesion) and 10 types of smells. Results showed that the analyzed refactorings focused on changing components whose quality metrics did not indicate problems in code.

Once again, regarding internal quality attributes, Chavez et al [17] investigated how root-canal and floss refactorings relate with internal quality attributes, including cohesion, coupling, (code) complexity and inheritance. They found that over 94% of applied refactorings have negative impacts on at least one internal quality attribute. Moreover, most refactorings improve their quality attributes, while others keep them unaffected.

On the context of complex refactorings, Bibiano et al [9] performed a study on 5 different open-source projects, in order to analyze the impact of incomplete composite refactorings in their quality attributes. They found that most incomplete composites have a neutral effect on internal quality

– neither increasing nor decreasing code quality. Also on the context of composite refactorings, Fernandes et al [21] evaluated how composites compare to single refactorings in improving cohesion, (code) complexity, coupling, inheritance and size of affected elements. Among the analyzed refactorings, 65% improved attributes associated with their refactoring types, while 35% kept them unaffected. Thus, while the aforementioned works studied potential correlations between refactoring effectiveness and other factors, they did not consider refactoring complexity and the presence of NFCs as potential factors.

Explicitness. The presence of SARs in software development has been only recently explored – with the term "self-affirmed refactoring" being coined in the last few years. Ratzinger [42] proposed a keyword-based approach to detecting developer discussions related to refactorings. In the explored context, the goal was the prediction of potential refactorings, which they achieved with a high accuracy. Likewise, for SAR-related research, AlOmar et al [2] performed an empirical study on SARs in order to identify if academia-standard design metrics reflect what developers consider as quality. Results indicate that, for cohesion, coupling, (code) complexity, and inheritance, the academia-standard metrics do reflect developers' definition of quality. However, for encapsulation, abstraction, and design size, there is a mismatch on how metrics were proposed and how they are used in practice. While both of the aforementioned works analyzed the presence of SARs, they did not analyze a potential correlation with the complexity and effectiveness of refactorings performed in the code.

Non-Functional Requirements The relationship between internal quality attributes and NFRs is addressed by a number of works. Thus, we focus our analysis on works addressing the NFRs that compose the NFCs analyzed in this work.

Regarding performance, Siegmund et al [45] analyzed refactorings' effects on the performance of software product lines. Refactorings such as *Inline Method* and *Inline Class* can reduce the execution time on method calls, thus improving performance. Also, Gotz et al [26] showed that removing code delegation and indirection can improve software performance by around 50%. Demeyer [19] reported performance improvements after replacing conditional statements with call methods through polymorphism. Some works also analyzed the relation between size, defined in terms of code statements, and performance [24, 41]. Smith et al [46] discussed performance anti-patterns based on coupling, cohesion, and (code) complexity of the inheritance hierarchy, concluding that God Classes are detrimental to software performance.

In the context of robustness, it is common to use exception flow infor-

mation [12, 13]. It aims to improve code reliability by providing constructs for sectioning code into exception scopes (e.g. try blocks) and exception handlers (e.g. catch blocks). Jakobus et al [28] evaluated the robustness of 50 projects by using the internal quality attributes of size and (code) complexity. Their findings suggest that exception handlers are usually simplistic, and that developers often pay little attention to exception scoping and behavior handling.

We also found works relating internal quality attributes to security. Chowdhury et al [18] evaluated how internal quality metrics of coupling, cohesion, and (code) complexity can indicate security risks in software. They concluded that size metrics can indicate structures that could be exploited to cause a denial of service attacks, while coupling can impact on how damage may propagate to other components of the software. Yet, their results showed that these metrics are not sufficient to indicate specific vulnerability types. Other studies [33, 44] also support these findings.

The identification of NFCs in text and developer discussions has also been explored. Lu et al [31] proposed an automatic classification of user reviews into concerns with four NFR types: reliability, usability, portability and performance. They evaluated the combinations of the classification techniques and machine learning algorithms with user reviews collected from two popular mobile apps from different platforms and domains. Results show that a combination of algorithms achieves an F-measure of 71.8%. Casamayor et al [14] applied a semi-supervised learning approach to identify NFCs in textual specifications, using a collection of requirements-related documents from 15 different software development projects, consisting of 370 mentions to NFRs and 255 functional ones.

These works differ from our work because, while they attempted to collect NFCs from project development history, they did not explicitly investigate the association of refactoring complexity, effectiveness, explicitness, and NFCs.

3.3 Methodology

In this section, we describe the methodology adopted in our study. The main goal of this work is to investigate to what extent the refactoring complexity, as well as their associated concerns (explicitness and NFCs), relates to its effectiveness, that is, to the improvement of internal quality attributes in the software.

3.3.1 Research Questions

Our analysis is guided by the following three research questions:

- **RQ1. Is the complexity of refactorings related to their effectiveness?** RQ1 is motivated by a search of a potential correlation between the complexity and effectiveness of refactorings. Thus, with this RQ, if this correlation exists, we aim at understanding its nature. We define complexity as *the number of different refactoring types that compose the applied refactoring*. Similarly, we define effectiveness as *the impact of the refactoring in improving internal quality attributes by improving their associated metrics*. We fully explain the reasoning for these definitions in Section 3.3.4.
- **RQ2. Are refactorings' complexity and effectiveness related to their explicitness?** Since the complexity of refactorings may not be the only factor that affects their effectiveness, RQ2 is built on top of the idea that this other factor may be the developers' explicit concern about refactoring. This explicitness is defined as *the presence of a SAR in either a commit message, issue or pull request (or comment) related to the changes where a refactoring was applied*. If this explicitness is related to the refactoring complexity and effectiveness, this RQ aims at understanding how it takes place.
- **RQ3. Do NFCs relate to refactoring effectiveness?** Finally, RQ3 aims at investigating if there are other concerns that affect the effectiveness of code refactoring. Thus, we define NFCs as *the presence of one of four analyzed NFRs in either a commit message, issue or pull request (or comment) related to the changes where a refactoring was applied*.

The choices and artifacts analyzed to answer the research questions are listed as follows:

Selection of Internal Quality Attributes. We chose to individually analyze each of the four internal quality attributes of cohesion, complexity, coupling and size for two reasons: (i) their connection to the NFRs chosen for this work (*e.g.*, size and complexity correlates to performance), and (ii) due to their uses in other works [9].

Selection of Non-Functional Requirements. *Maintainability* was chosen due to its potential likeliness of influence in refactoring, *i.e.*, it can be considered as an usual concern, as according to previous works. The other three NFRs were selected because they strongly relate with refactorings.

The increase of *security* can be linked to refactoring strategies that redesign application structures; preventing intruders from accessing sensitive code commands. Improvements in *robustness* can be reached by the reorganization of modules for integrating patterns geared at error handling (e.g, Chain of Responsibility [25]). Finally, *performance* can be increased through the detection, and subsequent refactorings, of code redundancies or a suboptimal distribution of code entities.

Figure 3.1 summarizes the methodology adopted to answer the research questions. First, we selected four projects, based on the criteria described in Section 3.3.2. Second, we collected data regarding internal quality attributes, refactorings and developer discussions, which is described further in Section 3.3.3. Third, we performed an analysis of the collected data to obtain composite refactorings (see Section 3.3.4) – which combine two or more refactoring types. In addition, we collected data about refactoring effectiveness, as well as the presence of SARs and NFCs. Finally, we used this data to answer each of the aforementioned research questions.

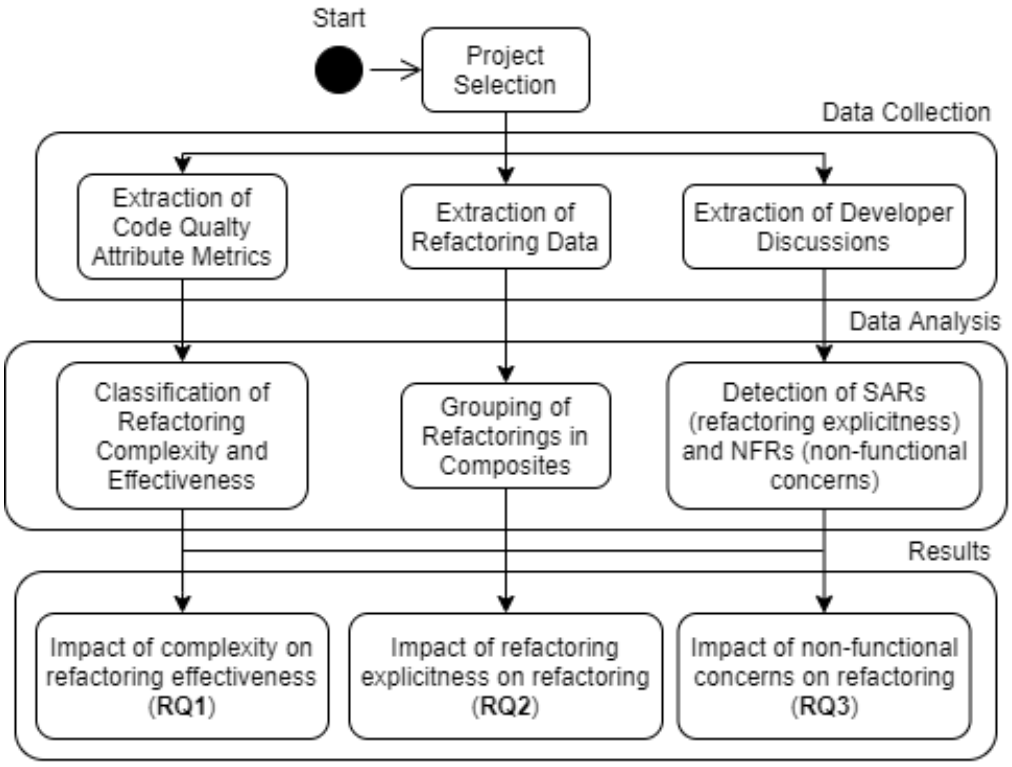

Figure 3.1: Adopted methodology.

3.3.2 Project Selection

Due to our focus on understanding the impact of complex refactorings in real-world projects, we selected 4 projects – Couchbase Java Client, Dubbo,

OKHttp and JGit –, using the following criteria:

Java Open Source Projects. The project must be open-source, and developed primarily in Java. This increases this work's replicability. Alongside this, Java has the most support from outside tools, especially for internal quality analysis – which are required for this work; **Number of Contributors and Activity.** The project must still be actively worked on at the time of analysis, and must have a considerable number of contributors. Active projects are more likely to accurately represent the state of the industry. Similarly, the large amount of active contributors in these projects allow us to have a larger selection of discussions; **Project Age.** The project must have been in development for at least 5 years. Architecture degradation over time is more clearly seen in older projects, which would then require developers to more frequently apply refactorings.

Variety of Refactoring Types. The project must have a large variety of different refactoring types (*i.e.*, one of the refactorings as defined by Fowler et al [23]) being used along its maintenance. This is important due to our focus that is on refactoring complexity – which is defined as the amount of different refactoring types used in a composite or single refactoring; **Presence of Composite Refactorings.** The presence of composite refactorings in the project must have been proven in other works. Once again, due to our focus being on the analysis of complex refactorings, the presence of composite refactorings in the analyzed projects is important. Thus, we decided to choose projects in which the presence of composite refactorings was already proven by other works, such as [9].

3.3.3 Data Collection

Thus, in order to attain the goals described previously, we extracted data from three main points of view: (i) *Internal Quality Data*, which is necessary for all **RQ**s; (ii) *Refactoring Data*, which is also necessary for all **RQ**s, and; (iii) *Developer Discussions*, for **RQ2** and **RQ3**. Their collection would enable their correlation, thus allowing an understanding of refactoring complexity's effect on internal quality attributes. It also allows an understanding of a correlation between the presence of SARs and NFCs in changes that contain refactorings, and the complexity and effectiveness of these refactorings. The collection of this data was done as follows:

Internal Quality Data. In order to obtain data related to internal quality attributes, we must first collect the metrics that compose such attributes. To this end, we used a tool called Understand [54]. Understand is a static code

analysis tool, which collects the internal quality metrics from each element, in each commit, of each project. This then allows us to view a continuous trend in the change of each metric in the project's history. In turn, this enables a detection of how changes improved – or worsened – the state of each attribute. However, Understand has a very large amount of metrics related to the size and complexity of the code, and lacks metrics for code cohesion. Thus, in order to balance the amount of metrics in each attribute, we selected a set of 15 metrics, which are listed in the work's companion website [48].

Refactoring Data. For the data set related to the second point of view, we focused on which kinds of refactorings were used during the project's development, as well as how they form composites. In order to detect and classify refactorings based on the types proposed by Fowler [23], we used RefMiner [52], a tool that collects refactoring information from the history of Java projects. RefMiner has a high reported precision of 98%, and a recall of 87% [52], which makes it a reliable tool for refactoring detection.

Developer Discussions. Finally, for the third point of view, we focused on attempting to extract the messages and discussions written by developers when changing the code. Thus, we extracted the following items from the projects' repositories, by using the GitHub API: (i) Commit Messages; (ii) Related Issues and Pull Requests; and (iii) Developer Comment Discussions. While we were able to extract commit messages for all 4 projects, 2 of them (Couchbase Java Client and JGit) did not have Issue and Pull Request information on their GitHub repositories, since they utilize an external issue/review tracker. However, this distinction in information availability may prove useful in understanding how much SAR and NFC detection is capable of detecting potential candidates with limited information.

3.3.4 Data Analysis

Once the aforementioned data was collected, we started the process of data analysis. While the main steps and results are described in this paper, additional information about it can be found in the work's companion website[48]. The analysis consisted of combining internal quality attribute and refactoring complexity data with the presence of SARs and NFCs. Thus, this allowed us to answer the proposed **RQ**s (Section 3.3.1). This analysis was performed as follows:

Refactoring Complexity and Effectiveness. Once we had access to both refactoring-related and internal quality attribute data, we were able to determine what would consist as "refactoring complexity", and how this would

affect their effectiveness. Originally, we intended to determine complexity as a combination of how many refactoring types were used in a composite/single refactoring, as well as how many code elements were affected. This metric for affected elements was defined as the amount of unique classes and methods related to the refactoring in question. However, we decided against using this metric, as it did not have any relation with refactoring effectiveness in our analysis. Another potential complexity metric for refactoring exists – the amount of single refactorings that are part of the same composite, regardless of type. Nonetheless, we decided against using it, as it was already seen to not affect refactoring effectiveness in other works [8].

Thus, we defined the metric for refactoring types as the number of unique detected refactorings that composed either a composite or single refactoring. In total, we were able to detect up to 52 unique refactoring types, at method and class level – such as *Extract Class*, *Move Method* and *Split Attribute* –, through the use of RefMiner. The full list of refactoring types is in the work's companion website [48].

To determine refactoring effectiveness, we analyzed each of the four internal quality attributes (complexity, cohesion, coupling and size) individually, analyzing how a change in other parameters (refactoring complexity, presence of SARs and NFCs) affected each attribute. In order to classify a change as *positive*, *negative* or *neutral*, we used the following criteria: (i) if there was no positive (reduction) nor negative (increase) change in any metric related to the attribute, the change was *neutral*; (ii) if at least one of the metrics related to the attribute had its value changed, and most metrics changed positively, the change was classified as *positive*; (iii) in any other case, it was classified as *negative*. Thus, by combining this effectiveness data with complexity data, we could better understand if they relate with each other, and if so, how they relate. Thereby, we were able to answer **RQ1**.

Composite Refactoring. Due to analyzing refactoring complexity – we needed to group the previously obtained refactorings into composites. To do so, we first used the *range-based* heuristic, as defined by Sousa et. al. [50]. This heuristic groups refactorings that affect similar elements in different points of time as a composite, which would allow us to analyze changes that spanned multiple commits. Afterwards, we grouped the remaining refactorings into composites through a *commit-based* heuristic, also proposed by Sousa et. al. [50]. This heuristic groups refactorings that were applied in the same commit as a composite refactoring. Then, the remaining refactorings were classified as single refactorings. Thus, we were able to group the highest possible amount of interrelated refactorings into composites, in order to better detect the

complexity of the refactorings used by the developers when changing the code. By doing so, we can combine this data with the internal quality attribute metrics in order to answer **RQ1**.

Presence of Self-Affirmed Refactorings. By using the previously collected developer discussions, we were able to automatically detect and classify which refactorings had a correlated discussion in which a SAR was present. This was done through keyword-matching, by using a set of 11 keywords and 8 phrases, which are listed in the work's companion website [48]. The original set of keywords was based on Ratzinger's work [42], though we changed the set in order to improve its accuracy for the data set we used in this work. Thus, by combining this data with the previously collected complexity and effectiveness data, we answered **RQ2**.

Non-Functional Concerns. By using the previously collected developer discussions, we were able to automatically detect and classify them if they are NFCs or not. This was also done automatically through keyword-matching, by using a set of 69 keywords, which are listed in this work's website [48]. We decided on the keyword set based on previous manual analyses of the projects' issue/PR messages. However, as we describe further in Section 3.4, this NFC classifier has a very low accuracy, but a relatively high recall.

Considering this, we decided to focus our analysis on a manually validated refactoring sample, designated by the classifier. Thus, we had a final analyzed group of 196 refactorings. From this group, 92 refactoring changes did not mention NFRs, while the other 104 did. In addition, 33 refactorings were related to maintainability, 20 to security, 36 to performance, and 40 to robustness – with potential intersections between these classifications. As such, the combination of this manually-validated data with all the aforementioned data sets allows us to answer **RQ3**.

3.4 Validation

With the goal of determining the reliability of the automatic detection of SARs and NFCs, we performed manual validations for each, determining their precision and recall.

3.4.1 Self-Affirmed Refactoring Validation

For the manual validation of SARs, we selected a set of 124 different commits split into 63 commits from OKHttp, and 61 commits from Couchbase Java Client. The set was also divided equally between three different groups:

(i) commits classified as SAR, and also classified as refactorings by RefMiner; (ii) commits classified as SAR, but not classified as refactorings by RefMiner, and; (iii) commits that were not classified as SARs.

We decided to validate commits from these two projects for the following reasons: (i) both have a high frequency of SARs in relation to other projects – enabling a more accurate check of which keywords are problematic and which are missing; and (ii) While OKHttp has available information for commits, issues, pull requests and comments, Couchbase Java Client only has commit messages available. Thus, we can test if an actual difference exists in the accuracy when less information is available. Likewise, the three groups were decided for the following reasons: (i) ensuring that all keywords in the set were verified. To do so, we selected SARs from the keyword sets with less occurrences, and only later did we select those with more common keywords; (ii) being able to detect if an actual refactoring was present when a SAR was falsely detected.

We then performed a manual validation with 3 participants – all 3 being authors of this work, and knowledgeable in the context of refactoring. 2 authors classified a set of 42 random commits, while one last author classified a set of 40. These classified sets were also equally balanced between the three sets described previously. In this validation, the participants were asked to identify the following: (i) if there was any explicit mention of a term or phrase that would lead to a direct association to a refactoring; (ii) which phrase led them to understand identify the SAR, and; (iii) which keywords were related to it. The full results of this validation are available in this work's companion website [48].

Through the completed validation, we can quantify the precision and recall of the SAR detector. From this data set, it had a precision of 81.2%, and a recall of 91.5% – leading to an F1-Score of 86%. Individually, the precision was slightly lower for Couchbase Java Client in relation to OKHttp. However, the classifier had over 80% recall in both projects. This can mean that the unavailability of other kinds of discussions in two of the four projects analyzed in this work might not have much impact in the final results.

3.4.2 Non-Functional Concern Validation

Similarly to the SARs validation, we also made efforts in attempting to manually validate the NFC classifier – which was needed, due to the complexity involved in identifying NFCs on textual data. First, we performed a validation of 1200 issues from OKHttp. The data was split into those that

were detected as NFC and those that were not. This first validation was made with 6 participants, all of them computer science master/doctorate students knowledgeable in the context of NFRs. They were asked to classify the issues as one (or more) of the NFCs used in this study (maintainability, robustness, security and performance). They were also asked to determine which phrase led them to their decision, and which keywords in that phrase could be used to identify and classify that NFC.

Once again, we chose OKHttp due to the high frequency of NFR-related discussions in that project, as well as the presence of issues and pull requests as potential discussions to be analyzed. However, we found that the NFC classifier precision was below-average (50.43%), even if the recall was good (82.66%) for this data set. As such, we thought that, by allowing the classifier to focus on commits in which changes actually occurred, and not only issues and pull requests (which could be closed without any action in the code itself), we could more accurately classify in terms of their related concerns.

Thus, we then specialized our validation for a new set of 553 commits in which refactorings occurred – in order to perform the analysis for finding a potential trend between the presence of NFCs and refactoring effectiveness. This data set was split between three projects: OKHttp, Couchbase Java Client and JGit. Thus, we could balance the set between projects in which we have full information available, as well as projects in which we have only commit information. We also attempted to balance the data set between mentions to all four detected NFRs.

This validation was performed by 5 participants – once again, with all of them being computer science master/doctorate students knowledgeable in the context of refactoring. The validation process was very similar to the previous one, though now the participants were also asked to describe how directly the NFR was mentioned (if it was). In this case, they should identify if the NFR was directly mentioned in the discussion, or if it could be indirectly derived from the text under analysis.

Through this second validation, we can quantify the precision and recall of the NFC classifier – leading to a precision of 53.90% and a recall of 72.13%, correlating to an F1-score of 61.70%. The low precision might make the use of the automatically classified data not a good approach for a quantitative analysis. However, the good recall makes using it to collect a sample of NFC discussions for validation – as discussed in Section 3.3.4 – a valid approach.

3.5 Results and Discussions

In this section, we present the results that answer the **RQ**s introduced in Section 3.3.1. We begin by analyzing the correlation between refactoring complexity and effectiveness (**RQ1**), then correlating our findings to the presence of SARs (**RQ2**) and, finally, correlating all findings to the presence of NFCs (**RQ3**). The numerical results for each of the analyses described in this paper are available on the project's website [48].

3.5.1 Refactoring Complexity vs. Effectiveness

At first, we attempted to understand how much the increase in refactoring complexity affects each internal quality attribute of the refactored code. To this end, we grouped refactorings into 5 categories, based on the number of different refactoring types they were composed of. Categories 1 to 4 contained refactorings with 1 to 4 refactoring types, respectively, while category 5+ contained those with 5 or more refactoring types, since these were rare occurrences and, by combining them, we have a more balanced data set.

Figure 3.2 shows that, for the more complex refactorings – containing 4 or 5+ refactoring types –, the proportion of refactorings that had neutral effects on the code decreased in all 4 attributes. In average, the difference between 1 to 5+ was 21.2%, with cohesion having a drastic reduction of 51.4%, and high reductions in coupling and complexity. Conversely, the proportion of positive and negative changes also increased. We can also notice that, proportionately, the frequency of negative changes increased more than those of the positive changes. In some cases (namely coupling and size), the number of negative changes even overcame the number of positive changes. We observed a significant difference in the distribution of effectiveness between refactoring complexity levels for all quality attributes. The difference is with 95% confidence ($\alpha = 0.05$) using Kruskal-Wallis Chi-Squared test and Dunn's post-hoc test with Bonferroni adjustment, except for cohesion (p-value > 0.6). These increases in the magnitude of effects for more complex refactorings follow what would be expected – since, when adopting more types of refactorings, the possibility of making significant improvements (or deteriorations) in code quality attributes is higher. Thus, complex refactorings could be a risky, but potentially fruitful, endeavor.

Conversely, this increased complexity could also lead to more mistakes during its execution, and thus potentially cause an increase in negative effects. This increase in negative effects we experienced, thus, may be due to the lack

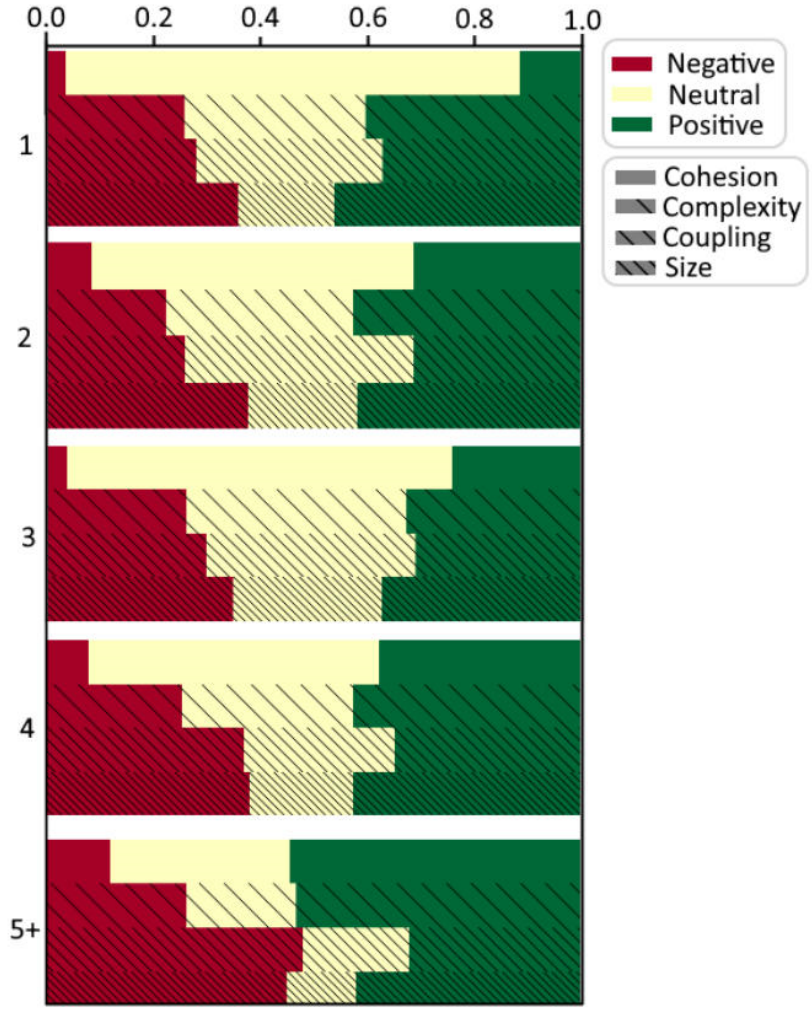

Figure 3.2: Distribution (decimal percentage) of effects based on the refactoring complexity. Each 0.1 on the horizontal scale represents 10% in change frequency of the corresponding effectiveness.

of tool support for the application of complex refactorings spanning multiple refactoring types.

> **Finding 1**: **(RQ1)** The more complex the refactorings are, the higher their impact on structural code quality. This impact can be both positive and negative.

Implications. Our results are interesting, since previous studies that analyzed other ways of defining refactoring complexity – such as the number of refactoring instances in a composite [8], or the number of commits in a refactoring [8, 50] – found no relation between these complexity indicators and internal quality attributes. In contrast, our results indicate that the diversity of types in a refactoring tend to impact internal quality attributes. As such,

future studies may consider this indicator when evaluating the impact of refactoring complexity in internal and external quality attributes. From a practical point of view, developers can benefit from our findings of how the addition of more refactoring types to a composite affects the code: the increase in the risk of causing negative changes, yet the potential for more frequent positive changes.

3.5.2 SARs vs. Complexity and Effectiveness

By using the information of which refactorings were self-affirmed or not, we are able to combine them with our previous results to uncover more findings. To this end, we analyzed the correlation between refactoring complexity and its self-affirmation, through SARs. Thus, Figure 3.3 presents the frequency of self-affirmed (Has SAR) and non self-affirmed (No SAR) refactorings composed of 1, 2, 3, 4, or 5+ refactorings. It can be seen that, while comparing the SAR and No SAR sets, respectively, the proportion of single refactorings severely decreased (57% to 34%), while the proportion of complex changes with 5 or more refactorings severely increased (7% to 28%). The results are statistically significant, using Wilcoxon Rank-Sum test at 95% confidence level ($\alpha = 0.05$).

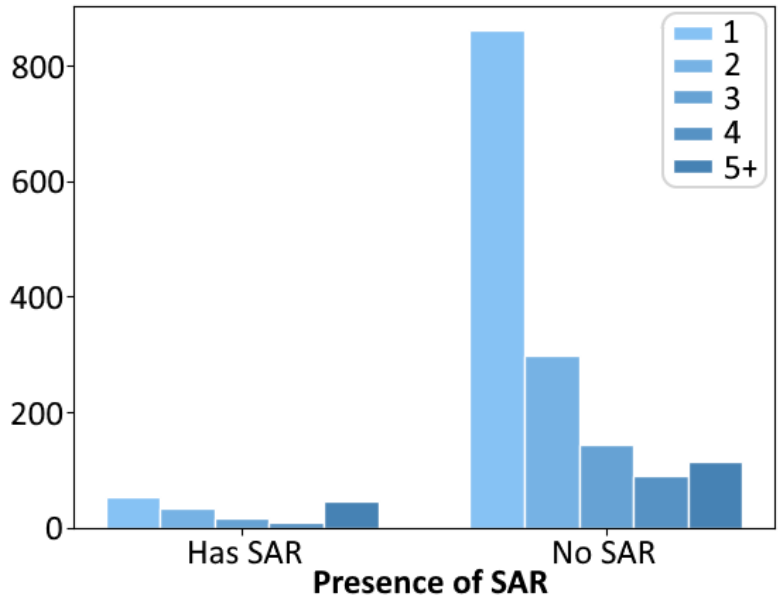

Figure 3.3: The frequency of self-affirmed and non self-affirmed refactorings composed of 1, 2, 3, 4, or 5+ refactorings.

Thus, our findings reveal that, when the primary focus of developers is to perform a refactoring – which could be indicated by a self-affirmation of their refactoring [2] – they tend to perform more complex, and thus, more impactful, refactorings.

Finding 2: **(RQ2)** Developers tend to perform more complex refactorings when they manifest an explicit concern with refactoring.

Figure 3.4 presents the effectiveness of self-affirmed (Has SAR) and non self-affirmed (No SAR) refactorings. As discussed, we define effectiveness as the improvement of cohesion, complexity, coupling and size. As in Section 3.5.1, the frequency of neutral changes also decreased, in average, 7.4%. The results for effectiveness between SARs vs non-SARs are statistically significant using Wilcoxon rank sum test at 90% confidence level, except for cohesion (80% level).

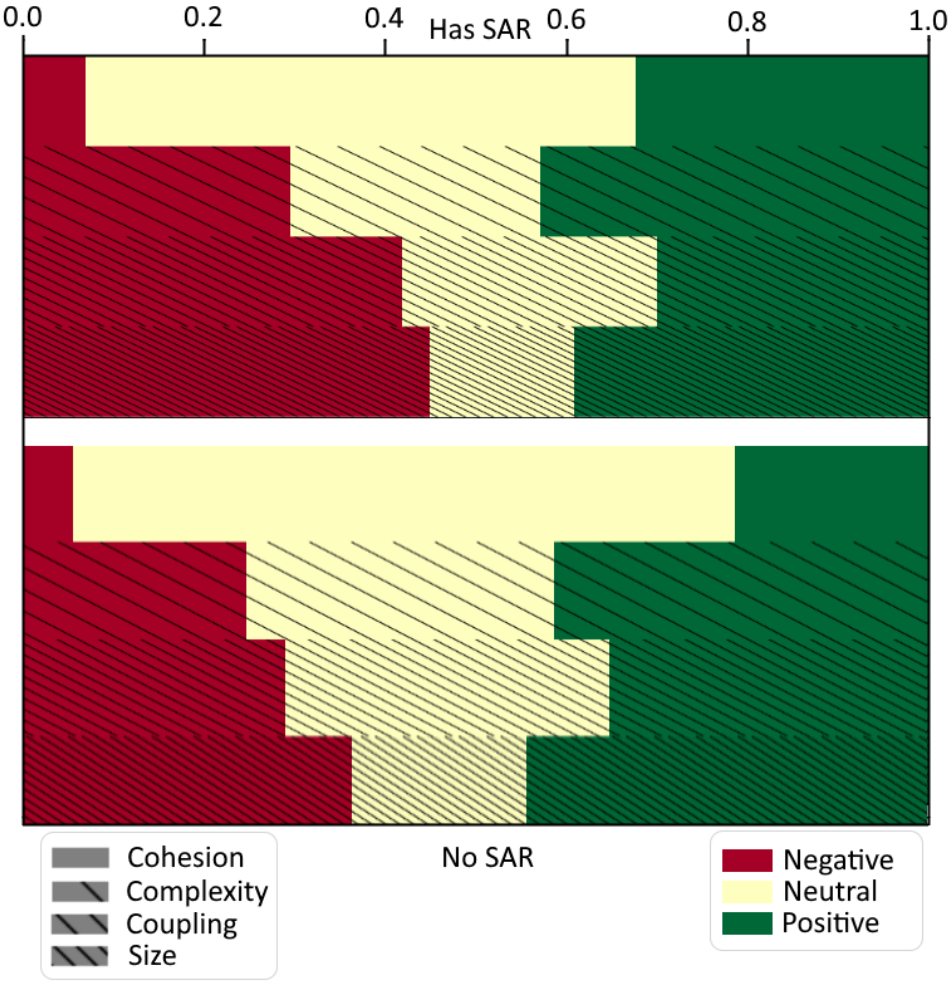

Figure 3.4: The negative, neutral and positive effect of self-affirmed and non self-affirmed refactorings. Each 0.1 on the horizontal scale represents 10% in change frequency of the corresponding effectiveness.

Moreover, we also observed that the proportion of negative changes for SARs increased much more expressively (more than ten times) than the proportion of positive changes. Thus, our results indicate that having refactorings as the developers' primary focus may actually hinder their effectiveness

– potentially due to this increase in complexity causing more mistakes in the refactoring process. One possible reason for these results may be due to the lack of tool support for applying complex refactorings, as described in other works [30, 55, 51]. Therefore, we can expect that developers are performing refactorings mostly manually. Thus, we can summarize the aforementioned results as the following finding:

> **Finding 3**: **(RQ2)** Self-affirmed refactorings tend to have a more negative effect on the code than their counterparts.

Implications. Our results indicate that, when developers explicit their concern in the refactoring process, they tend to do more complex refactorings – though they also tend to perform worse. This may imply that developers have a higher concern with the refactoring process when they apply complex refactorings. Thus, they tend to explicit their refactorings concern on the commit messages. Consequently, it is surprising that they also yield more negative results. Knowing this, developers should be more aware, beyond just the concern they demonstrated with refactorings, of the process they follow when implementing refactorings. This is important, as the main basic objective of refactoring, improving the structure of the refactored code, is not being reached.

Interestingly, we also found that over 30% of the validated sample was detected as an SAR by both the manual validation as well as the automatic detector – but was not detected as a refactoring by RefMiner. Alongside this, over 50% of the total commit messages (spanning all projects) were detected as SARs and were not detected by RefMiner. Thus, this might mean that developers frequently apply refactorings that differ from those defined by Fowler [23], and are thus not detected by RefMiner. We also observed that developers often customize even single refactorings, similar to was reported by Tenorio et al[51] – which would impact their detection by RefMiner, as they do not follow the steps defined by Fowler [23].

3.5.3 NFCs vs. Complexity and Effectiveness

In this section, we perform an analysis of how NFCs related to each of the four NFRs (maintainability, security, performance and robustness), can affect both the complexity and the effectiveness of their related refactorings. As described in 3.3.4, we only used a set of 196 refactorings for this analysis, split into a set of 92 which do not contain mentions to NFRs, and a set of 104 that do. Thus, Figure 3.5 presents the frequency of refactorings composed of 1,

2, 3, 4, or 5 or more refactoring types – divided into a set of changes in which there was an explicit concern with one of the four NFRs, and one in which there was not.

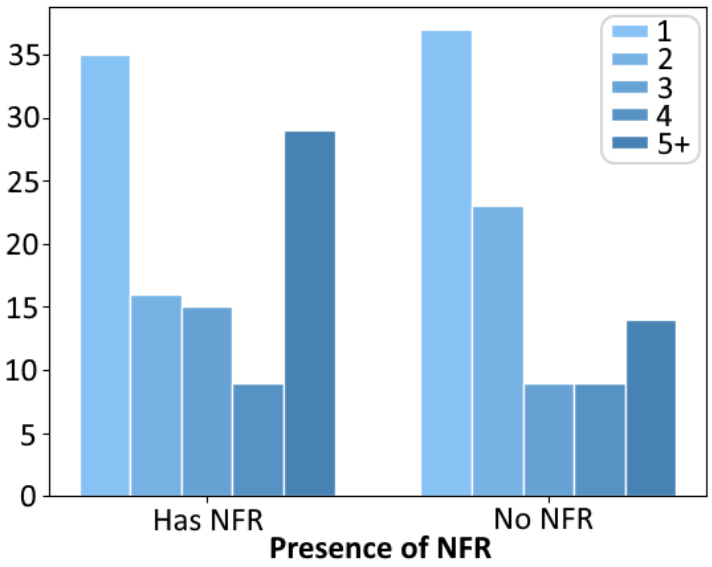

Figure 3.5: The frequency of refactorings composed of 1, 2, 3, 4, or 5 or more refactorings grouped by the presence of mentions to NFRs.

From this, we can observe that, very similarly to when refactorings are explicitly mentioned, complex refactorings are more frequent when the developers express an explicit concern with NFRs. We found no statistical difference (p-value > 0.6) between them. This is potentially a result of the the NFR detector's low accuracy, leading us to use a small validated data set – further motivating a search for more effective automated NFR detection. Thus, this can be summarized into the following finding:

Finding 4: **(RQ3)** Developers usually perform more complex refactorings when they are explicitly attentive to NFRs.

Finally, we analyzed the impact of refactorings when coupled with NFCs. First, by analyzing the difference between NFR-related changes for each of the 4 NFRs, we saw that the increase in the negative effects of refactorings was significant – with no apparent increase in its positive effects. However, positive changes were still more frequent than negative changes. This correlates to the findings described in Section 3.5.2 – in which developer concerns reduce the neutral effects of refactorings, but also increase the possibility for negative effects. We also focused on the analysis of how mentions to each individual NFR impacts on the effectiveness of refactorings. Figure 3.6 displays the results

of this analysis, by showing how each concern relates to affecting the internal quality attributes. The attributes were analyzed individually by grouping their related metrics.

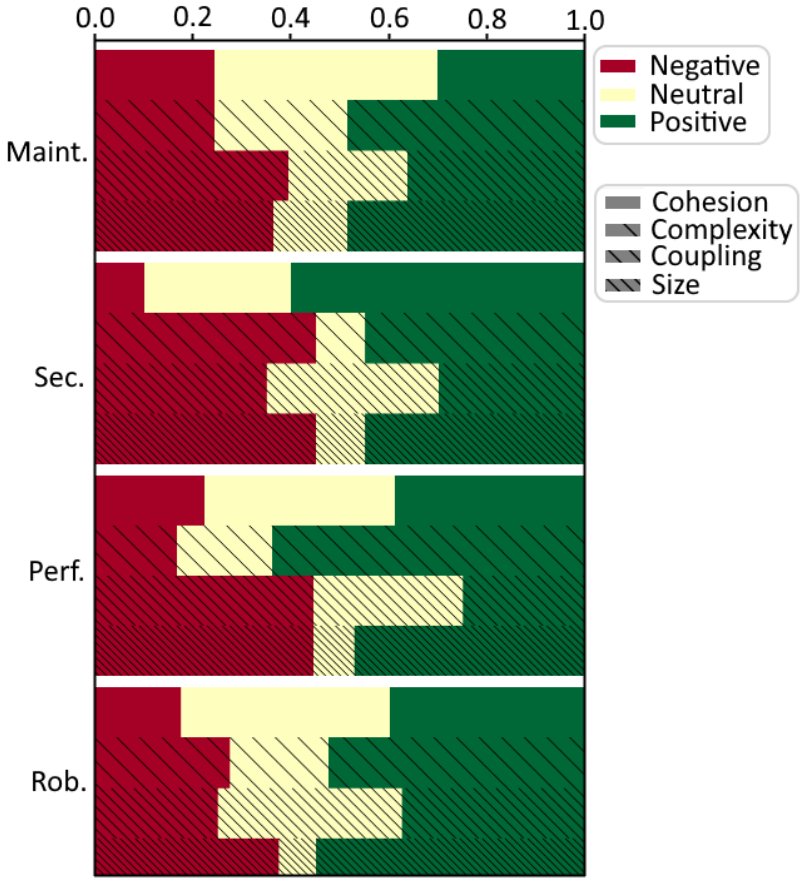

Figure 3.6: The negative, neutral and positive effects of refactorings when coupled with changes in NFRs. Each 0.1 on the horizontal scale represents 10% in change frequency of the corresponding effectiveness.

As expected, refactorings related to maintainability affect all internal quality attributes rather equally, and with more positive than negative effects, in general. Refactorings related to robustness also have a similarly-distributed effect, though with less pronounced negative effects. This is interesting, as maintainability would be the NFR most related to refactoring itself, yet it tends to cause more negative changes, in general, when compared to robustness.

Conversely, refactorings related to either security or performance are very focused – tending to change a single attribute very positively, while negatively affecting all others. In the case of security, we found that changes were usually focused in improving cohesion between elements. Following the principle of information hiding – it is expected for security-related changes to improve the cohesion of the code, by keeping information close to only the classes that use them. However, the fact that they often increase coupling in the code is surprising as, according to Chowdhury et al [18], high coupling can be

considered a security flaw, as it may cause a higher propagation of potential attacks.

In the case of performance, we found that changes were usually focused on reducing code complexity. Some works also correlate performance to code complexity-related changes, such as the addition of polymorphism, even with an increase in the number of sentences [19]. This also corroborates with our findings, as we detected that refactorings related to performance have a high frequency of negative effects in size-related metrics. We found statistical significance for internal quality attribute changes between the NFR types at 95% confidence level for coupling, and at 65% level for cohesion. But, we did not find statistical difference for code complexity and size. Thus, we can summarize these results in the following finding:

> **Finding 5**: **(RQ3)** Refactorings concerned with maintainability and robustness have a somewhat positive effect distributed through all metrics. Refactorings concerned with performance and security, however, have very focused effects – drastically improving one attribute with respect to the others.

Implications. We observed that when developers explicit their concerns with NFRs, they tend to perform more complex refactorings, with usually negative impact. However, by looking more closely at the impact of each concern, we can see that, usually, maintainability- and robustness-related concerns still have a mostly positive effect on the code. Security- and performance-related concerns, however, tend to cause mostly negative effects on the code, even if they do improve one particular attribute severely. As such, we can derive that these other concerns that developers may have can cause them to disregard the refactoring process itself, thus causing the increase in the frequency of negative changes.

We can also derive that specific concerns may even cause developers to focus entirely on one specific attribute, with negative impacts over most other attributes. Thus, when implementing refactorings, developers should once again be more aware of the followed process – since balancing the improvement of necessary attributes without negatively affecting others may change the situation found in these results. This also motivates potential search-based solutions, since there might be a lack of external support for suggesting potential changes that attempt to balance improvement for most affected attributes.

3.6 Threats to Validity

Although we attempted to mitigate them to the best of our ability, this work contains some threats to its validity, as follows:

Generalizability. We selected 4 projects from different fields (database, network, distributed computing and git integration), and from different developers. However, the results we found might still not be generalizable to other contexts (e.g., closed-source projects).

Accuracy of Refactoring Detection. Though RefMiner has high reported accuracy in many different works, we did not directly evaluate its accuracy for our set of projects. However, other works have used RefMiner for a similar set of projects [9], and have reported good accuracy in its detection.

Accuracy of Self-Affirmed Refactoring Detection. Though we did manually validate a sample of the data set in order to identify the detector's accuracy, it still relies on a keyword-based classification to determine whether or not a refactoring is self-affirmed. Thus, the detector might not be generalizable to other projects, and its accuracy might change with new updates to the analyzed projects.

Accuracy of NFC Classification. The accuracy of the NFC classifier is still below-average, even when applied to a specialized data set, which makes it unreliable for analysis. However, we attempted to mitigate this threat by only using a manually validated data set from a sample of instances detected by the classifier.

3.7 Final Remarks

This work attempted to understand the relationships between: (i) refactoring complexity; (ii) refactoring effectiveness; (iii) refactoring self-affirmation, and; (iv) the presence of NFCs during the refactoring process, in improving internal quality attributes. We performed a quantitative analysis of 2648 refactorings, from four different open-source projects. Our results demonstrate that developers tend to apply more complex refactorings when they are explicitly concerned with either the refactoring process, or NFRs (i.e., security, performance, robustness and maintainability). We also observed that complex refactorings are both more impactful in affecting the code quality, and much riskier than single refactorings. This is due to the fact that, the more complex the refactorings are, the higher their positive and negative impact on structural code quality.

These findings, when combined with other studies, such as the one proposed by Tenorio et al [51], which proposes that automated refactoring tools may not currently provide support for customized refactoring, may in fact present an even more entrenched problem. These available tools in commonly-used IDEs only support simple, standardized single refactorings – having little or no support for complex, customized composites. A possible explanation for the drastic increase in the negative impacts of complex refactorings might be, thus, that they had to be performed manually, with little aid from supporting tools. Thus, our work intends to motivate tool developers into improving support for more complex refactorings composed of multiple different types.

Our results might also drive researchers to understand why developers tend to make worse refactorings when they explicitly mention their concern with refactoring. Our findings can also motivate practitioners and researchers in analyzing how to perform complex, yet effective, refactorings. This can be helpful, as complex refactorings seem to be more effective at actually impacting the code than their simpler counterparts.

As future work, we plan on further analyzing how other factors can impact the refactoring process. We also plan on investigating specific patterns of complex refactorings, and how they relate to their effectiveness. Finally, we aim at extracting developers' goals during the refactoring process to assess whether additional non-refactoring goals affect how developers refactor their code.

3.8 Summary

In the study presented in this chapter, we performed the initial steps into solving the problems presented in Section 1.1.

For Problem 1, i.e., the lack of understanding of to what extent refactoring complexity correlates to their effectiveness, we have seen that the complexity aspect of *number of unique transformation types composing a refactoring* is the one most related to refactoring effectiveness. With this, we also discovered that complex refactorings, containing a higher variety of transformations, are more impactful. Their simpler counterparts, containing only one or two transformation types, have a higher chance of neutral effects. We described complex refactorings as being able to offer a higher chance of positive effects on the refactored design, though at a risk of a higher chance of introducing negative effects as well. These results reinforce that existing techniques [23] need to be extended to progressively support developers while they are working on the various transformations and their compositions to produce a complex

refactoring.

For Problem 2, i.e., the lack of understanding to what extent explicit refactoring-related concerns relate to refactoring effectiveness, we observed that, in general, developers explicit their concerns with refactorings through SARs more often when performing more complex refactorings. However, refactorings accompanied by SARs surprisingly tended to have more frequently negative effects overall than their non-SAR counterparts, even if the frequency of positive effects was also higher. This observation reveals that developers need refactoring guidance specialized in a stepwise fashion as they progress along complex refactoring transformations. Our study is the first to consider both SARs and non-SARs, and our observations encourage future research to do the same.

Finally, for Problem 3, i.e, the lack of understanding on how NFCs affect refactoring usage and effectiveness, we observed that, in general, developers explicit their concerns with NFCs more often when performing more complex refactorings. Alongside this, they perform more balanced refactorings, i.e., affecting all quality attrbutes equally, when concerned with maintainability and robustness. In the case of security and performance, however, the refactorings they perform are very skewed towards improving a specific internal quality attribute to the detriment of others. This result shows that techniques should present alternatives to developers in which the other quality attributes could be either maintained or improved.

In the next chapter, we present a replication of the study presented in this chapter, though with additions to the dataset and the methodology. The goals of this replication are to (i) increase both the total and the validated data set, to check the generalizability of the results; (ii) improve the validation process, with a stricter methodology; (iii) understand how different SAR classifiers operate under the same data set, and which is more appropriate for our set; and (iv) improve our refactoring detection through the usage of RefactoringMiner 2.0, a new version of the refactoring detection tool used in the aforementioned work, which can detect a higher variety of refactorings, with higher accuracy [53].

4
Relating Complexity, Explicitness, Effectiveness of Refactorings and Non-Functional Concerns: A Replication Study

The first published work performed through this Master's research (Chapter 3) revealed various findings regarding the relation between refactoring complexity, refactoring explicitness, NFCs, and refactoring effectiveness [47]. However, many new developments occurred in the area after the publication of that work. RefactoringMiner, the tool used for refactoring detection in the original work, was updated to version 2.0, which included a variety of new detected refactorings and with increased accuracy [53]. Alongside this, a state-of-the-art machine learning-based approach for detecting self-affirmed refactorings was developed [4]. Finally, there was a possibility of improving certain parts of the methodology and increase the generalizability of its results.

Thus, we conducted an analysis on four new software projects as well as re-analyzing data from the original four projects used in the previous work (Chapter 3). We once again collected data related to refactoring complexity and effectiveness, as well as developer concerns related to refactoring and four non-functional requirements. We then performed a strict manual validation of a sample of the data set, and performed a comparative analysis between our SAR classifier and the state-of-the-art approach. Then, we then analyzed the potential relations that could exist between refactoring complexity, refactoring concerns, and non-functional concerns, with regards to refactoring effectiveness.

As such, in this chapter, the paper *Relating Complexity, Explicitness, Effectiveness of Refactorings and Non-Functional Concerns: A Replication Study* will be presented in its entirety. The paper's title is still temporary, as it is still to be published in a journal in the future. Due to this, Section 3.1 describes themes similar to those already presented in Chapter 1 of this dissertation, though with Chapter 1 describing it in more detail. Similarly, Section 3.2 covers the same related works as those discussed in Chapter 2 of this dissertation, with Chapter 2 once again being in more detail. The sections were kept in order to preserve the paper in its entirety, but they may be skipped due to the aforementioned repetition.

Relating Complexity, Explicitness, Effectiveness of Refactorings and Non-Functional Concerns

A Replication Study

Vinícius Soares, Anderson Oliveira, Juliana Alves Pereira
PUC-Rio, Rio de Janeiro, Brazil
[vsoares,aoliveira,juliana]@inf.puc-rio.br

Ana Carla Bibano, Alessandro Garcia
PUC-Rio, Rio de Janeiro, Brazil
[abibiano,afgarcia]@inf.puc-rio.br

Paulo Roberto Farah[1,2] Silvia Regina Vergilio[1]
[1]DInf-UFPR, Paraná, Brazil
[2]UDESC, Santa Catarina, Brazil
[prfarah,silvia]@inf.ufpr.br

Marcelo Schots
Universidade do Estado do Rio de Janeiro, Rio de Janeiro, Brazil
schots@ime.uerj.br

Caio Silva, Daniel Coutinho
PUC-Rio, Rio de Janeiro, Brazil
[csilva,dcoutinho]@inf.puc-rio.br

Daniel Oliveira, Anderson Uchôa
PUC-Rio, Rio de Janeiro, Brazil
[doliveira,auchoa]@inf.puc-rio.br

4.1 Introduction

Refactoring is a key part of software maintenance, consisting of the application of one or more transformations on the code, focusing on improving its quality in order to increase the code's maintainability [23]. Thus, a refactoring is considered effective when it successfully improves internal code quality attributes [11, 23, 27, 40], such as enhancing cohesion or reducing coupling, complexity and size, which are the academy-standard metrics for detecting problematic code [2]. Empirical studies (see Section 4.2) also show that these internal quality attributes are not only able to quantify software maintainability, but also have relationships with other non-functional requirements, such as security, performance and robustness.

However, recent studies show that refactorings are not always effective in terms of improving structural quality attributes [8, 9, 50]. These studies primarily focus on analyzing if a refactoring is effective or not with regards to the refactorings' main characteristics, i.e., which refactoring type is being used, and to combat which smell. Thus, this means that there are still key refactoring characteristics that are still lacking in investigation in order to determine to which extent they relate to refactoring effectiveness. These characteristics include refactoring complexity, as well as the explicitness of the developers' associated concerns when performing such refactorings. For example, a complex refactoring in which the developer explicitly mentioned their own concern with the code maintainability could have a different effect on the software's structural quality.

Refactorings were originally documented only as singular transformations. However, developers often apply them in composition, through *batch* or *composite* refactorings. Recent studies have described the many kinds of composite refactoring patterns used in practice [50]. They can range from a simple, repeated usage of one or two different transformation types, to com-

plex processes spanning over 5 different transformation types [9]. This joint application of various transformation types seems to have some level of correlation to the effectiveness of those refactorings in the improvement of code quality [47]. Currently, these composite refactorings comprise about half of the refactorings applied in software projects [8, 34]. Thus, research on these composite refactorings has been growing in popularity in the last few years [8].

Alongside the application of well-known code refactoring transformations, developers can have a variety of different concerns, including the improvement of code characteristics related to its conformity to non-functional requirements (NFRs) [30]. However, even though these refactorings are often applied in practice, explicit mentions about refactoring, and NFR-related concerns are not trivial to be detected alongside such changes. A previous study developers seemed to be applying more complex refactorings when they explicitly manifested their refactoring and NFR-related concerns, and that refactorings could have recurring effects when paired with a specific NFR-related concern [47].

In this context, this study intends to replicate and extend the study performed by Soares et al [47]. To this end, we analyzed a set comprised of the four projects analyzed by Soares et al [47], as well as four other projects, thus doubling the size of the original work's data set in terms of project count. Alongside this, due to the upgrade to RMiner 2.0 [53] and the updates to improve the methodology, we have more than tripled the total number of analyzed refactorings of the original work, obtaining a total of 8,408 refactorings. We have obtained information on both these refactorings' effects on internal quality attributes as well as their complexity (number of unique transformation types). We used the same keyword-based classification method as the original work, with the same keyword list, which maintained a relatively high F1-Score of 78%, even with the addition of the new projects. Through this analysis, we achieved the following findings:

Refactoring Complexity. We were able to confirm that, just like in the original work, the complexity of a refactoring has a correlation to its effect. As refactoring complexity increases, both the potential of positive and negative changes increase as well. Thus, this observation reinforces the importance of studying refactoring complexity even further, showing that it is a factor that should be considered when designing and analyzing refactoring techniques. With proper support, the application of complex refactorings might be able to achieve truly positive results, when compared to the more commonly-seen non-positive effects.

Refactoring Explicitness. We were once again able to confirm some of the results of the original work, as we saw that refactorings in which developers showed explicit concern with the refactoring process tended to be more frequently complex than other refactorings. However, differently from the original work, we have not seen an increase in negative effects. On the contrary, negative effects were less common in refactorings that were self-affirmed. Thus, this shows that developers may perform refactorings more effectively if they are primarily concerned with the refactoring process itself. This does not, however, exclude the possibility that the lack in tool support can be problematic for complex refactorings, as was also stated by other works [30, 55, 51].

Non-Functional Concerns. We were unable to replicate the results of the original work with regards to concerns with NFRs relating to refactoring effectiveness. This could be justified by the original work not having a large enough commit sample, as there is no automatic way to classify NFRs and, thus, they had to use a manually-classified sample. In this work, we have seen that when developers are concerned with NFRs, especially with Maintainability, there is an increase in positive effects, and a decrease in negative effects – though this increase is not very noticeable for Security and Performance concerns. This, in turn, may suggest that developers concerned with the quality of the code, in its many aspects, perform more directed, and thus, more effective changes. This may suggest that a deeper analysis of potential correlations between refactoring applications and NFR-related concerns can lead to interesting results.

4.2 Related Work

The primary related work for this study is the original work we have replicated and extended in this study: “On the Relation between Complexity, Explicitness, Effectiveness of Refactorings and Non-Functional Concerns”, by Soares et al [47]. In this work, the authors performed an analysis on a set of 4 projects, looking to understand the possible relationships between refactoring complexity, refactoring explicitness, and refactoring effectiveness. Thus, it differs from our work due to our work extending upon this previous work’s methodology and data set. Due to this, we have confirmed some of the findings of this previous work, though in some cases, we have found differing results.

The remainder of this section is split into the three following groups: *effectiveness*, *explicitness*, and *non-functional concerns*.

Effectiveness. Alshayeb [5] performed an empirical analysis of three Java systems, in order to evaluate claims that refactoring improves software quality. They evaluated possible correlations of the internal quality attributes of cohesion, coupling, (code) complexity, inheritance and size with the external quality attributes such as maintainability and testability. They concluded that refactorings rarely had positive effects on these attributes, being mostly neutral. Similarly, on the same context of internal quality attributes, Bavota et al [7] studied whether internal quality attributes or code smells relate with refactoring needs, based on 11 attributes (including size, coupling and cohesion) and 10 types of smells. Results showed that the analyzed refactorings focused on changing components whose quality metrics did not indicate problems in code.

Once again, regarding internal quality attributes, Chavez et al [17] investigated how root-canal and floss refactorings relate with internal quality attributes, including cohesion, coupling, (code) complexity and inheritance. They found that over 94% of applied refactorings have negative impacts on at least one internal quality attribute. Moreover, most refactorings improve their quality attributes, while others keep them unaffected.

On the context of complex refactorings, Bibiano et al [9] performed a study on 5 different open-source projects, in order to analyze the impact of incomplete composite refactorings in their quality attributes. They found that most incomplete composites have a neutral effect on internal quality – neither increasing nor decreasing code quality. Also on the context of composite refactorings, Fernandes et al [21] evaluated how composites compare to single refactorings in improving cohesion, (code) complexity, coupling, inheritance and size of affected elements. Among the analyzed refactorings, 65% improved attributes associated with their refactoring types, while 35% kept them unaffected. Thus, while the aforementioned works studied potential correlations between refactoring effectivness and other factors, they did not consider refactoring complexity and the presence of non-functional concerns (NFCs) as potential factors.

Explicitness. The presence of SARs in software development has been only recently explored – with the term "self-affirmed refactoring" being coined in the last few years. Ratzinger [42] proposed a phrase-based approach to detecting developer discussions related to refactorings. In the explored context, the goal was the prediction of potential refactorings, which they achieved with a high accuracy. Likewise, for SAR-related research, AlOmar et al [2] performed an empirical study on SARs in order to identify if academia-standard design metrics reflect what developers consider as quality. Results indicate that, for

cohesion, coupling, (code) complexity, and inheritance, the academia-standard metrics do reflect developers' definition of quality. However, for encapsulation, abstraction, and design size, there is a mismatch on how metrics were proposed and how they are used in practice. While both of the aforementioned works analyzed the presence of SARs, they did not analyze a potential correlation with the complexity and effectiveness of refactorings performed in the code.

Non-Functional Requirements The relationship between internal quality attributes and NFRs is addressed by a number of works. Thus, we focus our analysis on works addressing the NFRs that compose the NFCs analyzed in this work.

Regarding performance, Siegmund et al [45] analyzed refactorings' effects on the performance of software product lines. Refactorings such as *Inline Method* and *Inline Class* can reduce the execution time on method calls, thus improving performance. Also, Gotz et al [26] showed that removing code delegation and indirection can improve software performance by around 50%. Demeyer [19] reported performance improvements after replacing conditional statements with call methods through polymorphism. Some works also analyzed the relation between size, defined in terms of code statements, and performance [41, 24]. Smith et al [46] discussed performance anti-patterns based on coupling, cohesion, and (code) complexity of the inheritance hierarchy, concluding that God Classes are detrimental to software performance.

In the context of robustness, it is common to use exception flow information [12, 13]. It aims to improve code reliability by providing constructs for sectioning code into exception scopes (e.g. try blocks) and exception handlers (e.g. catch blocks). Jakobus et al [28] evaluated the robustness of 50 projects by using the internal quality attributes of size and (code) complexity. Their findings suggest that exception handlers are usually simplistic, and that developers often pay little attention to exception scoping and behavior handling.

We also found works relating internal quality attributes to security. Chowdhury et al [18] evaluated how internal quality metrics of coupling, cohesion, and (code) complexity can indicate security risks in software. They concluded that size metrics can indicate structures that could be exploited to cause a denial of service attacks, while coupling can impact on how damage may propagate to other components of the software. Yet, their results showed that these metrics are not sufficient to indicate specific vulnerability types. Other studies[44, 33] also support these findings.

The identification of NFCs in text and developer discussions has also been explored. Lu et al [31] proposed an automatic classification of user reviews into concerns with four NFR types: reliability, usability, portability and

performance. They evaluated the combinations of the classification techniques and machine learning algorithms with user reviews collected from two popular mobile apps from different platforms and domains. Results show that a combination of algorithms achieves an F-measure of 71.8%. Casamayor et al [14] applied a semi-supervised learning approach to identify NFCs in textual specifications, using a collection of requirements-related documents from 15 different software development projects, consisting of 370 mentions to NFRs and 255 functional ones.

While these works analyzed the applications of refactoring on the context of NFRs, all of them focused on how refactorings could be used in order to improve attributes specific to each NFR. Our work, on the other hand, focuses on how developer concerns with specific NFRs can affect the refactoring's effectiveness, both in general, and looking at each specific internal quality attribute.

4.3 Methodology

In this section, we describe the methodology adopted in our study. Due to this being a replication, and extension, of a previous study, similarities and differences between the two methodologies will be highlighted. Similar to the original work, the main goal of this work is to investigate to what extent refactoring complexity, as well as quality-related developer concerns (explicitness, and NFR-related concerns), relates to its effectiveness, i.e., the improvement of the software's internal quality attributes.

4.3.1 Research Questions

Our analysis is guided by the following three research questions, similarly to the original work:

– **RQ1. Is the complexity of refactorings related to their effectiveness?** In the original work, the authors proposed a potential correlation between refactoring complexity and their effectiveness. Thus, with RQ1, we aim at not only possibly confirming its existence, but also understanding its nature. In this work, we use the same definition for complexity as the original work, i.e., *the number of different refactoring types that compose the applied refactoring*. Similarly, we also use the same definition for effectiveness as the original work, i.e., *the impact of the refactoring in improving internal quality attributes by improving their associated metrics*. We fully explain the reasoning for these definitions in Section 4.3.4.

- **RQ2. Are refactorings' complexity and effectiveness related to their explicitness?** The original work also showed a potential correlation between refactoring explicitness and both their complexity and effectiveness. Thus, RQ2 attempts to confirm the existence of such correlation, as well as understanding how it affects the correlated factors. This explicitness follows the same definition as the original work, i.e., *the presence of a SAR in either a commit message, issue or pull request (or comment) related to the changes where a refactoring was applied.*
- **RQ3. Do NFCs relate to refactoring effectiveness?** Finally, the original work showed a potential correlation between NFCs and refactoring complexity and effectiveness. Thus, RQ3 has at its goal an attempt at confirming the existence of such correlation, as well as understanding how it affects the correlated factors, similarly to how RQ2 analyzes refactoring explicitness. In this work, we define NFCs in the same manner as they were defined in the original work, i.e., as *the presence of one of four analyzed NFRs in either a commit message, issue or pull request (or comment) related to the changes where a refactoring was applied.*

The choices and artifacts analyzed to answer the research questions are listed as follows:

Selection of Internal Quality Attributes. We kept the same internal quality attributes used in the original work: cohesion, (code) complexity, coupling and size. These attributes are not only already used by a variety of other works [9], but they are also connected to the NFRs chosen for this work (*e.g.*, size and complexity correlates to performance).

Selection of Non-Functional Requirements. We also kept the same NFRs as the original work: Maintainability, Robustness, Performance and Security. *Maintainability* can be considered the primary concern of refactoring, as the main goal of refactoring transformations is the improvement of code maintainability. As for the other NFRs, they have some level of correlation to refactoring processes. *Robustness* can be improved by reorganizing modules in order to integrate patterns geared at error handling (e.g., Chain of Responsibility [25]). *Performance* can be improved through refactoring out code redundancies, as well as fixing sub-optimal distributions of code entities.

The methodology we adopted is an adaptation of the original work's methodology, with the addition of four new projects, as well as the adoption of more strict methodology processes, in order to improve the potential statistical accuracy of this work. Thus, Figure 4.1 summarizes the methodology we adopted to answer the research questions. First, we selected a set of eight

projects, a combination of the four projects from the original work, as well as an additional four, chosen based on the criteria described in Section 4.3.2. With these projects, we then collected data regarding internal quality attributes, refactorings and developer discussions from all eight projects. This process is further described in Section 4.3.3. Finally, we then performed an analysis of the resulting data set, in order to group the individual transformations into composite refactorings (see Section 4.3.4), as well as the collection of data about refactoring effectiveness. Then, we classified the developer discussions connected to each commit based on the presence of SARs and NFCs. This resulting data set was then used in order to answer the proposed RQs.

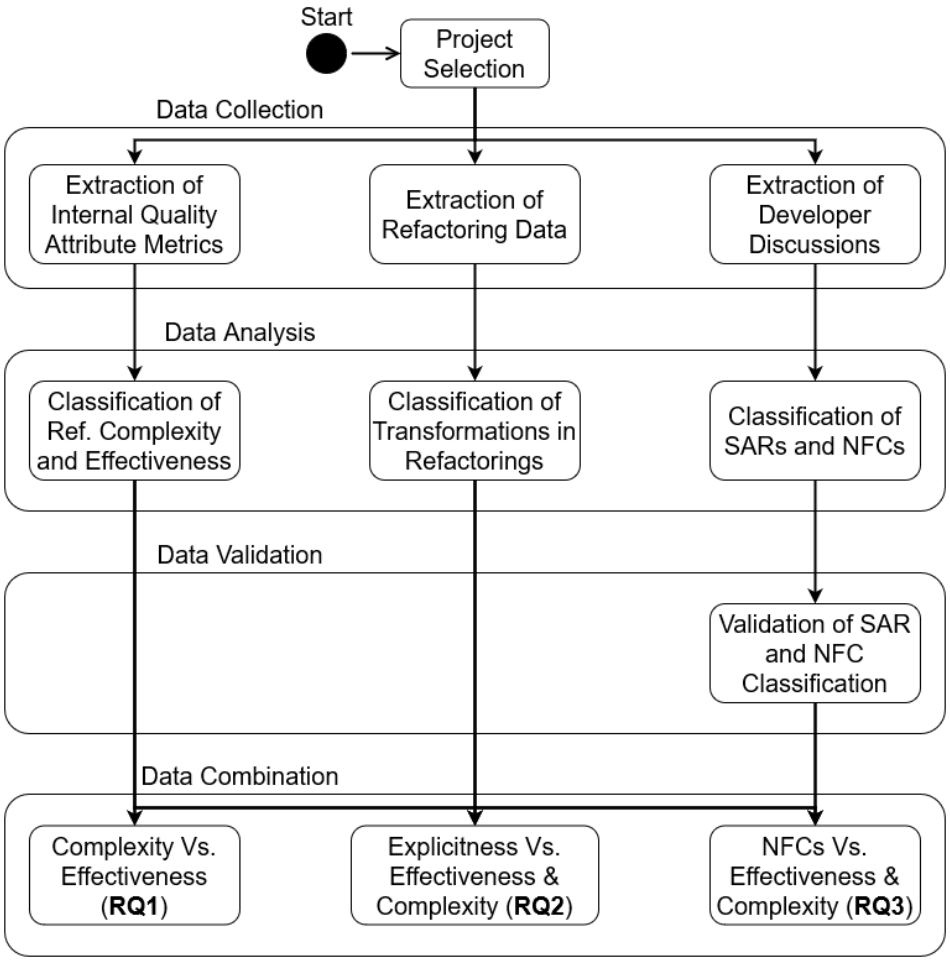

Figure 4.1: Adopted methodology.

4.3.2 Project Selection

We followed similar criteria to the original paper for project selection in this work. We kept the 4 projects from the previous work: Couchbase Java Client, Dubbo, OKHttp and JGit. However, we also added a new set of 4 projects: Fresco, RxJava, Presto and Netty. These four new projects bring new

points of view, as they are from different developers from the original four, have different ways of writing commit messages, and have different goals compared to the original four systems. The choice of doubling the amount of projects was made to balance both the time cost of analyzing each individual project, while still bringing a high amount of new data. These projects were chosen using the following criteria:

Java Open Source Projects. First of all, the project must be both open-source, and have Java as its primary language. The usage of open-source projects improves the work's replicability, while the usage of Java projects means we have access to a variety of support from outside tools, especially for analyzing structural quality.

Number of Contributors and Activity. For this criteria, it means that the project must both still be active at the time of analysis, as well as having a considerable number of contributors. This means that the project can more accurately represent the state of the industry throughout the years, including current times. Also, with the projects having a significant amount of contributors, it allows us to collect and analyze a larger set of developer discussions, essential for SAR and NFC detection.

Project Age. For this criteria, the project must be at least 5 years old a the time of the analysis. With this, we can more clearly see the potential architecture degradation that occurred over the project's lifespan, which is more common in older projects, which then allows us to see the developers' actions to mitigate such degradation.

Variety of Refactoring Types. For this criteria, the project must have undergone the application of a large variety of different refactoring transformations (as defined by Fowler et al [23]) used throughout its lifespan. With this, we can more clearly analyze the complexity of these refactoring actions, which uses different transformation types as its main metric.

4.3.3 Data Collection

With the intent to answer the previously-stated RQs, we performed data collection, extracting data from three different groups: (i) *Structural Quality Data*, composed of the internal quality attributes for each element of the project, required for determining refactoring effectiveness; (ii) *Refactoring Data*, composed of which refactoring transformations were applied in each commit, required for determining the presence of refactoring, and refactoring complexity, and; (iii) *Developer Discussions*, composed of both commit messages and issues/pull requests, which are required for the classification of SARs

and NFCs. With this data, we are then able to perform a correlation analysis in order to determine the potential relationships between them, thus confirming (or refuting) the ones described by Soares et al [47] in the original paper. This collection process is described in more detail as follows:

Structural Quality Data. As previously stated, structural quality data is comprised of the internal quality attributes for each element of the project. Thus, we first collected metrics related to those internal quality attributes, in order to quantify the project quality. For this, we used the Understand tool [54]. It is a static code analysis tool, which collects metrics related to the internal quality attributes from each element in each commit of each project. Thus, we can then analyze the metrics in each pair of subsequent commits in order to determine the changes in quality caused by each commit – which then allows us to classify the changes as improving, or worsening, the state of each attribute. Similarly to the original work, however, we have collected only a subset of Understand's metrics, in order to balance out the metric amount in each attribute, comprised of a total of 15 metrics, listed in the work's companion website [49].

Refactoring Data. As previously stated, refactoring data is comprised of a listing of which refactoring transformations were applied in each commit. Thus, in order to detect, and classify these refactorings based on the transformations proposed by Fowler et al [23], we used the RefactoringMiner 2.0 tool [53], an updated version of the RefactoringMiner tool used in the original work. This new version of the tool collects a large variety of refactoring information, including 40 different refactoring types, up from the original version's 15, from each commit a Java project. Alongside this, it also has a high reported precision of 99%, with a recall of 94% [53]. Thus, this makes RefactoringMiner a very reliable tool for refactoring detection and classification.

Developer Discussions. Developer discussion data is comprised of discussions logged in the projects' repositories, during their evolution. Thus, we used the GitHub API to extract the following items from the repositories of the selected projects: (i) Commit Messages; (ii) Issues and Pull Requests. Similarly to the original project, Couchbase Java Client and JGit did not have Issue and Pull Request data, and thus were only comprised of Commit Messages. However, as described by Soares et al [47], SAR detection is still accurate even with only this information.

4.3.4 Data Analysis

Once the data collection process was finished, we began the data analysis process. Due to space-related concerns, some additional information about this process can be found in the work's companion website [49], though all of the main steps and results are described in this paper. The analysis process had at its goal combining the data related to *refactoring effectiveness* with that related to *refactoring complexity*, *refactoring explicitness*, and *non-functional concerns*. This, in turn, would allow us to answer our proposed **RQ**s (Section 4.3.1). This analysis was performed as follows:

Refactoring Complexity and Effectiveness. Through structural quality and refactoring data, we were able to determine both *refactoring complexity* and *refactoring effectiveness*. The original work by Soares et al [47] already described that both *the number of individual transformations*, as well as *the number of affected code elements* in a refactoring have no relation to refactoring effectiveness. However, one aspect of refactoring complexity that does is *the number of unique transformation types used in each refactoring*. Thus, this was the chosen metric for *refactoring complexity*. In total, we were able to extract 40 unique transformation types with RefactoringMiner 2.0, at both method and class level – such as *Extract Superclass*, *Inline Method*, and *Move Attribute*. The full list of supported refactoring types is in the work's companion website [49].

For *refactoring effectiveness*, we analyzed the metrics of each internal quality attribute – (code) complexity, cohesion, coupling and size –, and how they changed from one commit to another, when a refactoring was applied. In order to determine each change as *positive*, *negative* or *neutral*, we first determined which elements were affected by the refactorings in a specific commit. Then, we determined which changes occurred to the metrics in the internal quality attributes of these elements between the previous and the current commit. Finally, we used the following criteria: (i) if at least one of the metrics related to a specific attribute changed positively between the two versions, we classified the change as *positive*; (ii) if no positive changes occurred, and at least one of the metrics changed negatively, we classified the change as *negative*; (iii) finally, if none of the previous two conditions were fulfilled, we classified the change as *neutral*. This mirrors the classification performed in the original work [47]. Thus, by combining refactoring effectiveness and complexity data, we are able to better understand whether and how they relate to each other – answering **RQ1**.

Grouping of Transformations in Refactorings. In order to correctly determine refactoring complexity, we needed to group individual transformations

into single, or composite, refactorings. To do so, we utilized the *commit-based* heuristic, proposed by Sousa et. al. [50]. This means that all transformations applied in the same commit are considered as part of the same composite refactoring. The rationale is that these within-commit transformations are cohesively contributing to the same task. Thus, they should be analyzed as composing a well-cohesive refactoring. Finally, the remaining refactorings were classified as single refactorings. Thus, this allows us to keep the commit as our main unit of time, allowing us to more easily determine the relationship between SARs and NFCs (which are directly correlated to a single commit) and the refactoring actions themselves. With this, we strike a balance in how many transformations we can combine into composite refactorings, and how much we can ensure that the SARs and NFCs are directly correlated to the changes being analyzed. Thus, by doing so, we can combine this data with the internal quality attribute metrics in order to answer **RQ1**.

Presence of Self-Affirmed Refactorings. For the detection and classification of SARs, we used the same approach as the one defined by Soares et al [47], by creating an automatic, keyword-based classifier that matches a set of 11 keywords and 8 key-phrases with the developer discussions in order to determine if refactoring was discussed or not. The original keyword set they developed was based on Ratzinger's work [42], with the keyword set changed in order to improve its accuracy for the projects they were analyzing. Initially, we considered using a state-of-the-art machine learning-based classifier [4], but through our validation process (described in more detail in Section 4.4), we found out that the original keyword-based classifier, even without changing the keywords, had a higher accuracy in SAR detection with our new set of 4 projects. Thus, by combining this data with the previously collected complexity and effectiveness data, we are able to answer **RQ2**.

Non-Functional Concerns. For NFC detection and classification, the original work already described the very low accuracy of NFC detectors [47]. Thus, we skipped this step in this work, and performed the NFC detection and classification manually (this process is described in more detail in Section 4.4). By combining the NFC data set from the original work with our new set, we were able to have a manually-validated set of 775 commits. Thus, by using this manually-validated set, we can answer **RQ3**.

4.4 Validation

With the goal of (i) determining the trustability on the automatic SAR classification, and (ii) forming a manually classified data set of NFCs, we

performed a manual validation with a subset of commits from the 4 new projects. The reason we focused on performing a validation on only the four new projects is due to the original work already releasing their own validated data set, thus allowing us to combine both sets to perform further verifications.

4.4.1 The Validation Process

For this new manual validation, we decided to perform both the SAR and NFC validation in the same set of commits. Thus, we selected a set of 56 commits from each project, forming a complete set of 224 different commits. This set was divided equally into four different groups, based on the results from the automatic SAR detection, and RefactoringMiner's refactorings: (i) commits classified as SAR, and classified as refactorings by RMiner; (ii) commits classified as SAR, but not classified as refactorings by RMiner; (iii) commits not classified as SAR, but classified as refactorings by RMiner, and; (iv) commits not classified as SAR, and also not classified as refactorings by RMiner.

By equally splitting the data set between the four projects, we can mitigate potentially biased results from focusing too much on a single project. And by splitting the results in those four categories, we can be able to detect interesting results, such as possible non-standard refactorings (classified as SAR, but not detected by RMiner), as well as have initial ideas about the frequency of actual SARs vs. refactorings in which no SAR was present.

This validation was performed with 7 participants – all 7 being knowledgeable in both the context of refactoring and NFRs. Alongside this, we also gave them a document with clear definitions of refactorings, self-affirmed refactorings, and NFRs/NFCs. This validation was done in two steps: first, each participant would validate a set of 32 commits, equally split between the four projects. Once all validations in the first step were completed, the sets were reshuffled, ensuring no participant had a repeating commit, and a second round of validations were performed. Finally, with both validations complete, 2 authors from this work then determined, for each conflicting classification, based on their own experience, and the confidence of the answers by the participants, which classification would be kept. The participants were asked to identify the following: (i) if the commit contained a SAR in any of its discussions; (ii) which sentence, and in which location (commit message, issue, etc.) was the SAR located; (iii) which keywords in the sentence could be used to detect the SAR; (iv) if any NFC was present, and which sentences contained potential NFCs; (v) if a maintainability-related NFC was present; (vi) if a robustness-related

NFC was present; (vii) if a performance-related NFC was present, and; (viii) if a security-related NFC was present. Finally, for (i), (v), (vi), (vii), and (viii), the participants were asked to determine, in a scale of 1 to 5, which confidence level they had in their classification. The full results of this validation are available in this work's companion website [49].

4.4.2 Self-Affirmed Refactoring Validation

With the results of this validation, we are then able to quantify the precision and the recall of the SAR detector. Using only this new classification, the resulting precision was of 69.7%, while the recall was of 80%, thus leading to an F1-Score of 74.5%. As previously mentioned, however, we considered using the state-of-the-art machine learning-based classifier proposed by AlOmar et al [4]. Thus, we used the Azure service the authors made available in the paper's website [1] to classify this validated dataset, in order to determine its accuracy for these four projects. However, this classifier was only able to reach a precision of 52.3%, with a recall of 80.3%, thus leading to an F1-Score of 63.3%. With these results, we could see that, even without changing the keyword set, the keyword-based classifier still had a higher accuracy overall, so we decided to use the same classifier as the original work. Finally, by considering both the original work's validated set, and the newly-validated set, the keyword-based classifier reaches a precision of 74.9%, and a recall of 81.4%, thus leading to an F1-Score of 78%.

4.4.3 Non-Functional Concern Validation

By using the same validation methodology as previously described in Section 4.4.1, we are also able to form a manually-validated set of commits, based on the presence of NFCs. Due to the original work's classifier being unable to reach a high accuracy [47], we decided to completely abandon the automatic classification, and simply validate them manually in order to have a data set to perform NFC analyses on. Thus, by combining this new classified set with the original validated set released by Soares et al on their project's website [47], we were able to have the set of 775 classified commits, as described in Section 4.3. From this group, 407 mentioned NFRs in their discussions, while 367 did not. Considering each individual NFR, we had 197 refactorings related to *Maintainability*, 126 related to *Robustness*, 86 related to *Performance*, and 40 related to *Security*, including refactorings that can be classified as multiple types.

[1] This was performed on July of 2021.

4.5 Results and Discussions

In this section, we present the results of our analysis on the data, which answers the **RQ**s introduced in Section 4.3.1. First, we perform an analysis on the potential relationship between refactoring complexity and effectiveness (**RQ1**). Then, we perform an analysis on the results of **RQ1** with regards to the presence of SARs (**RQ2**). Finally, we then correlate the findings of **RQ1** with the presence of NFCs (**RQ3**). While we describe the results in this paper, the numerical results for each of the described analyses is available on the project's website [49].

4.5.1 Refactoring Complexity vs. Effectiveness

As our first goal, we focused on understanding how the different levels of refactoring complexity interacts with refactoring effectiveness, when applied to each internal quality attribute of the refactored code. We kept the original work's 5 categories, which divided commits into complexity levels based on the number of different transformation types in a refactoring. Thus, categories 1 to 4 contained refactorings composed of 1 to 4 transformation types, while category 5+ contained those with 5 or more transformation types, as once the number goes above 5, it tends to vary a lot – thus making it better to group them, and consider that group as the most complex kinds of refactorings. The results of this analysis are in Figure 4.2.

In Figure 4.2, it is possible to see that as complexity increases, so does the impact of refactorings – similarly to one of the findings of the previous work. This is signified by the reduction in neutral effects, thus making it more possible to have a non-neutral effect, or, in other words, an impactful effect. Considering all projects, the average percentage of neutral effects reduced from 28% (at complexity 1) to 5.5% (at complexity 5+) – a very large reduction. While not as intense as the neutral effects, there was also a clear reduction in the frequency of negative effects – the average began at 34%, and reduced to 11.6%. This differs from the original work of Soares et al [47], as the increase in negative effects on high-complexity refactorings reported in their work did not occur. By using a the Kruskal-Wallis test, we could see that the statistical significance for the results found in (code) Complexity and Coupling are quite close to significant (p-values < 0.06). However, the results for Cohesion, and especially Size did not achieve the same statistical significance (p-values of 0.10 and 0.19, respectively)

In fact, in these results, it is possible to see that the positive effects

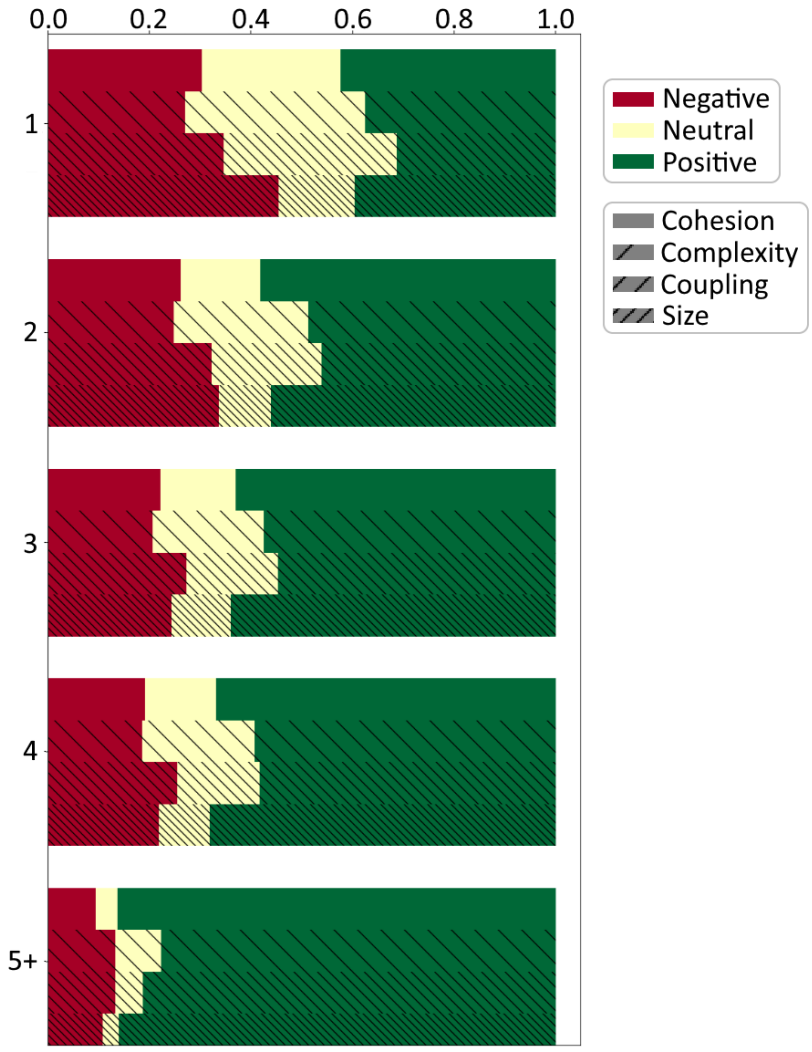

Figure 4.2: Distribution (decimal percentage) of effects based on the refactoring complexity. Each 0.1 on the horizontal scale represents 10% in change frequency of the corresponding effectiveness.

increased dramatically, while the non-positive effects drastically reduced in frequency as the complexity increased. This means that high-complexity refactorings, i.e., those that combine a high variety of different transformation types into a single, large refactoring, can have significantly better effects on the code. Thus, this means that the increased effort of complex refactorings might be a worthy endeavor, as the potential for positive results is clearly seen. Alongside this, the inherent effort of applying complex refactorings can be reduced by improvements in automated tool support, which can allow for a much easier application of complex refactorings, spanning a variety of possible transformations.

Finding 1: **(RQ1)** As refactoring complexity increases, the frequency of positive effects on the code increase with it – thus reducing the possibility of refactorings causing non-positive effects.

Implications. The results of this **RQ** are interesting, as it confirms the previous study's findings about refactoring complexity, when defined as the number of different transformation types that compose a refactoring, is related to refactoring effectiveness. As such, this can be an interesting point for future studies – an understanding of to what extent specific combinations can influence refactoring effectiveness. For developers, the results of this work can benefit from the understanding that, even though the effort is higher, the application of complex refactorings is worth pursuing, with a high potential for frequent positive changes.

4.5.2 SARs vs. Complexity and Effectiveness

Thus, by taking the information related to refactoring explicitness (i.e., the presence of SARs), we are able to combine them with the results of **RQ1** in order to achieve new findings. For this end, Figure 4.3 presents the difference in frequency of refactorings with different complexity levels between the refactoring set where SARs were not present, and the refactoring set in which SARs were present, respectively. By performing this comparison, we can understand if explicit refactoring-related concerns during the refactoring process can actually be related to the effectiveness of the applied refactoring. Comparing the two, it is possible to see that, between the non-SAR and the SAR sets, respectively, the proportion of single refactorings decreased (46.8% to 32%), while the proportion of complex changes (5+ transformation types) increased (14.3% to 26.6%). By using the Wilcoxon Rank-Sum Test to determine the statistical significance of the results, we can see a resulting p-value of 0.008, thus showing that these results are statistically significant (p-value < 0.05).

With this, our findings confirm the ones from the original paper – explicit refactorings are more frequently complex than their non-explicit counterparts. However, there is a difference between the two works, as in this result, the more complex refactorings (5+) were already much more common even in the non-SAR data set, which might have been caused by the increase in the amount of transformation types from Refactoring Miner 2.0. With these results, we could then say that, when the developers are concerned with refactoring, to the point of expressing and documenting their concern explicitly, represented

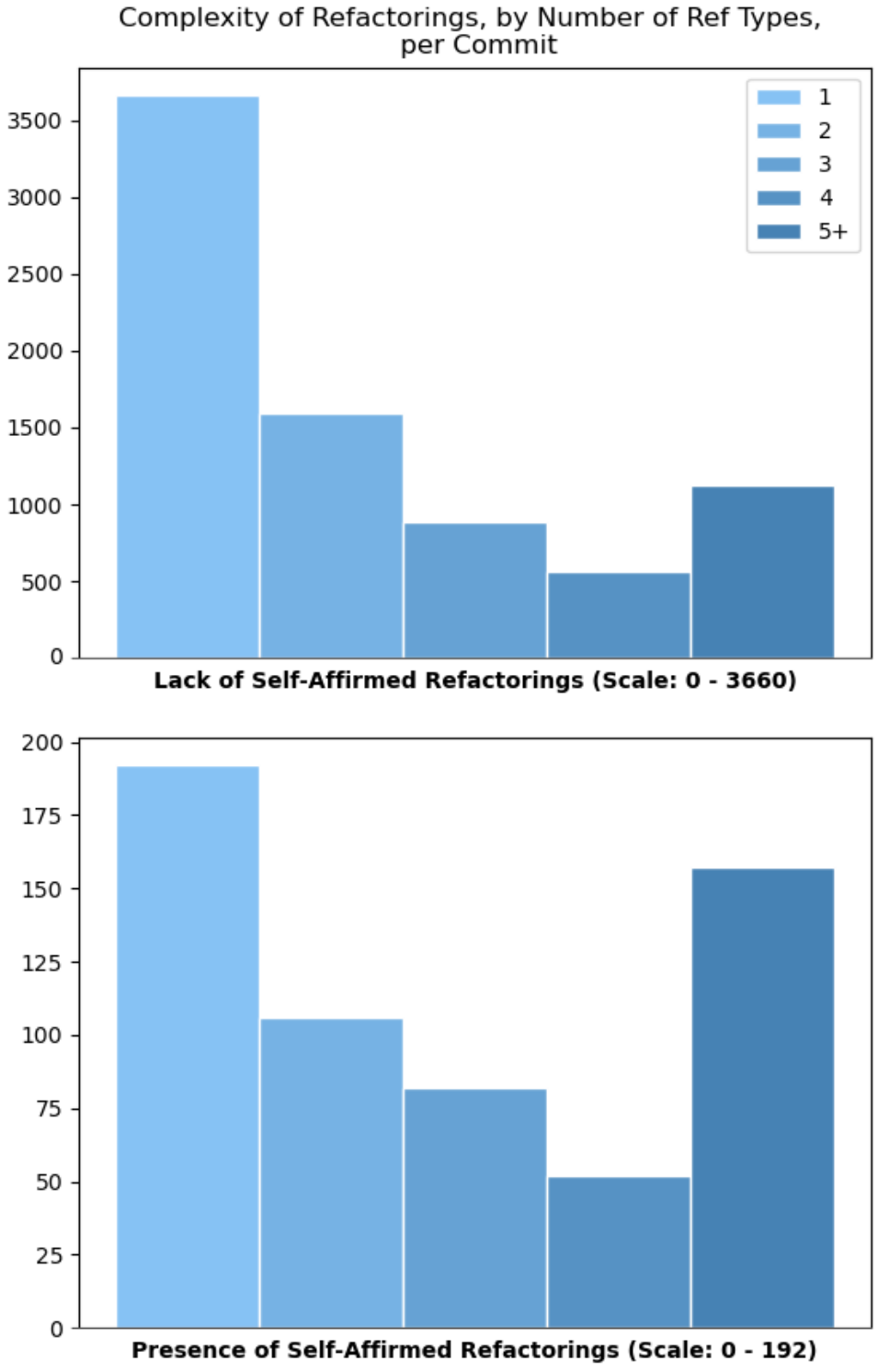

Figure 4.3: The frequency of self-affirmed and non self-affirmed refactorings composed of 1, 2, 3, 4, or 5+ refactorings.

through a self-affirmation of their refactoring [2] – they tend to perform more complex refactorings, spanning a variety of different refactoring types.

Finding 2: **(RQ2)** When developers manifest an explicit concern with refactorings, they also tend to perform refactorings containing a higher variety of transformation types.

By then analyzing the potential relationship between the presence of SARs and refactoring effectiveness, we reached the results presented in Figure 4.4. Keeping the same definition of complexity between the four internal quality attributes, we can see the following results: Between non-SARs and SARs, respectively, the percentage of neutral changes did not change much, in average (20.4% to 20.3%), while the percentage of negative effects visibly reduced (28.2% to 19.1%). Comparing these results with the ones described by Soares et al [47], we can see that the conclusions clearly differ. In Soares et al's work, they described SARs as having usually more detrimental changes, increasing the frequency of negative effects. On the other hand, in our work, we show SARs as having a definite increase in positive changes, mainly reducing the frequency of negative effects. By performing the Wilcoxon Rank-Sum Test on these results in order to test their statistical significance, all internal quality attributes reached the same p-values of 0.1.

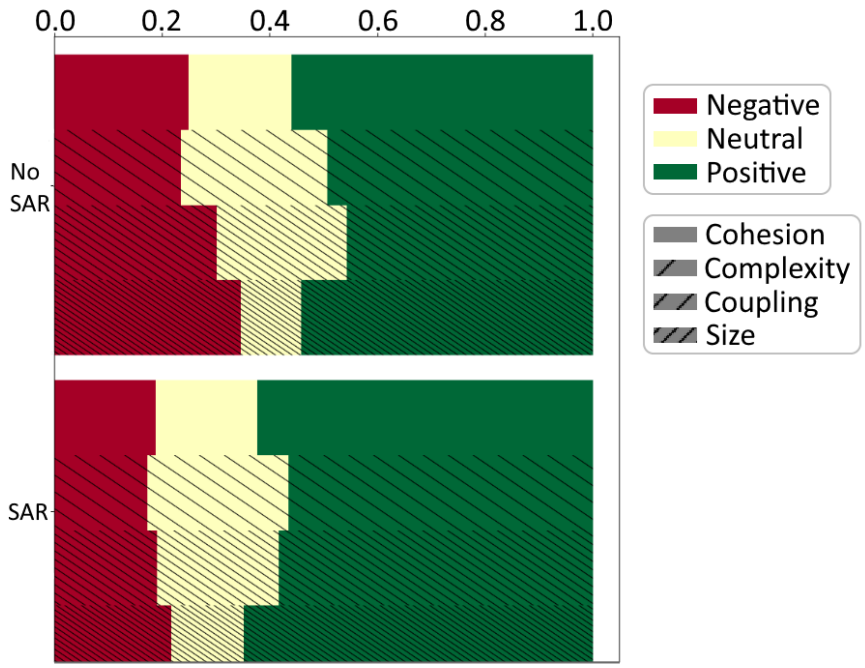

Figure 4.4: The negative, neutral and positive effect of self-affirmed and non self-affirmed refactorings. Each 0.1 on the horizontal scale represents 10% in change frequency of the corresponding effectiveness.

This result is very different from the one found by the work by Soares et al [47]. The original work described SARs as having a much more frequently

negative effect on the code in comparison to their non-SAR counterparts. Our work directly contradicts this finding, as the frequency of negative effects was the only aspect which was visibly reduced between the non-SAR and SAR data sets. Thus, our results show that when developers explicitly show their concern with refactorings, they tend to make refactorings with less negative effects on the code: the possibility of neutral effects stays the same, but the possibility of positive effects is larger. This might mean that developers do perform more effective refactorings when their primary concern is the refactoring process itself, which is an effect that can be even more intensified by a proper aid of tool support for the application of complex refactorings (which are more common in SARs), as other works describe the current state of refactoring support tools as lacking [30, 55, 51]. Thus, we can summarize the aforementioned results as the following finding:

> **Finding 3**: **(RQ2)** When a developer explicitly describes their concern with refactorings through self-affirmed refactorings, they also more frequently perform refactorings with less negative effects on the code, compared to their non-SAR counterparts.

Implications. These results confirm the previous work's findings of developers performing more complex refactorings when they make their concern explicit along the change process. Thus, this might mean that developers write about their refactoring processes in commit messages more often when the refactorings reach higher levels of complexity. However, our results also contradict the previous work's findings regarding refactoring explicitness's effects on refactoring effectiveness – as we found that refactorings in which developers explicitly talk about the refactoring process have usually less negative effects on the code.

These new results make sense, considering the increase in refactoring complexity, and the effects of complexity on refactoring effectiveness. Thus, this means that developer concerns can, in fact, influence the effectiveness of the refactorings they apply. As such, this means that correlating developer concerns and refactoring usage can be an interesting research direction for future studies. For practitioners, our findings imply that a good documentation process is important for refactorings, potentially letting collaborators uncover and fix errors in the process before the proponent commit. Undocumented refactorings can remain unnoticed, and negative effects can easily find their way into the code base.

Alongside this, in the previous work, it was reported that around 30% of the refactorings classified as SARs by the manual validations were not detected as refactorings by RefactoringMiner. This leads the authors to believe that those SARs included applications of refactorings that were not supported by RefactoringMiner. In this work, with the usage of RefactoringMiner 2.0, the percentage of SARs (as classified by the manual validations) that were not detected as refactorings by RefactoringMiner was only 6%. This means that RefactoringMiner's updates did cause it to detect a larger, more realistic variety of refactorings that appeared in SARs and were not previously collected. However, the remaining 6% SARs undetected by RefactoringMiner could show that developers may apply refactorings that do not necessarily fit the refactoring types described by Fowler et al [23] and, thus, are undetected by RefactoringMiner.

4.5.3 NFCs vs. Complexity and Effectiveness

Finally, we analyze whether and how concerns with each of the four NFRs (maintainability, robustness, performance and security) can affect both the complexity and the efficiency of the refactorings applied in their associated commits. Due to there not being an automated solution for NFR classification, we have done this process manually (as detailed in Section 4.4), having a total set of 407 validated commits in which NFRs were mentioned, and another set of 367 in which no NFRs were mentioned. Thus, Figure 4.5 shows the frequency of refactorings in each complexity category (1, 2, 3, 4 or 5+), comparing those in which NFRs were present with the set in which no NFRs were mentioned.

Once again, we can see that, very similarly to what occurred between SARs and non-SARs, complex refactorings are more common when developers are explicitly expressing their NFCs. Unfortunately, similarly to the original work, we were unable to achieve complete statistical significance for the results of this **RQ**, even with the increased data set. By applying the Wilcoxon Rank-Sum Test between the NFR and non-NFR distributions, the resulting p-value was 0.91, a resulting value far from statistical significance. The previously-described results can be summarized into the following finding:

> **Finding 4**: **(RQ3)** When explicitly showing concerns with NFRs, developers perform more complex refactorings with a higher frequency.

Once this was done, we then analyzed the potential relationship between NFCs and refactoring effectiveness. To do so, we analyzed the difference

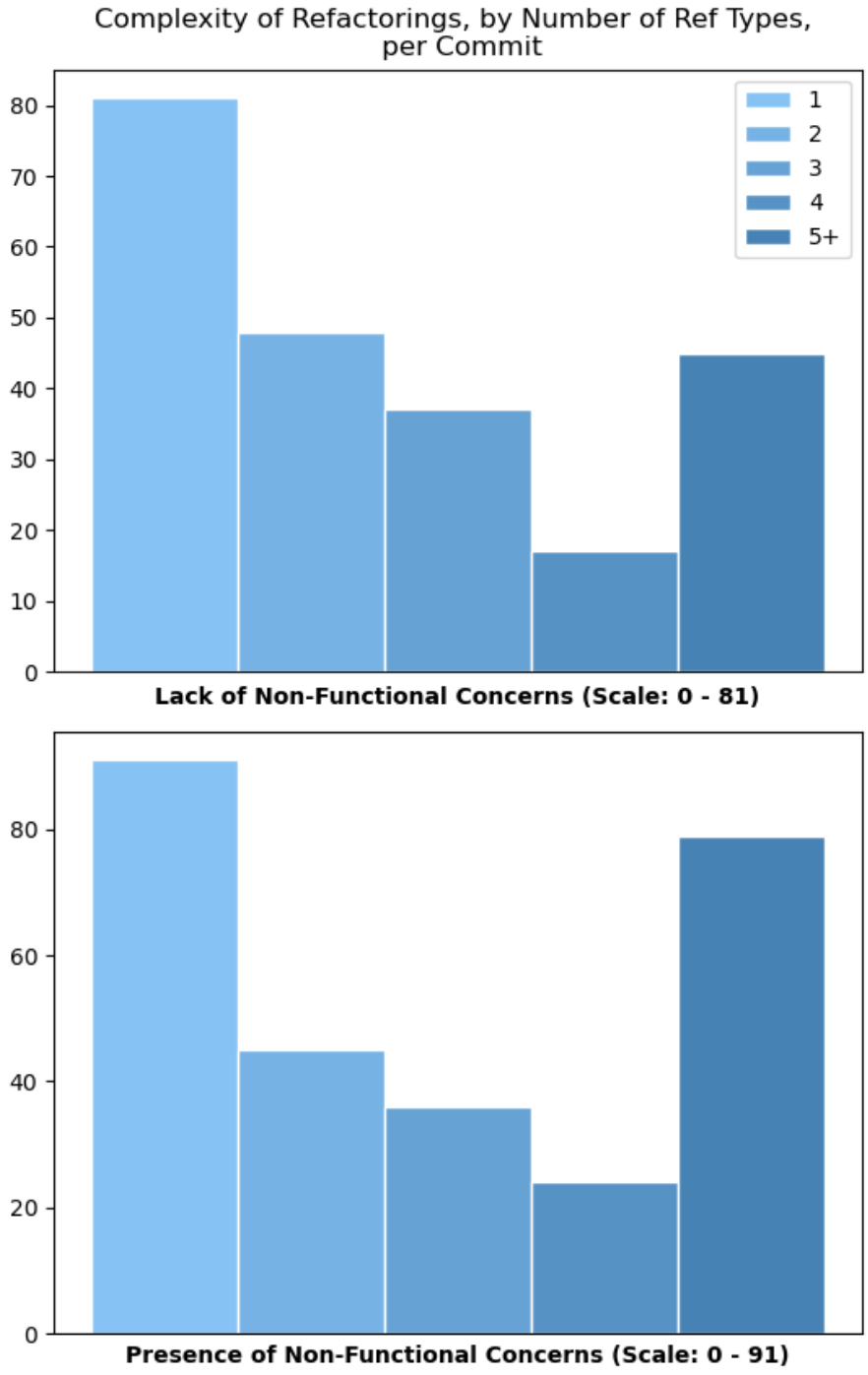

Figure 4.5: The frequency of refactorings composed of 1, 2, 3, 4, or 5 or more refactorings grouped by the presence of NFCs.

between NFR-related changes for each one of the four NFRs. Finally, we analyzed the impact of refactorings when coupled with NFCs. First, by analyzing the difference between NFR-related changes for each of the 4 NFRs, we saw that the increase in the negative effects of refactorings was significant – with no apparent increase in its positive effects. However, positive changes were still more frequent than negative changes. This correlates to the findings

described in Section 4.5.2 – in which developer concerns reduce the neutral effects of refactorings, but also increase the possibility for negative effects. We also focused on the analysis of how mentions to each individual NFR impacts on the effectiveness of refactorings. Figure 4.6 provides the results of this analysis, by showing how each concern relates to affecting the internal quality attributes. The attributes were analyzed individually by grouping their related metrics.

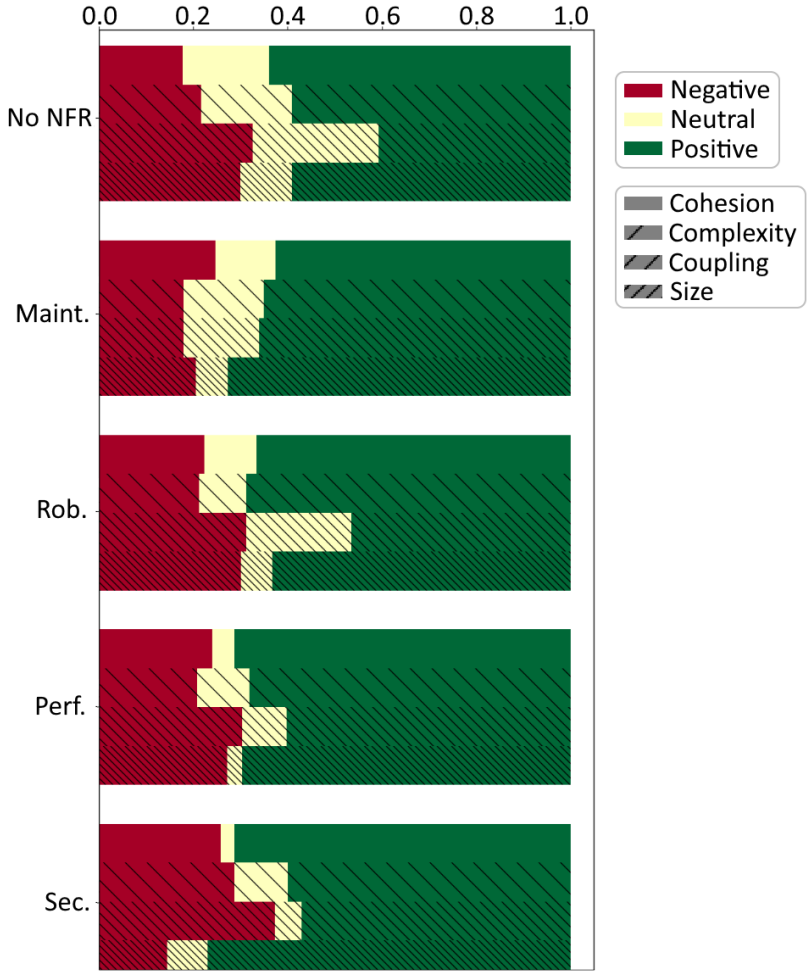

Figure 4.6: The negative, neutral and positive effects of refactorings when coupled with changes in NFRs (considering only the validated data set). Each 0.1 on the horizontal scale represents 10% in change frequency of the corresponding effectiveness.

Considering only the validated data set, we can see that refactorings in which developers were concerned with NFRs tended to have a slight increase in frequency of positive effects, when compared to the non-NFR set. Of the NFR groups, refactorings with a focus on Maintainability, Performance and Security tended to have more positive effects (averaging on around 66% of refactorings being positive), while Robustness had a lower average (61%), though it was still more than those without NFRs (56%). Individually analyzing each internal quality attribute, Maintainability and Performance seemed to have more

balanced effects (ranging from 60-70% of the refactorings being positive), while Robustness and Security seemed to have more skewed effects (ranging from 45 to 80% of the refactorings being positive).

Thus, this partially contradicts the findings of the original work, which stated that Maintainability and Robustness had balanced effects, while Performance and Security were skewed towards specific internal quality attributes. In these results, while Maintainability kept its balanced results, refactorings applied when developers were concerned with Robustness tended to lack improvements to Coupling. On the other hand, Security kept its skewed results, though this time with most refactorings causing improvements to Size, while refactorings applied when developers were concerned with Performance had balanced results, similar to those of Maintainability. Similarly to the original work, we were also unable to find complete statistical significance for the results of this finding. However, in this case, the correlations between Coupling and the NFR Types, with the usage of the Kruskall-Wallis test, reached a relatively low p-value of 0.07. Meanwhile, Cohesion, (code) Complexity and Size reached p-values of 0.24, 0.16 and 0.33, respectively. The previously-described results can be summarized in the following finding:

> **Finding 5**: **(RQ3)** Refactorings applied when developers are concerned with NFRs have more positive effects than those in which they do not show such explicit concern. When developers are concerned with Maintainability or Performance, they tend to make more balanced refactorings in terms of the internal quality attributes of the code. Conversely, when they apply refactorings while concerned with Robustness or Security, they tend to have skewed results, with regards to the internal quality attributes.

Implications. We first observed that refactorings applied when developers were concerned with NFRs had similar, though slightly different effects than those applied when developers were concerned with refactorings. While both NFRs and SARs led to more positive effects in refactorings, SARs had less negative effects overall, while keeping the frequency of neutral effects. Conversely, NFRs other than Maintainability led to less neutral effects overall, while keeping the frequency of negative effects, while Maintainability led to effects similar to those observed in the SARs, as expected. This might mean that mixing different concerns into a single change may be risky, as it keeps a non-negligible chance of the refactorings having a negative effect overall – which is mitigated when developers are mainly concerned with Maintainability.

4.6 Threats to Validity

Although we performed a variety of actions to attempt to mitigate potential threats to the validity of this work to the best of our ability, some threats still remain, described as follows:

Generalizability. While we now selected a total group of 8 projects, from a variety of different fields and from different developers, the results we found might still not be able to be generalized for other contexts, especially those related to closed-source projects.

Accuracy of Refactoring Detection. RefactoringMiner, while having a high reported accuracy [53], did not have its accuracy evaluated for our specific data set. However, other works have used RefactoringMiner for similar sets of projects [9], while still having a good level of accuracy in refactoring detection.

Accuracy of Self-Affirmed Refactoring Classification. The keyword-based classifier we used for SAR classification in this work may not be generalizable to other projects, due to the keyword set potentially needing to change. However, to mitigate this, we have performed an evaluation of the classifier's accuracy with regards to a manually validated sample of the data set, leading to a decently high level of accuracy, and still surpassing the accuracy of another, state-of-the-art classifier in this specific data set.

Lack of NFR-related Data. Due to automatic classification of NFR discussions still not being possible with a reasonable accuracy, we had to perform our NFR-related analyses with a manually validated sample of the data set, containing only 770 out of the 8,408 refactorings analyzed in the other **RQ**s. Thus, this means that the results of that analysis might not be generalizable as well.

4.7 Final Remarks

This work had at its goal confirming whether and how the relationships between four factors manifest, these being: (i) refactoring complexity (represented by the amount of different transformation types in a single refactoring); (ii) refactoring effectiveness (represented by the improvement in internal quality attributes); (iii) refactoring explicitness (represented by the presence of SARs), and; (iv) the presence of NFCs during the refactoring process (represented by the presence of NFR-related discussions). We performed a quantitative analysis of 8,408 refactorings, from eight different open-source projects. With our results, we can show that developers tend to apply more complex

refactorings when they are explicitly concerned with either the refactoring process itself, or with other NFRs. We also observed that the application of these complex refactorings can be an endeavor worth pursuing, as they tended to have more frequently positive effects on the refactored code.

These findings show that these complex refactorings can be a powerful tool in the developers' hands for code maintainability, and thus may require more support from automated tools. Recent studies, such as the one proposed by Tenorio et al [51], describe automated refactoring tools as lacking in providing support for customized, complex refactorings – supporting only simple, standardized transformations. Thus, a proper support for these complex transformations may lead developers to perform them with less effort, making them a more enticing practice, and thus potentially leading to more positive code maintenance results overall. Thus, this work can motivate tool developers into improving support for these complex refactorings, combining a variety of different transformation types into a single change. Alongside this, our results might also motivate researchers into considering factors such as refactoring complexity, refactoring explicitness, and the presence of NFCs as being potentially important in code quality improvement. This, thus, can lead researchers into, for example, analyzing potential techniques for performing more complex refactorings efficiently.

As future work, we plan on further analyzing how other developer-related factors can impact the refactoring process. Alongside this, we also plan to verify how refactoring patterns – i.e., a specific combination of transformations – can relate to their potential effectiveness. Finally, we aim at uncovering a potential way of using these developer-related factors (such as refactoring explicitness and NFCs) to determine the possible developers' goals during the refactoring process.

4.8 Summary

This chapter aimed at replicating the study performed in Chapter 3. As previously described in that Chapter, our goals with this paper are increasing the project data set – by doubling the project amount from 4 to 8 –, improving the validation process – by performing a stricter, multi-phase validation –, improving SAR detection – by comparing our approach with a state-of-the-art, machine learning-based approach –, and improving refactoring detection – through the usage of RefactoringMiner 2.0.

Our results both confirm and contradict those from the first study, in all three problems presented in Section 1.1. For Problem 1, i.e., the

lack of understanding of to what extent refactoring complexity correlates to their effectiveness, we contradict the finding of the first study regarding the relationship between complexity and effectiveness. In these results, we discovered that complex refactorings have more frequent positive effects when compared to their simpler counterparts, with a noticeable reduction in neutral effects. In the next chapter, we revisit the main contributions of this Masters' dissertation, and present new challenges and opportunities for improvement and future work, which have emerged along the studies performed throughout this Masters' dissertation.

For Problem 2, i.e., the lack of understanding to what extent explicit refactoring-related concerns relate to refactoring effectiveness, we confirm the results of the first study regarding the relation between refactoring explicitness and complexity – that, in general, developers explicit their concerns with refactorings through SARs more often when performing more complex refactorings. However, we contradict the first study's results regarding the relation between SARs and refactoring effectiveness, with refactorings accompanied by SARs having more frequent positive effects overall, with also a clear reduction in the frequency of negative effects, compared to their non-SAR counterparts.

Finally, for Problem 3, i.e, the lack of understanding on how NFCs affect refactoring usage and effectiveness, we confirm the results of the first study regarding the relation between NFCs and refactoring complexity – that developers explicit their concerns with NFCs more often when performing more complex refactorings. However, we contradict the first study's results regarding the relation between NFCs and refactoring effectiveness, with refactorings related to NFCs having, on average, more positive effects overall. Alongside this, we have also seen that developers perform more balanced refactorings when they are concerned with Maintainability and Performance, while having more skewed results when performing refactorings related to concerns with Robustness and Security – though not to the same extent as reported in the first study.

5
Final Conclusions

Refactoring is a process that tries to improve software quality and maintainability, to generally mixed results. While definitely capable of achieving positive changes, many refactorings tend to either not change the code in any significant manner, or cause the code's quality to decay even more than it previously was. Thus, uncovering the factors that can potentially increase the effectiveness of such changes is paramount for its usage in software engineering as a whole. Thus, the analysis of factors such as refactoring complexity, refactoring explicitness, and developer concerns with NFRs, can reveal interesting observations on how this refactoring effectiveness can vary depending on those conditions.

In this work, our goal was to understand how these factors correlate to each other, and to refactoring effectiveness, so we can then help direct future studies in which characteristics can lead to interesting results, and also direct developers in understanding how their actions can influence the effectiveness of the refactorings they perform. To do so, first, we quantified refactoring complexity through the number of different transformation types in a refactoring, and refactoring effectiveness through the variations in the metrics related to the four internal quality attributes. Then, we analyzed the potential relationships between this complexity and refactoring effectiveness, so we could then understand how either refactoring explicitness – through the presence of SARs – or non-functional concerns can affect these two factors.

5.1
Contributions and Future Work

In summary, the main contributions and their possible impact on collaborative software communities are described as follows.

Contribution 1: *A Dataset of Internal Software Quality, Refactoring Characteristics, and Developer Discussions.* In this work, we collected data from a total of 8 projects. For each project, we collected 15 metrics related to software quality, divided into the four internal quality attributes of Cohesion, Coupling, (code) Complexity and Size, for each commit. We collected the refactoring history of each project, describing which transformations were per-

formed in each commit of the project. Finally, we collected commit messages, issues and pull requests from these projects, and classified them based on the presence of SARs. We also have a data set containing a group of 775 commits, manually validated and classified based on the presence of SARs and NFCs on their related discussions.

Contribution 2: *A Keyword-Based Classifier of Self-Affirmed Refactorings.* By automating Ratzinger [42]'s proposed keyword-based SAR classification, while also adapting the keyword set to the 4 projects analyzed in the first study, we created an automated keyword-based classifier of SARs, with a respectable F1-Score of 78%. We also compared the accuracy of the classifier on only the new set of 4 projects, for which it was not adapted to, with the accuracy of a state-of-the-art machine learning-based classifier, showing that our F1-Score of 74.5% on this second data set surpassed the machine learning approach's F1-Score of 63.3%.

Contribution 3: *A Set of Initial Insights on Refactoring Complexity, Effectiveness, Explicitness, and Non-Functional Concerns.* With the first study, we were able to determine some preliminary results that were later confirmed, even if many of them were contradicted. The presence of both SARs and NFCs are related to an increase in refactoring complexity, and an increase in refactoring complexity leads to a reduction in the frequency of neutral effects. We also discovered that the complexity aspect of the number of different transformation types in a refactoring is the one most related to refactoring effectiveness. These findings show that not only the mechanical aspects of refactoring could be related to their effectiveness. Developer-related aspects, such as their concerns, could impact refactoring effectiveness as well.

Contribution 4: *An Analysis on the Relationship Between Refactoring Complexity, Effectiveness, Explicitness, and Non-Functional Concerns.* With the second study, we were able to uncover some interesting results related to the relationship between these various factors. We found that more complex refactorings lead to a higher frequency of positive effects – a trait that is also shared by refactorings performed when developers are explicitly concerned with the refactoring process. We found that refactoring effectiveness increases, even if slightly, when developers are concerned with NFRs. We also saw that concerns with some NFRs might have differing results when compared to others. Some refactorings with NFR-related concerns had their effectiveness balanced among all 4 internal quality attributes, while others have a higher disparity between the effectiveness in each of the 4 attributes.

Taking these contributions into account, we could potentially suggest future steps to be taken to improve this research. The first, and primary

potential future work, is the addition of new projects, and the increase in the size of the data set. Changes arose in the results after updating RefactoringMiner to RefactoringMiner 2.0, and changing the project count. However, without complete statistical accuracy, this result might change further as more projects and refactoring types are added into the data set. Secondly, we can analyze in further detail why the machine learning-based approach failed at detecting self-affirmed refactorings correctly, and where it can improve, in order to reach high margins of accuracy with its classifiers (>90%). Finally, we can also look more deeply into specific combinations, in order to understand which specific combinations of refactoring types, when coupled with specific concerns, leads to positive results. An understanding of these aspects can allow us to recommend specific refactorings depending on the developers' intent with their changes.

5.2 Implications

This dissertation provides several findings that lead to implications that can be useful to researchers, tool developers, and practitioners. These implications are described as follows.

The study and application of complex refactorings is a worthy endeavor. In both studies, we were able to see that refactoring complexity is very much related to refactoring effectiveness – thus meaning that researchers might be able to uncover potentially interesting results from exploring this characteristic of refactoring in greater depth. Alongside this, in both studies, we saw a greater potential for positive effects in complex refactorings, meaning practitioners could use this information when planning code maintenance – as a well-implemented application of complex refactorings can lead to much more frequent positive changes. Finally, the relevance of refactoring complexity revealed in this work can imply that tool developers should look into improving automated refactoring tools to allow for better support of complex, customized refactorings, as the widely-used tools currently fail to do so [51].

Developer concerns can influence refactoring. In both studies, we were able to see that both the presence of SARs and NFCs were correlated to refactorings of higher complexity, as well as differing levels of effectiveness between refactorings accompanied by SARs or NFCs, and those not accompanied by them. Thus, once more, this means this is a promising research direction for future studies, as a deeper analysis on how developer concerns, even outside of just quality-related concerns, can affect refactoring can lead to much more accurate and customizable refactoring recommendation systems in

the future. Alongside this, practitioners can use this information to determine the importance of actually documenting their concerns, as this can lead to them being more attentive to the changes they performed, and allow them to detect potentially negative results, fixing them before actually committing the changes.

Bibliography

[1] AL DALLAL, J.; ABDIN, A.. **Empirical evaluation of the impact of object-oriented code refactoring on quality attributes: A systematic literature review**. IEEE Transactions on Software Engineering, 44(1):44–69, 2017.

[2] ALOMAR, E. A.; MKAOUER, M. W.; OUNI, A. ; KESSENTINI, M.. **Do design metrics capture developers perception of quality? an empirical study on self-affirmed refactoring activities**. arXiv preprint arXiv:1907.04797, 2019.

[3] ALOMAR, E.; MKAOUER, M. W. ; OUNI, A.. **Can refactoring be self-affirmed? an exploratory study on how developers document their refactoring activities in commit messages**. In: 3RD IWOR, p. 51–58. IEEE, 2019.

[4] ALOMAR, E. A.; MKAOUER, M. W. ; OUNI, A.. **Toward the automatic classification of self-affirmed refactoring**. Journal of Systems and Software, 171:110821, 2021.

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

[6] BARBOSA, C.; UCHÔA, A.; COUTINHO, D.; FALCÃO, F.; BRITO, H.; AMARAL, G.; SOARES, V.; GARCIA, A.; FONSECA, B.; RIBEIRO, M. ; OTHERS. **Revealing the social aspects of design decay: A retrospective study of pull requests**. In: PROCEEDINGS OF THE 34TH BRAZILIAN SYMPOSIUM ON SOFTWARE ENGINEERING, p. 364–373, 2020.

[7] BAVOTA, G.; DE LUCIA, A.; DI PENTA, M.; OLIVETO, R. ; PALOMBA, F.. **An experimental investigation on the innate relationship between quality and refactoring**. Journal of Systems and Software, 107:1 – 14, 2015.

[8] BIBIANO, A. C.; FERNANDES, E.; OLIVEIRA, D.; GARCIA, A.; KALINOWSKI, M.; FONSECA, B.; OLIVEIRA, R.; OLIVEIRA, A. ; CEDRIM, D..

A quantitative study on characteristics and effect of batch refactoring on code smells. In: ESEM, p. 1–11. IEEE, 2019.

[9] BIBIANO, A. C.; SOARES, V.; COUTINHO, D.; FERNANDES, E.; CORREIA, J.; SANTOS, K.; OLIVEIRA, A.; GARCIA, A.; GHEYI, R.; FONSECA, B.; RIBEIRO, M.; BARBOSA, C. ; OLIVEIRA, D.. **How does incomplete composite refactoring affect internal quality attributes?** In: 28TH ICPC, 2020.

[10] BIBIANO, A. C.; ASSUNÇAO, W.; COUTINHO, D.; SANTOS, K.; SOARES, V.; GHEYI, R.; GARCIA, A.; FONSECA, B.; RIBEIRO, M.; OLIVEIRA, D. ; OTHERS. **Look ahead! revealing complete composite refactorings and their smelliness effects**.

[11] BOURQUIN, F.; KELLER, R. K.. **High-impact refactoring based on architecture violations**. In: 11TH EUROPEAN CONFERENCE ON SOFTWARE MAINTENANCE AND REENGINEERING (CSMR'07), p. 149–158. IEEE, 2007.

[12] CACHO, N.; CÉSAR, T.; FILIPE, T.; SOARES, E.; CASSIO, A.; SOUZA, R.; GARCIA, I.; BARBOSA, E. A. ; GARCIA, A.. **Trading robustness for maintainability: An empirical study of evolving c# programs**. In: 36TH ICSE, p. 584–595. ACM, 2014.

[13] CACHO, N.; BARBOSA, E. A.; ARAUJO, J.; PRANTO, F.; GARCIA, A.; CESAR, T.; SOARES, E.; CASSIO, A.; FILIPE, T. ; GARCIA, I.. **How does exception handling behavior evolve? an exploratory study in java and c# applications**. In: IEEE ICSME, 2014.

[14] CASAMAYOR, A.; GODOY, D. ; CAMPO, M.. **Identification of non-functional requirements in textual specifications: A semi-supervised learning approach**. Information and Software Technology, 52(4):436 – 445, 2010.

[15] CEDRIM, D.; SOUSA, L.; GARCIA, A. ; GHEYI, R.. **Does refactoring improve software structural quality? a longitudinal study of 25 projects**. In: PROCEEDINGS OF THE 30TH BRAZILIAN SYMPOSIUM ON SOFTWARE ENGINEERING, p. 73–82, 2016.

[16] CEDRIM, D.; GARCIA, A.; MONGIOVI, M.; GHEYI, R.; SOUSA, L.; DE MELLO, R.; FONSECA, B.; RIBEIRO, M. ; CHÁVEZ, A.. **Understanding the impact of refactoring on smells: A longitudinal study of 23 software projects**. In: PROCEEDINGS OF THE 2017 11TH JOINT

MEETING ON FOUNDATIONS OF SOFTWARE ENGINEERING, p. 465–475, 2017.

[17] CHÁVEZ, A.; FERREIRA, I.; FERNANDES, E.; CEDRIM, D. ; GARCIA, A.. **How does refactoring affect internal quality attributes? a multi-project study**. In: 31ST SBES, p. 74–83. ACM, 2017.

[18] CHOWDHURY, I.; CHAN, B. ; ZULKERNINE, M.. **Security metrics for source code structures**. In: 4TH INTERNATIONAL WORKSHOP ON SOFTWARE ENGINEERING FOR SECURE SYSTEMS, p. 57–64. ACM, 2008.

[19] DEMEYER, S.. **Maintainability versus performance: What's the effect of introducing polymorphism?** 2003.

[20] DI, Z.; LI, B.; LI, Z. ; LIANG, P.. **A preliminary investigation of self-admitted refactorings in open source software (s)**. In: PROCEEDINGS OF THE 30TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING AND KNOWLEDGE ENGINEERING. KSI Research Inc. and Knowledge Systems Institute Graduate School, 2018.

[21] FERNANDES, E.; CHÁVEZ, A.; GARCIA, A.; FERREIRA, I.; CEDRIM, D.; SOUSA, L. ; OIZUMI, W.. **Refactoring effect on internal quality attributes: What haven't they told you yet?** Information and Software Technology, 126:106347, 2020.

[22] FOKAEFS, M.; TSANTALIS, N. ; CHATZIGEORGIOU, A.. **Jdeodorant: Identification and removal of feature envy bad smells**. In: 2007 IEEE INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE, p. 519–520. IEEE, 2007.

[23] FOWLER, M.. **Refactoring: improving the design of existing code**. Addison-Wesley Professional, 2018.

[24] AN, G.; BLOT, A.; PETKE, J. ; YOO, S.. **Pyggi 2.0: Language independent genetic improvement framework**. In: 27TH JOINT MEETING ON EUROPEAN SOFTWARE ENGINEERING CONFERENCE AND SYMPOSIUM ON THE FOUNDATIONS OF SOFTWARE ENGINEERING, p. 1100–1104. ACM, 2019.

[25] GAMMA, E.; HELM, R.; JOHNSON, R. ; VLISSIDES, J.. **Design Patterns: Elements of Reusable Object-Oriented Software**. Pearson Educationl, 1994.

[26] GÖTZ, S.; PUKALL, M.. **On performance of delegation in java**. In: 2ND INTERNATIONAL WORKSHOP ON HOT TOPICS IN SOFTWARE UPGRADES, HotSWUp '09. ACM, 2009.

[27] HAYASHI, S.; SAEKI, M. ; KURIHARA, M.. **Supporting refactoring activities using histories of program modification**. Transactions on Information and Systems, 89(4):1403–1412, 2006.

[28] JAKOBUS, B.; BARBOSA, E. A.; GARCIA, A. ; DE LUCENA, C. J. P.. **Contrasting exception handling code across languages: An experience report involving 50 open source projects**. In: IEEE 26TH ISSRE, p. 183–193, 2015.

[29] KIM, M.; ZIMMERMANN, T. ; NAGAPPAN, N.. **A field study of refactoring challenges and benefits**. In: PROCEEDINGS OF THE ACM SIGSOFT 20TH INTERNATIONAL SYMPOSIUM ON THE FOUNDATIONS OF SOFTWARE ENGINEERING, p. 1–11, 2012.

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

[31] LU, M.; LIANG, P.. **Automatic classification of non-functional requirements from augmented app user reviews**. In: 21ST EASE, p. 344–353. ACM, 2017.

[32] MENS, T.; TOURWÉ, T.. **A survey of software refactoring**. IEEE Transactions on software engineering, 30(2):126–139, 2004.

[33] MOSHTARI, S.; SAMI, A.. **Evaluating and comparing complexity, coupling and a new proposed set of coupling metrics in cross-project vulnerability prediction**. In: 31ST ANNUAL SYMPOSIUM ON APPLIED COMPUTING, p. 1415–1421. ACM, 2016.

[34] MURPHY-HILL, E.; PARNIN, C. ; BLACK, A. P.. **How we refactor, and how we know it**. TSE, 38(1):5–18, 2011.

[35] OIZUMI, W.; SOUSA, L.; OLIVEIRA, A.; GARCIA, A.; AGBACHI, A. B.; OLIVEIRA, R. ; LUCENA, C.. **On the identification of design problems in stinky code: experiences and tool support**. Journal of the Brazilian Computer Society, 24(1):13, 2018.

[36] OIZUMI, W.; SOUSA, L.; OLIVEIRA, A.; CARVALHO, L.; GARCIA, A.; COLANZI, T. ; OLIVEIRA, R.. **On the density and diversity of**

degradation symptoms in refactored classes: A multi-case study. In: 2019 IEEE 30TH INTERNATIONAL SYMPOSIUM ON SOFTWARE RELIABILITY ENGINEERING (ISSRE), p. 346–357. IEEE, 2019.

[37] OLBRICH, S.; CRUZES, D. S.; BASILI, V. ; ZAZWORKA, N.. **The evolution and impact of code smells: A case study of two open source systems**. In: 2009 3RD INTERNATIONAL SYMPOSIUM ON EMPIRICAL SOFTWARE ENGINEERING AND MEASUREMENT, p. 390–400. IEEE, 2009.

[38] PAIXAO, M.; HARMAN, M.; ZHANG, Y. ; YU, Y.. **An empirical study of cohesion and coupling: Balancing optimization and disruption**. IEEE Transactions on Evolutionary Computation, 22(3):394–414, 2017.

[39] PALOMBA, F.; BAVOTA, G.; DI PENTA, M.; OLIVETO, R. ; DE LUCIA, A.. **Do they really smell bad? a study on developers' perception of bad code smells**. In: 2014 IEEE INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE AND EVOLUTION, p. 101–110. IEEE, 2014.

[40] PARNIN, C.; GÖRG, C.. **Lightweight visualizations for inspecting code smells**. In: SYMPOSIUM ON SOFTWARE VISUALIZATION, p. 171–172. ACM, 2006.

[41] PETKE, J.; HARMAN, M.; LANGDON, W. B. ; WEIMER, W.. **Specialising software for different downstream applications using genetic improvement and code transplantation**. IEEE TSE, 44(6):574–594, 2018.

[42] RATZINGER, J.. **sPACE – Software Project Assessment in the Course of Evolution**. Doctoral dissertation, Vienna University of Technology, 2007.

[43] SANTOS, J. A. M.; ROCHA-JUNIOR, J. B.; PRATES, L. C. L.; DO NASCIMENTO, R. S.; FREITAS, M. F. ; DE MENDONÇA, M. G.. **A systematic review on the code smell effect**. Journal of Systems and Software, 144:450–477, 2018.

[44] SIAVVAS, M.; KEHAGIAS, D.AND TZOVARAS, D.. **A preliminary study on the relationship among software metrics and specific vulnerability types**. In: INTERNATIONAL CONFERENCE ON COMPUTATIONAL SCIENCE AND COMPUTATIONAL INTELLIGENCE (CSCI), p. 916–921, 2017.

[45] SIEGMUND, N.; KUHLEMANN, M.; PUKALL, M. ; APEL, S.. **Optimizing non-functional properties of software product lines by means of refactorings**. In: 4TH INTERNATIONAL WORKSHOP ON VARIABILITY MODELLING OF SOFTWARE-INTENSIVE SYSTEMS, volumen 37, p. 115–122. Universität Duisburg-Essen, 2010.

[46] SMITH, C. U.; WILLIAMS, L. G.. **Software performance antipatterns**. In: 2ND INTERNATIONAL WORKSHOP ON SOFTWARE AND PERFORMANCE, 2000.

[47] SOARES, V.; OLIVEIRA, A.; PEREIRA, J. A.; BIBANO, A. C.; GARCIA, A.; FARAH, P. R.; VERGILIO, S. R.; SCHOTS, M.; SILVA, C.; COUTINHO, D. ; OTHERS. **On the relation between complexity, explicitness, effectiveness of refactorings and non-functional concerns**. In: PROCEEDINGS OF THE 34TH BRAZILIAN SYMPOSIUM ON SOFTWARE ENGINEERING, p. 788–797, 2020.

[48] SOARES, V.; OLIVEIRA, A.; PEREIRA, J.; BIBANO, A. C.; GARCIA, A.; FARAH, P. R.; VERGILIO, S.; SCHOTS, M.; SILVA, C.; COUTINHO, D.; OLIVEIRA, D. ; UCHÔA, A.. **Website**, 2020.

[49] SOARES, V.; OLIVEIRA, A.; PEREIRA, J.; BIBANO, A. C.; GARCIA, A.; FARAH, P. R.; VERGILIO, S.; SCHOTS, M.; SILVA, C.; COUTINHO, D.; OLIVEIRA, D. ; UCHÔA, A.. **Website**, 2021.

[50] SOUSA, L.; CEDRIM, D.; GARCIA, A.; OIZUMI, W.; BIBIANO, A. C.; TENORIO, D.; KIM, M. ; OLIVEIRA, A.. **Characterizing and identifying composite refactorings: Concepts, heuristics and patterns**. In: 17TH ICSE, 2020.

[51] TENORIO, D.; BIBIANO, A. C. ; GARCIA, A.. **On the customization of batch refactoring**. In: 3RD IWOR, p. 13–16. IEEE Press, 2019.

[52] TSANTALIS, N.; MANSOURI, M.; ESHKEVARI, L. M.; MAZINANIAN, D. ; DIG, D.. **Accurate and efficient refactoring detection in commit history**. In: 40TH ICSE, p. 483–494. ACM, 2018.

[53] TSANTALIS, N.; KETKAR, A. ; DIG, D.. **Refactoringminer 2.0**. IEEE Transactions on Software Engineering, 2020.

[54] SCIENTIFIC TOOLWORKS, INC. **Understand**, 2020.

[55] VAKILIAN, M.; CHEN, N.; NEGARA, S.; RAJKUMAR, B. A.; BAILEY, B. P. ; JOHNSON, R. E.. **Use, disuse, and misuse of automated refactorings**. In: 34TH ICSE, p. 233–243. IEEE Press, 2012.

[56] VAN EMDEN, E.; MOONEN, L.. **Java quality assurance by detecting code smells**. In: NINTH WORKING CONFERENCE ON REVERSE ENGINEERING, 2002. PROCEEDINGS., p. 97–106. IEEE, 2002.