Pontifícia Universidade Católica do Rio de Janeiro

Eduardo Moreira Fernandes

On the Relation between Refactoring and Critical Internal Attributes when Evolving Software Features

Tese de Doutorado

Thesis presented to the Programa de Pós–graduação em Informática of PUC-Rio in partial fulfillment of the requirements for the degree of Doutor em Ciências – Informática.

Advisor: Prof. Marcos Kalinowski

Rio de Janeiro
March 2021

Eduardo Moreira Fernandes

On the Relation between Refactoring and Critical Internal Attributes when Evolving Software Features

Thesis presented to the Programa de Pós–graduação em Informática of PUC-Rio in partial fulfillment of the requirements for the degree of Doutor em Ciências – Informática. Approved by the Examination Committee.

Prof. Marcos Kalinowski
Advisor
Departamento de Informática – PUC-Rio

Prof. Helio Côrtes Vieira Lopes
Pontifícia Universidade Católica do Rio de Janeiro – PUC-Rio

Prof.ª Simone Diniz Junqueira Barbosa
Pontifícia Universidade Católica do Rio de Janeiro – PUC-Rio

Prof. Marco Tulio de Oliveira Valente
Universidade Federal de Minas Gerais – UFMG

Prof.ª Tayana Uchôa Conte
Universidade Federal do Amazonas – UFAM

Rio de Janeiro, March 12th, 2021

Eduardo Moreira Fernandes
Substitute Professor in the Faculty of Computing (FACOM) at the Federal University of Mato Grosso do Sul (UFMS) since 2021. He obtained his PhD in Informatics from the Pontifical Catholic University of Rio de Janeiro (PUC-Rio) in 2021. He also obtained his Master's and Bachelor's degrees in Computer Science from the Federal University of Minas Gerais (UFMG) and UFMS in 2017 and 2014, respectively. Eduardo was a visiting researcher in Data Analytics at the Newcastle University (NCL) in 2017. He also served as software engineering intern for Tecgraf Institute at PUC-Rio (2018-2019) and Governança Brasil (2017). He has joined Research & Development (R&D) projects in many topics, such as applied software engineering, information systems for healthcare, and education in computing. His collaborative work resulted in more than 30 peer-reviewed papers and three best paper awards or nominations. Eduardo has earned the 2019 FAPERJ distinguished PhD scholarship, the 2014 SBC distinguished undergraduate student award, and a few other acknowledgments.

Bibliographic data

Moreira Fernandes, Eduardo

On the Relation between Refactoring and Critical Internal Attributes when Evolving Software Features / Eduardo Moreira Fernandes; advisor: Marcos Kalinowski. – Rio de janeiro: PUC-Rio, Departamento de Informática, 2021.

v., 172 f: il. color. ; 30 cm

Tese (doutorado) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática.

Inclui bibliografia

1. Informática – Teses. 2. Refatoração;. 3. Atributo Interno de Qualidade;. 4. Evolução de Software;. 5. Funcionalidade;. 6. Mineração de Repositório de Software;. 7. Grupo Focal.. I. Kalinowski, Marcos. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

Acknowledgments

I thank God almighty for who I am and for all I have achieved, especially during the last decade of study and work. He has been constantly showing love and care for me, even though I do not deserve any of this. I also thank my beloved family, as well as my extended family and friends spread across Brazil and all over the globe, for their kindness and support.

I thank the advisor of this doctoral thesis for his willingness to guide me in a moment of great need. Some people may destroy us at our core, but others are as you have been – they help us recover. I also thank each well-inclined person I had the chance to collaborate with as an early career researcher. Thanks for letting us learn and evolve together – this is priceless.

I particularly thank the committee members of my doctoral thesis defense for letting us share experiences, points of view, and lessons learned. In addition, I heartily thank the Pontifical Catholic University of Rio de Janeiro (PUC-Rio) family – professors, secretaries, cleaners, janitors etc. – for creating such a welcoming environment to professional and personal growth.

Last, but not least, I thank the Brazilian National Council for Scientific and Technological Development (CNPq) and the Fundação de Amparo à Pesquisa do Estado do Rio de Janeiro (FAPERJ) for providing me with doctoral scholarship, thereby helping this doctoral thesis come to life.

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

Abstract

Moreira Fernandes, Eduardo; Kalinowski, Marcos (Advisor). **On the Relation between Refactoring and Critical Internal Attributes when Evolving Software Features**. Rio de Janeiro, 2021. 172p. Tese de doutorado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Context: Several software changes applied while evolving software features aim at improving internal quality attributes, e.g. cohesion. These changes are the refactorings. Non-assisted refactorings might worsen, rather than improve, internal attributes. However, current knowledge is insufficient for managing internal attributes during software evolution. **Objective:** Our first objective is assessing how refactorings affect internal attributes during software evolution by filling gaps of past work on study scope. Our second objective is filling gaps of qualitative evidence on how to manage critical internal attributes via refactorings while evolving features. An internal attribute is critical when its measurement has anomalous values. Low cohesion is an example of critical attribute. **Method:** Our first study extends a large quantitative assessment of the relationship between refactorings and five internal attributes: cohesion, complexity, coupling, inheritance, and size. We include a more detailed statistical analysis and address major threats to validity of past work. Our second study is a qualitative case study based on focus group. We selected two industry cases to promote discussions on how much (and why) critical attributes are relevant while evolving features. Finally, we crossed the findings from both conducted studies aimed at discussing how critical attributes can be addressed via refactoring when evolving features. **Results:** About 64% of refactorings either improve or keep the internal attributes unaffected. Developers seem to perform refactorings until the most relevant internal attributes are improved, thereby neglecting other internal attributes that may be critical. Low cohesion and high complexity are perceived as relevant because they often make evolving features harder than usual. High coupling, large inheritance, and large size are perceived as irrelevant when developers implement especially complex features. By crossing the findings from both studies, we discuss how refactorings can improve internal attributes, especially the critical ones. **Conclusions:** The findings of our studies can support managing critical attributes that developers typically find relevant, while preserving other attributes that may become critical.

Keywords

Refactoring; Internal Quality Attribute; Software Evolution; Feature; Mining Software Repository; Focus Group.

Resumo

Moreira Fernandes, Eduardo; Kalinowski, Marcos. **Sobre a Relação entre Refatoração e Atributos Internos Críticos ao Evoluir Funcionalidades de Software**. Rio de Janeiro, 2021. 172p. Tese de Doutorado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Contexto: Várias mudanças de código aplicadas ao evoluir funcionalidades visam melhorar atributos internos de qualidade como coesão. Tais mudanças são as refatorações. Refatorações não dirigidas podem piorar, e não melhorar, atributos internos. Porém, o saber atual é insuficiente para gerir atributos internos durante a evolução do sistema. **Objetivo:** Nosso primeiro objetivo é entender como refatorações afetam atributos internos ao evoluir sistemas, mitigando limitações de escopo de estudos anteriores. Nosso segundo objetivo é atender uma carência por evidência quantitativa sobre como gerir atributos internos críticos via refatorações ao evoluir sistemas. Um atributo interno é crítico se sua medição assume valores anômalos. Baixa coesão é um exemplo de atributo crítico. **Método:** O primeiro estudo estende uma avaliação quantitativa da relação entre refatorações e cinco atributos internos: acoplamento, coesão, complexidade, herança e tamanho. Incluímos novas análises e resolvemos ameaças à validade da literatura. O segundo estudo contém estudos de caso qualitativos baseados em grupo focal. Em dois casos industriais, promovemos discussões sobre o quanto (e por que) atributos críticos são relevante ao evoluir funcionalidades. Por fim, cruzamos os achados dos dois estudos para discutir como gerir atributos críticos via refatoração ao evoluir funcionalidades. **Resultados:** Aproximadamente 64% das refatorações melhoram ou não afetam os atributos internos. Desenvolvedores parecem refatorar até melhorar os atributos mais relevantes, ignorando outros atributos internos possivelmente críticos. Baixa coesão e alta complexidade são percebidos como relevantes e tornam mais difícil evoluir funcionalidades. Alto acoplamento, herança larga e tamanho largo são percebidos como irrelevantes ao implementar funcionalidades especialmente complexas, por exemplo. Ao cruzar dados entre estudos, discutimos como refatorações podem melhorar atributos internos, inclusive atributos críticos. **Conclusões:** Os achados dos nossos estudos podem apoiar a gestão de atributos críticos relevantes aos desenvolvedores, mas também preservar outros atributos que podem se tornar críticos.

Palavras-chave

Refatoração; Atributo Interno de Qualidade; Evolução de Software; Funcionalidade; Mineração de Repositório de Software; Grupo Focal.

Table of contents

List of figures

List of tables

But I shall look to Yahweh,
my hope is in the God who will save me;
my God will hear me.

Micah, *The Book of Micah 7:7.*

1
Introduction

Software evolution means incrementally adapting an existing software system according to the demands of stakeholders and the system operating environment (Lehman, 1980). This process consists of applying one or more changes over versions of the same system (Mens et al., 2010). After completing the initial development of a system, software evolution aims at adapting the system based on ever-changing software features and environment settings. Feature is a unit of functionality of the system associated to one or more design or domain decisions (Apel and Kästner, 2009). These decisions cover all software development phases rather than exclusively architectural design.

Based on the current knowledge (Elfatatry, 2007; Lehman, 1980; Paixao et al., 2019), software change is a unit of modifications applied to a system. These modifications adapt the existing features as new stakeholders' demands and environment settings emerge (Elfatatry, 2007). Features are not restricted to the documentation elaborated in the earliest development phases. This is because, in practical settings, new software features may emerge anytime along with the software design, implementation, and other subsequent phases (Martin, 2002). Each change reflects a particular demand for adapting the existing software system.

Developers apply several changes to their systems along with software evolution. This observation stands for both open source systems and closed source systems (Kim et al., 2014; Murphy-Hill et al., 2012; Silva et al., 2016a). Changes vary in granularity, i.e., the scale of the software artifact to be changed (Elfatatry, 2007). Many of these changes target the code structure, and the effect of a change may be either local or extensive. Each change affects the code structure at different granularities: from fine-grained elements, e.g. attributes and methods, to coarse-grained elements, e.g. components and interfaces (Chávez et al., 2017; Paixao et al., 2019).

In addition, each change is often associated with one or more developer intents (Paixao et al., 2019; Silva et al., 2016a; Tao et al., 2012). An intent reflects the developer expectation behind the application of a change. One of the most frequent intents is evolving software features (Kim et al., 2014; Paixao et al., 2019; Silva et al., 2016a). Evolving features consists of either incorporating new features into a system or enhancing existing features (Burke, 2014). This particular intent often co-occurs with the enhancement of code structures (Gousios et al., 2015), especially through refactorings (Fernandes, 2019a; Paixao et al., 2019; Silva et al., 2016a).

Applying changes while evolving features is not trivial. An undisciplined application of changes may degrade the code structure and its design (Szőke et al., 2015a; Tufano et al., 2017). As a system evolves, developers – consciously or not – introduce anomalous code structures that may threaten software evolution (Fernandes et al., 2017b; Palomba et al., 2014; Taibi et al., 2017). Moreover, developers may take sub-optimal design decisions leading to code and design degradation (Lin et al., 2016; Tufano et al., 2017).

Neglecting or postponing the enhancement of anomalous code structures may increase software evolution costs (Lehman, 1980; Mens et al., 2010). Extreme cases imply reengineering the entire system (Lanubile and Visaggio, 1995).

1.1 Degradation Symptoms and Software Evolution

The literature in software engineering has introduced different mechanisms for tracking anomalous code structures that suggest a degraded code structure or design (Chávez et al., 2017; Chidamber and Kemerer, 1994; Lanza and Marinescu, 2006). In particular, many anomalous code structures emerge along with software evolution (Bavota et al., 2015; Bibiano et al., 2019; Fernandes et al., 2017b; Tufano et al., 2017). Hereafter, we refer to these mechanisms as degradation symptoms. We summarize below two degradation symptoms often investigated by past work. Prior to that, however, we introduce the concept of internal quality attribute.

Each internal attribute targets a particular quality aspect of the internal structure of a system (Chávez et al., 2017; Fernandes et al., 2020). Cohesion, complexity, coupling, inheritance, and size (Chidamber and Kemerer, 1994; Lanza and Marinescu, 2006; Lorenz and Kidd, 1994; McCabe, 1976) are examples of traditional internal attributes. Developers usually capture an internal attribute based on one or more software metrics (Chávez et al., 2017; Fernandes et al., 2020). A plenty of metrics address different internal attributes (Chidamber and Kemerer, 1994; Lanza and Marinescu, 2006; Lorenz and Kidd, 1994). For instance, Lack of Cohesion (LCOM2) is an example of metric aimed at capturing the degree of class cohesion (Chidamber and Kemerer, 1994), while Cyclomatic Complexity

(CC) captures the complexity degree of a method (McCabe, 1976).

Internal attributes may help monitor parts of the source code that may increase the difficulty of evolving features (Chaparro et al., 2014; Chávez et al., 2017; Destefanis et al., 2014). Such monitoring is enabled by the analysis of a degradation symptom called *critical internal attribute*. Each critical attribute is an internal attribute whose metrics used for computing it assume anomalous values in comparison with a reference value (Vale et al., 2018). Developers may derive reference values in many ways, such as computing the distribution of metric values extracted from a set of systems (Vale et al., 2018). Examples of critical attributes often discussed in the literature include low cohesion, high complexity, high coupling, large inheritance, and large size (Fernandes et al., 2020).

Another degradation symptom is the so-called *design smell*. Design smells are recurring code structures that potentially hinder software evolution (Fowler, 2018; Lanza and Marinescu, 2006). Design smell types include Large Class, i.e. a too large and complex class, and Long Method, i.e. a too long and complex method. Critical attributes serve as a basis for detecting design smells (Bibiano et al., 2019; Fowler, 2018). That is, the definition of many design smell types relies on capturing two or more critical attributes (Fowler, 2018; Lanza and Marinescu, 2006). For instance, Large Class is usually computed by combining low cohesion, high complexity, and large size (Fernandes et al., 2017b; Vale et al., 2018).

By definition, both critical attributes and design smells are purely symptoms of degraded code structure and its design. Hence, their occurrence in a system may not represent a threat to evolving features. Representing a threat or not depends on the developer's perception (Oliveira et al., 2020b). This doctoral thesis is particularly concerned on *how much (and why) developers perceive critical attributes as relevant for evolving features*. That is, we aim at understanding the circumstances that make developers want to mitigate or fully address critical attributes for the sake of software evolution.

1.2 Limited Knowledge on Critical Attributes and Software Evolution

Previous studies (Bibiano et al., 2019; Chaparro et al., 2014; Le et al., 2016; Pantiuchina et al., 2018) investigated the relationship between changes and the occurrence of degradation symptoms potentially harmful to software evolution. Nevertheless, the majority of these studies is limited in scope and has several flaws in methodology and procedures. A result is that our current knowledge is insufficient for managing degradation symptoms that developers find relevant, especially while evolving features, through refactoring recommendations. In particular, we discuss below how extending the current knowledge on critical attributes could help in evolving features.

A recent work (Silva et al., 2016b) suggests that about 40% of pull request rejections in open source systems is due to degraded code structure and design. In this case, understanding how refactorings affect internal attributes could help developers in deciding what refactorings to apply. Thus, developers could improve those internal attributes they prioritize as relevant. Moreover, revealing the most relevant critical attributes from the developer's perception could drive refactoring recommendations for effectively enhancing code structures. Thus, developers could reduce their rates of pull request rejections.

A major limitation of previous studies is **focusing too much on enhancing code structures** rather than addressing other intents. Certain studies (Bavota et al., 2015; Bibiano et al., 2019; Du Bois and Mens, 2003) gave us useful hints on how refactorings may unexpectedly degrade the code structure and its design. However, they assume that developers apply most refactorings intended at mitigating or fully addressing degradation symptoms. As a result, they ignore that the majority of refactorings are applied with other intents in mind (Murphy-Hill et al., 2012; Silva et al., 2016a). Particularly, 73% of refactorings are somehow related with evolving features (Chávez et al., 2017).

Additionally, current knowledge on the relationship between refactorings and the occurrence of degradation symptoms is quite discouraging. Although refactorings are especially designed for enhancing code structures, less than 10% of them suffice to fully address degradation symptoms such as design smells (Bavota et al., 2015; Bibiano et al., 2019; Bibiano et al., 2020). By assuming that enhancing code structures is the intent behind most refactorings, these studies ignore that other intents may have been a priority (e.g., evolving features). Thus, recommending refactoring solely based on how much they enhance code structures is a naive strategy for managing degradation symptoms – including critical attributes.

Another major limitation of previous studies is their **very strict investigation scope**, thereby supporting a limited view on the refactoring effect – i.e. improvement or worsening of internal attributes after performing refactorings. Anomalous metric values (Chaparro et al., 2014; Pantiuchina et al., 2018) and design smells (Bibiano et al., 2019; Mantyla et al., 2004; Palomba et al., 2014; Taibi et al., 2017) have been the main target of studies on degradation symptoms. Still, previous studies typically assessed a small variety of metrics and design smell types. Particularly, studies regarding critical attributes (Bavota et al., 2015; Chaparro et al., 2014; Kim et al., 2014;

Veerappa and Harrison, 2013) investigated only a few critical attributes – mostly, cohesion and complexity. Some studies adopted small case studies rather than large-scale studies.

In addition, previous studies (Palomba et al., 2014; Taibi et al., 2017) captured the developer's perception on how relevant design smells are to software evolution. Contrary to expectations, only a few design smell types are reportedly potential threats to evolving features. For instance, while Duplicated Code may hinder code reuse (Taibi et al., 2017), Large Class can increase the difficulty to understand and perform changes (Palomba et al., 2014). However, types like Lazy Class and Long Parameter List (Fowler, 2018) are not harmful to software evolution after all. Curiously, to the best of our knowledge, no past work captured the developer's perception on how much (and why) critical attributes are relevant for evolving features.

Different of previous studies, the majority of this doctoral thesis targets changes whose intent is software evolution rather than the pure enhancement of code structures. We aim to shed light on the refactoring effect on internal attributes that may become critical and, still, are not relevant for developers while evolving features. Furthermore, instead of purely quantifying the critical attributes phenomenon, we qualitatively capture the developer's perception on how much each critical attribute is relevant in practical settings.

We summarize below the **General Limitation** addressed by this thesis.

General Limitation: *The current knowledge on how refactorings affect internal attributes and on how critical attributes can be addressed through refactorings is insufficient to assist developers when evolving features.*

1.3 On the Relationship between Refactorings and Internal Attributes

Refactoring means applying changes particularly designed for enhancing the internal structure of a system (Fowler, 2018; Liu et al., 2011). Fowler's Refactoring book (Fowler, 2018) is the most extensive catalog of refactorings to this date. Since the first release of this catalog in 1999, refactoring practices spread across major software companies, e.g. Google (Potvin and Levenberg, 2016) and Microsoft (Kim et al., 2014), and large open source projects (Bavota et al., 2015; Chaparro et al., 2014; Paixao et al., 2019). Particularly, refactoring has been incorporated into practices of software quality assurance, such as code review (Paixao et al., 2019; Sadowski et al., 2018).

Each refactoring varies in type according to the expected change. Examples of popular refactoring types in industry (Murphy-Hill et al., 2012; Silva et al., 2016a) are Extract Method, i.e. extracting a new method from an existing method, and Move Method, i.e. moving an existing method across classes. Refactoring types are associated with the expected enhancement of code structures. For instance, Extract Method may mitigate of fully address critical attributes like high complexity and large size in a method. Similarly, Move Method may reduce high coupling and increase cohesion in the original class of a moved method (Al Dallal and Abdin, 2017; Chávez et al., 2017; Fernandes et al., 2020).

The relationship between refactorings and internal attributes still lacks empirical validation. Two major literature limitations restrict our view on how refactorings affects traditional internal attributes, such as cohesion and complexity. The **first limitation** regards the very strict scope of internal attributes and metrics analyzed so far. Recent studies (Bavota et al., 2015; Chaparro et al., 2014; Kim et al., 2014; Veerappa and Harrison, 2013) assessed either a few internal attributes, mostly cohesion and coupling, or a few metrics for capturing each attribute. As a result, these studies say little on how refactorings affect internal attributes, regardless of being critical attributes.

The **second limitation** regards the lack of empirical investigation on re-refactoring. The *re-refactoring* phenomenon occurs whenever one or more refactorings are applied to a previously refactored code element, e.g. a method or a class (Fernandes et al., 2020). Once developers often perform re-refactorings, one could assume a more significant enhancement of the code structure and its design via re-refactoring when compared to each single refactoring. For instance, this is what previous studies assume to some extent (Chávez et al., 2017; Fowler, 2018; Jiau et al., 2013). That enhancement could be even more significant depending the inherent complexity of code structure and design degradation (Bibiano et al., 2019; Bibiano et al., 2020). Nevertheless, to the best of our knowledge, no empirical study has validated this assumption, especially through the analysis of a large set of systems.

To address the aforementioned limitations is essential for assisting developers with refactorings to help manage critical attributes while evolving features. For instance, while adding or enhancing features in a system, developers could carefully choose what refactoring types to apply for the sake of software evolution. Developers could also decide whether to keep refactoring the same code elements, in a re-refactoring manner, until the code structure and its design is sufficiently enhanced. Finally, researchers could recommend effective refactorings for improving internal attributes that developers typically find relevant, while preserving other internal attributes that may become critical.

We summarize below the **Specific Limitation 1** addressed by this doctoral thesis, based on the discussion above.

Specific Limitation 1: *The current knowledge on how refactorings (and re-refactorings) affect internal attributes, by means of attribute improvement or worsening, is insufficient for managing critical attributes while evolving features.*

We address Specific Limitation 1 via a **large quantitative study**. We extend a previous work (Chávez et al., 2017) about the refactoring effect on five internal attributes – i.e., cohesion, complexity, coupling, inheritance, and size. In order to overcome limitations of past work (Bavota et al., 2015; Chaparro et al., 2014), we analyze 23 open source systems with 29,303 refactorings in total, from which nearly 50% of them constitutes re-refactorings. Our study has the largest scope to this date: 11 refactoring types and 25 metrics. We assess three categories of refactoring effect: attribute improvement, attribute worsening, and cases in which neither an attribute improvement nor an attribute worsening occurs.

We advocate the importance of further investigating such a fine-grained degradation symptom as internal attributes. The literature suggests that degradation symptoms are often interrelated (Fernandes et al., 2017b). For instance, low cohesion and high coupling are closely associated with degradation symptoms at the architectural level (Samarthyam et al., 2016). We also discussed in Section 1.1 that detecting design smells like Large Class requires combining two or more critical attributes, e.g. low cohesion, high complexity, and large size.

Our **study results** reveal unprecedented details about the effect of refactorings and re-refactorings on internal attributes. In contrast to a recent work (Bavota et al., 2015), we found out that *94% of refactorings and re-refactorings occur in code elements with at least one critical attribute*. In other words, the majority of (re-)refactorings occur in parts of the source code with some sort of degraded code structure and design. Moreover, *refactorings are 85 times more likely to occur in code elements without critical attributes than re-refactorings*. This result suggests that developers apply re-refactorings on parts of the source code that remain degraded along with software evolution.

Other results are equally interesting. About 73% of refactorings co-occur with other changes often aimed at evolving features. This result suggests that mitigating or fully addressing critical attributes is relevant to perform software evolution. In addition, refactorings either improve or keep unaffected the internal attributes in 64% of the cases. Re-refactorings had similar results. We hypothesize that either i) refactorings were insufficient to fully address critical attributes in the 36% remaining cases or ii) developers only address internal attributes they find relevant for evolving features. However, our quantitative data is insufficient to confirm any of these hypotheses.

1.4 On the Relevance of Critical Attributes for Evolving Features

Many previous studies (Bavota et al., 2013; Bavota et al., 2015; Bibiano et al., 2019; Bibiano et al., 2020; Chaparro et al., 2014) investigated the refactoring effect on different degradation symptoms. These symptoms range from anomalous metric values to design smells and, in the case of the work we have extended (Chávez et al., 2017), critical attributes. Unfortunately, the majority of study results published so far somehow discourage the use of refactorings in practical settings. On the one hand, applying a single refactoring on a code element rarely suffices in fully addressing design smells (Yoshida et al., 2016). The same result is valid when combining or re-applying refactorings (Bibiano et al., 2019; Bibiano et al., 2020).

While analyzing previous studies on this research topic, we observe a research bias towards performing quantitative rather than qualitative studies on the refactoring effect. We have found very few studies aimed at capturing the developer's perception of what degradation symptoms are relevant for evolving features (Palomba et al., 2014; Taibi et al., 2017; Yamashita and Moonen, 2013). By the way, *we refer to relevance as the need for either mitigating or fully addressing a degradation symptom for the sake of software evolution.* Previous studies conclude, for instance, that Duplicated Code and Large Class may hinder code reuse and evolution, while Lazy Class and Long Parameter List represent no threat after all (Palomba et al., 2014; Taibi et al., 2017).

Unfortunately, all the aforementioned studies (Palomba et al., 2014; Taibi et al., 2017; Yamashita and Moonen, 2013) on the developer's perception of degradation symptoms target design smells only. As we discussed in Section 1.1, design smell types are typically detected by combining two or more critical attributes. Once past work provides evidence on how developers perceive combinations of attributes in the form of design smells, these studies give us preliminary hints of how relevant each critical attribute may be. Still, the lack of studies especially designed to capture the developer's perception of critical attributes as relevant while evolving features has two major implications. We discuss below each implication.

The **first implication** is this: refactoring recommendations derived from past work may not match the recurring intents behind refactorings in industry. Inherent limitations of quantitative data extracted from software repositories, as collected in (Bavota et al., 2015; Bibiano et al., 2019), make it hard to understand whether developers: i) actually failed in fully addressing critical attributes or ii) have just ignored the improvement or worsening of critical attributes they find irrelevant. Refactoring recommendations based on these studies tend to optimize attribute improvement, regardless of attributes being irrelevant.

The **second implication** derives from the first one. Shortly, developers will probably reject refactoring recommendations that overlook their perception of what critical attributes are actually relevant. Previous studies suggest that developers are often reluctant in refactoring their systems (Kim et al., 2014; Silva et al., 2016a), mostly because they fear unexpected behavioral changes. However, in our case, the problem is simpler: developers may not see value in adopting the recommended refactorings. They will probably prefer adding new features or enhancing existing features for rapidly meeting the stakeholders' needs.

This doctoral thesis proposes a different approach to provide recommendations for managing critical attributes. We aim at capturing the developer's perception on how much each traditional critical attribute is relevant for evolving features. As a complement, we aim at capturing why certain critical attributes are more relevant than others are and, thereby, developers must mitigate or fully address them for the sake of software evolution. To the best of our knowledge, this is the first qualitative study on this matter. We expect to fill an apparent gap between current refactoring recommendations and developer's concerns on managing critical attributes in practice.

We summarize below the **Specific Limitation 2** addressed by this doctoral thesis, based on the discussion above.

Specific Limitation 2: *The current knowledge on how much (and why) critical attributes are perceived as relevant by developers while evolving features is scarce if not non-existent.*

Aimed at addressing **Specific Limitation 2**, we designed a **qualitative case study** based on focus group sessions. We chose two industry cases in order to promote discussion on how much (and why) critical attributes are relevant while evolving features. This study relies on strict guidelines for conducting case study research in software engineering (Runeson and Höst, 2009).

We opted for investigating critical attributes at the class level only, e.g. high class coupling. We believe that our investigation at the class level is sufficient for acquiring a broad understanding on the relevance of critical attributes. Indeed, classes are key code elements constituted of many others, including attributes and methods. Additionally, we had the need for promoting discussions, in each focus group session, that do not take longer than real development teams may participate. Thus, we opted for restricting our study to class level, thereby discarding critical attributes at levels such as method.

Following strict guidelines to perform focus groups (Kontio et al., 2004), we recruited for participation two development teams of an industry-academy joint initiative for Research and Development (R&D) in Brazil – i.e., the ExACTa initiative[1]. Each team engaged with a particular focus group session, in which developers discussed the five critical attributes mentioned in Section 1.1 and investigated in our previous work (Chávez et al., 2017): low cohesion, high complexity, high coupling, large inheritance, and large size (Fernandes et al., 2020). Each focus group session lasted up to two hours and was fully taken online, with the support of the MURAL interactive platform[2] and the Zoom Meeting tool[3].

Our **study results** reveal important aspects on the relevance of the five critical attributes mentioned above. On the one hand, we found out that low cohesion and high complexity are ultimately relevant for developers. This is because both critical attributes tend to increase the difficulty of evolving features in a class. On the other hand, high coupling, large inheritance, and large size are especially irrelevance when developers perform code reuse. Still, these critical attributes may concern developers while evolving features.

In addition, we crossed our data obtained through the quantitative study (Section 1.3) with our data obtained via our qualitative study (Section 1.4). The data crossing allowed us to derive a simple, but empirically derived, catalog of refactoring recommendations to help manage critical attributes while evolving features. It is worth mentioning that our recommendations partially rely on data regarding critical attributes at the class level. Thus, generalizing our study results to critical attributes at other levels of the system requires further investigation.

[1]http://www.exacta.inf.puc-rio.br/
[2]https://www.mural.co/
[3]https://zoom.us/pt-pt/meetings.html

1.5 Thesis Contributions

This doctoral thesis expands the current knowledge on degradation symptoms that are relevant for developers during the software evolution. Particularly, we provide unprecedented insights on how much (and why) developers have to mitigate or fully address critical attributes while evolving features. We rely on both a large quantitative study, based on mining software repositories, and two industry cases with developers often engaged in software evolution. We summarize below the contributions of this doctoral thesis.

Contribution 1: *Additional empirical evidence on how (re-)refactorings affect internal attributes* – We identified opportunities to expand the scope of previous studies (Section 1.3), e.g. by investigating a wider set of internal attributes and metrics. Additionally, we did not restrict our study to changes associated with single refactoring only, as made by past work (Al Dallal and Abdin, 2017; Bavota et al., 2015; Du Bois and Mens, 2003). Instead, we provided an unprecedented re-refactoring analysis. Finally, we strive for providing practitioners and developers with a large quantitative study based on the analysis of real systems.

As a result, we contribute with new insights on how refactorings and re-refactorings affect critical attributes often discussed in the literature – e.g., low cohesion, high complexity, and large size. Besides introducing a catalog of refactoring types and their respective effect on internal attributes, we contradict past assumptions and common wisdom. Finally, the results of our large quantitative study are documented in a journal paper (Fernandes et al., 2020).

Contribution 2: *Empirical evidence on the developer's perception regarding the relevance of critical attributes for evolving features* – As aforementioned, evolving features is a major industry demand (Gousios et al., 2015). Feature additions and enhancements are often performed in conjunction with the enhancement of code structures (Chávez et al., 2017; Paixao et al., 2019), mostly through refactorings. Still, we know little about what critical attributes are more relevant for performing software evolution from the developer's perception. More critically, we barely know what makes developers mitigate or fully address a critical attribute.

We present preliminary evidence on how much (and why) developers find the five critical attributes investigated in our quantitative study relevant (Fernandes et al., 2020). We relied on two industry cases where developers discussed the relevance degree of each critical attributes at the class level – e.g., low class cohesion. The developers also provided recurring reasons for either mitigating or fully addressing each critical attribute while evolving features.

Through this scientific contribution, we expect to inspire study replications in other industry contexts – once generality of case study results is typically limited. We also aim at inspiring the design of novel tools for assisting refactorings during software evolution.

Contribution 3: *A catalog of refactoring recommendations to help manage critical attributes while evolving features* – The lack of empirical knowledge on what critical attributes are relevant for evolving features may, in parts, justify the limitations of the current tools for assisting refactorings, e.g. (Lin et al., 2016; Szőke et al., 2015a). Existing tools mostly address the removal of degradation symptoms (including critical attributes) regardless of the intent behind refactorings. Hence, they provide little or no assistance to mitigating or fully addressing those critical attributes that may hinder adding or enhancing features.

We cross data from our quantitative study (Section 1.3) and qualitative study (Section 1.4). Thus, we derive practical recommendations of refactorings that may improve critical attributes often reported as relevant by the developers, while preventing the worsening of eventually critical attributes. We expect not only support practitioners in their daily work, but also inspire the design of novel tools for assisting refactorings.

1.6 Thesis Summary

We structured the remainder of this doctoral thesis as follows.

Chapter 2 provides background information aimed at supporting the understanding of our work. This chapter also discusses related work.

Chapter 3 introduces our large quantitative study on the relationship between refactorings and internal attributes. We present and discuss the main study results, as reported in our recently published journal paper (Fernandes et al., 2020).

Chapter 4 introduces our qualitative study case based on focus group sessions. We describe in details our study, summarize our major study results, and discuss some threats to validity – as treatments applied whenever possible.

Chapter 5 concludes this doctoral thesis. We discuss some major implication of our study findings and describe the set of publications achieving during this PhD course – some of them derived from this doctoral thesis.

2
Background and Related Work

One of the most important concepts in contemporary software development is software evolution. Shortly, software evolution consists of incrementally adapting a software system according to the ever-changing demands of stakeholders and system operating environment settings (Lehman, 1980). While performing software evolution, developers apply several software changes over versions of the same system (Mens et al., 2010). Each change is a unit of modifications applied to the system (Elfatatry, 2007; Lehman, 1980; Paixao et al., 2019). In general, each change in isolation reflects a particular need for evolving the system (Elfatatry, 2007).

Changes vary in granularity, i.e. the scale of the software artifact affected by modifications (Elfatatry, 2007). A single change may affect the system at granularities ranging from methods and attributes to components (Chávez et al., 2017; Paixao et al., 2019). In addition, each change is associated with one or more developer intents (Paixao et al., 2019; Silva et al., 2016a; Tao et al., 2012). An intent reflects a particular expectation of the developer with respect to the applied change. Enhancing code structures through refactorings (Fowler, 2018) and evolving software features (Fernandes, 2019a; Paixao et al., 2019) are two of the most recurring intents. Both intents often co-occur in industry (Kim et al., 2014; Paixao et al., 2019; Silva et al., 2016a).

Developers should avoid the introduction of degradation symptoms – hints of degraded code structure and design – for the sake of software evolution (Bavota et al., 2015; Fowler, 2018; Tufano et al., 2017). Two examples of recurring degradation symptoms are anomalous metric values (Bavota et al., 2015; Chávez et al., 2017; Fernandes et al., 2020) and design smells (Fowler, 2018). In addition, monitoring and correcting degradation symptoms as soon as they occur may facilitate future changes in a system. Unfortunately, developers tend to unexpectedly introduce several degradation symptoms to their systems, especially while performing changes without proper assistance (Tufano et al., 2017). Even small systems have hundreds of degradation symptoms (Fernandes et al., 2017b).

Particularly, the occurrence of degradation symptoms may make it harder to perform feature additions and enhancements over time (Mens et al., 2010). In this doctoral thesis, we call *relevant* a degradation symptom that developers need to either mitigate or fully address for supporting software evolution. Although highly recommended to developers, managing degradation symptoms through refactorings is far from being a trivial task. This is mostly due to two limitations of the current empirical knowledge. We discuss below these limitations.

The first limitation regards an insufficient knowledge on how refactorings affect degradation symptoms. Although previous studies (Bavota et al., 2015; Bibiano et al., 2019; Chaparro et al., 2014; Du Bois and Mens, 2003) attempted to address this literature gap, they are very limited in scope. Most studies investigated a small variety of degradation symptoms – mostly design smells. They also lacked a deep understanding on whether refactorings are capable of fully addressing each degradation symptom in isolation.

The second limitation regards an insufficient knowledge on what degradation symptoms are the most relevant from the developer's perception. Past work (Bavota et al., 2015; Bibiano et al., 2019; Bibiano et al., 2020) implicitly assumes that developers would either mitigate or fully address any degradation symptoms with the proper guidance. However, recent studies (Palomba et al., 2014; Pantiuchina et al., 2018; Taibi et al., 2017; Yamashita and Moonen, 2013) suggest that many degradation symptoms are irrelevant for developers, especially while evolving features. Thus, recommendations to help manage degradation symptoms should consider the developer's perception of relevance for matching the intents behind refactorings.

This doctoral thesis addresses the two aforementioned limitations in the context of a particular degradation symptom: critical internal attribute (Chávez et al., 2017; Fernandes et al., 2020). Each critical attribute is an internal attribute whose metrics used for computing it assume anomalous values in comparison with a reference value (Vale et al., 2018). Our first study extends a large quantitative assessment (Chávez et al., 2017) of the relationship between refactorings and five internal attributes: cohesion, complexity, coupling, inheritance, and size (Chidamber and Kemerer, 1994; Lanza and Marinescu, 2006; Lorenz and Kidd, 1994; McCabe, 1976). Our second study is a qualitative case study based on focus groups (Kontio et al., 2004) in the industry. We aim at understanding how much (and why) critical attributes are relevant while evolving features.

This chapter summarizes background and related work for supporting the understanding of this doctoral thesis. Section 2.1 defines critical attributes based on anomalous metric values. Section 2.2 summarizes and exemplifies design smells. We emphasize examples closely associated with the critical attributes discussed throughout the thesis. Section 2.3 discusses the current knowledge of developer's perception on degradation symptoms with respect to software evolution. Section 2.4 summarizes the basic concepts of refactoring and re-refactoring. Section 2.5 discusses who refactorings may help either mitigate or fully address degradation symptoms. Finally, Section 2.6 concludes this chapter.

2.1 Critical Attributes

Metrics are measurements designed to provide insights on the quality of a system (Lorenz and Kidd, 1994; Vale et al., 2018). A plenty of metrics have been cataloged in the literature (Chidamber and Kemerer, 1994; Henry and Kafura, 1981; Lanza and Marinescu, 2006; Lorenz and Kidd, 1994). In this thesis, we are particularly concerned on object-oriented metrics, i.e., metrics targeting systems implemented in object-oriented programming languages, from which Java stands out in industry[1]. Examples of metrics are Method Lines of Code (MLOC) (Zimmermann et al., 2007), which counts the number of code lines that constitute a method, and Coupling between Objects (CBO) (Chidamber and Kemerer, 1994), which counts the number of classes connected to a class.

Metric thresholds typically drive the metric value interpretation. Each threshold is a reference value τ for determining whether a metric value is anomalous. For instance, one could define an arbitrary $\tau = 100$ in such a way that MLOC $> \tau$ represents the range of anomalous MLOC values, once they suggest a method is too long. Thresholds can be derived in several ways, e.g. from the analysis of metrics distribution considering a set of systems (Vale et al., 2018). As an example, the mean value of a metric, let us say μ, could be set as a MLOC threshold so that MLOC $> \mu$ is the range of anomalous metric values, while MLOC $\leq \mu$ is the range of regular metric values.

Anomalous metric values have been used to monitoring what we refer to as internal quality attributes (Bavota et al., 2015; Chaparro et al., 2014; Chávez et al., 2017; Fernandes et al., 2020). Each internal attribute concerns an internal property of the system. This doctoral thesis targets the five internal attributes defined below. In particular, cohesion and coupling are the most recurrently investigated attributes (Fernandes et al., 2020). It is worth

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

mentioning that the definitions below are quite strict due to the scope of our work. For instance, although complexity (Kataoka et al., 2002; McCabe, 1976) could target other granularities rather than class, we are concerned about the class complexity.

- *Cohesion* (Chidamber and Kemerer, 1994; Lanza and Marinescu, 2006) captures the interrelation degree of code elements, i.e. attributes and methods, that constitute a class;
- *Complexity* (Kataoka et al., 2002; McCabe, 1976) targets the cognitive complexity degree of the code structure within a given class;
- *Coupling* (Chidamber and Kemerer, 1994; Kataoka et al., 2002) regards the inter-dependency degree of a class with others in terms of methods and attributes being used;
- *Inheritance* (Chidamber and Kemerer, 1994; Lanza and Marinescu, 2006) encompasses parent-child relationships between classes that constitute the class hierarchy in a system; and
- *Size* (Kataoka et al., 2002; Zimmermann et al., 2007) measures the length or amount of source code implemented by a certain code element.

Table 2.1 summarizes the set of 25 metrics assessed throughout this doctoral thesis. We have explored this set in our previous work on refactorings and their effect on internal software quality (Bibiano et al., 2019; Bibiano et al., 2020; Chávez et al., 2017). All metrics are grouped (second column) according to the internal attribute they aim at capturing (first column). The fifth column informs when a metric becomes critical whether its value either increases or decreases after performing a change. This column relies on our experiences with software development in industry combined with insights extracted from previous studies (Bavota et al., 2015; Chávez et al., 2017). For instance, high CBO values suggest classes with several features and dependencies (Fowler, 2018; Lanza and Marinescu, 2006). Thus, CBO becomes critical as its value increases for a given class.

Changes affect internal attributes in three ways (Chávez et al., 2017; Fernandes et al., 2020). They can *improve an internal attribute* if the metrics used for capturing it increase or decrease (depending on the fifth column of Table 2.1) towards becoming non-critical metrics. They can also *keep unaffected an internal attribute* by neither increasing nor decreasing the metrics used for capturing it. Finally, they can *worsen an internal attribute* if the metrics used for capturing it increase or decrease (also depending on the fifth column of Table 2.1) towards becoming critical metrics.

Table 2.1: Metrics Grouped by Internal Attribute

Attribute	Metric	Acronym	Granularity	Critical if
Cohesion	Lack of Cohesion of Methods (Chidamber and Kemerer, 1994)	LCOM2	Class	Increases
	Lack of Cohesion of Methods (Li and Henry, 1993)	LCOM3	Class	Increases
	Tight Class Cohesion (Bieman and Kang, 1995)	TCC	Class	Decreases
Complexity	Cyclomatic Complexity (McCabe, 1976)	CC	Method	Increases
	Essential Complexity (McCabe, 1976)	Evg	Method	Increases
	Nesting	MaxNest	Method	Increases
	Paths (Nejmeh, 1988)	NPATH	Method	Increases
	Weighted Method per Class (Chidamber and Kemerer, 1994)	WMC	Class	Increases
Coupling	Coupling between Objects (Chidamber and Kemerer, 1994)	CBO	Class	Increases
	Coupling Dispersion (Lanza and Marinescu, 2006)	CDISP	Method	Increases
	Coupling Intensity (Lanza and Marinescu, 2006)	CINT	Method	Increases
	Fan-in (Henry and Kafura, 1981)	FANIN	Method	Increases
	Fan-out (Henry and Kafura, 1981)	FANOUT	Method	Increases
Inheritance	Base Classes (Destefanis et al., 2014)	IFANIN	Class	Decreases
	Depth of Inheritance Tree (Chidamber and Kemerer, 1994)	DIT	Class	Decreases
	Number Of Children (Chidamber and Kemerer, 1994)	NOC	Class	Decreases
	Override Ratio (Lanza and Marinescu, 2006)	OR	Class	Increases
Size	Classes (Lorenz and Kidd, 1994)	CDL	File	Increases
	Instance Methods (Lorenz and Kidd, 1994)	NIM	Class	Increases
	Instance Variables (Lorenz and Kidd, 1994)	NIV	Class	Increases
	Lines of Code (Lorenz and Kidd, 1994)	LOC	Method	Increases
	Lines with Comments (Lorenz and Kidd, 1994)	CLOC	Method	Decreases
	Number of Public Attributes (Lanza and Marinescu, 2006)	NOPA	Class	Increases
	Statements (Lorenz and Kidd, 1994)	STMTC	Method	Increases
	Weight of a Class (Lanza and Marinescu, 2006)	WOC	Class	Increases

Section 3.2 defines two cases for computing critical attributes. **Case 1** occurs when a metric typically becomes critical when its value decreases (Table 2.1). In this case, the internal attribute captured through this metric is critical when the metric value is *below a lower threshold*. **Case 2** occurs when a metric typically becomes critical when its value increases (Table 2.1). In this case, the internal attribute captured through this metric is critical when the metric value is *above an upper threshold*. We relied on our previous work (Chávez et al., 2017) to set the first (25%) and third (75%) quartiles as our lower and upper thresholds, respectively, based on the distribution analysis of each metric.

2.2 Design Smells

Design smells are symptoms of recurring code structures that are potentially harmful to software evolution (Fowler, 2018). Each design smell varies in type according to the recurring code structure it represents (Fowler, 2018; Lanza and Marinescu, 2006). Examples of design smell types often assessed by past work (Bibiano et al., 2019; Fernandes et al., 2017b; Liu et al., 2011; Mantyla et al., 2004) are Large Class and Long Method. Large Class consists of a class that is too large and complex, often realizing too many features. Similarly, Long Method is a too long and complex method. Both types indicate code elements whose understanding and changing may be hard for developers to perform (Palomba et al., 2014; Yamashita and Moonen, 2013).

Table 2.2 list six design smell types (second column) grouped by their granularity (first column), i.e. where each design smell occurs within the code structure of a system. We sampled these types, and adapted their definitions (third column), from Fowler's Refactoring book (Fowler, 2018). We cherry-picked design smell types that are closely related to our results reported in Chapter 4 – on how much and why developers perceive each critical attributes as relevant for evolving features. Similar to the case of refactoring types (Section 1.3), Fowler's book is a very comprehensive catalog of design smell types.

Detecting design smells often requires the measurement of one or more metrics, which are combined in the form of detection strategies (Fernandes et al., 2016b). Once each metric captures a particular critical attribute of the system (Lanza and Marinescu, 2006), design smells are more comprehensive degradation symptoms than critical attributes (Bibiano et al., 2019). The detection strategy is responsible comparing metric values to reference values (Vale et al., 2018) in clauses via logical

Table 2.2: Design Smell Types Grouped by Granularity

Granularity	**Design Smell Type**	**Definition**
Class	Large Class	Too large and complex class
	Data Class	Nothing but a data holder; the lack of a more intricate logic does not pay off its existence
Method	Long Method	Too long and complex method
	Feature Envy	Method that ultimately consumes resources from other classes rather than its host class
Both	Duplicated Code	Same feature implemented in two or more different parts of the source code
	Speculative Generality	Parts of the source code created for an hypothetical use that never really happens

operators, e.g. $>$ and $<$. All clauses are combined via logical connectives, e.g. AND and OR, in order to define whether a code element is affected by design smells.

For illustration purposes, let us consider the Large Class design smell type. Large Class is usually associated with critical attributes such as low cohesion, high complexity, and large size (Fernandes et al., 2017b; Vale et al., 2018). Thus, an arbitrary detection strategy for Large Class could be such that (LCOM2 $> \tau_A$) AND (CC $> \tau_B$) AND (WOC $> \tau_C$), where τ_A, τ_B, and τ_C are considerably high metric values. Another example regards Long Method. Once this design smell type is often associated with a large size, one could define the following detection strategy: (MLOC $> \tau_D$), where τ_D is also a considerably high metric value.

2.3 On the Developer's Perception of Degradation Symptoms

A few previous studies (Fernandes et al., 2017a; Meirelles et al., 2010; Palomba et al., 2014; Taibi et al., 2017; Yamashita and Moonen, 2013) aimed at investigating how developers perceive degradation symptoms. We discuss below some of these studies with an emphasis on how important if mitigating and fully addressing degradation symptoms for the sake of software evolution. We also highlight literature gaps that we address throughout this doctoral thesis.

Interesting insights are reported by previous studies on whether developers perceive anomalous metric values as useful symptoms of degraded code structure and design (Bavota et al., 2013; Fernandes et al., 2017a; Meirelles et al., 2010; Revelle et al., 2011). For instance, a previous work (Meirelles et al., 2010) aimed at characterizing factors leading to a high attractiveness of open source systems to new contributors, i.e., new developers. The authors have found that developers may feel discourage to contributing to systems with anomalous values of complexity metrics. In this particular case, a potentially high difficulty to evolving features leads to a low attractiveness.

Other previous studies (Bavota et al., 2013; Revelle et al., 2011) aimed at capturing the developer's perception on the usefulness of coupling metrics. Results suggest coupling metrics solely based on the analysis of code structures, such as CBO, are less effective than those derived from semantic aspects of the system are. In other words, metrics that consider where features are located within a system may benefit the analysis of internal attributes such as coupling. Finally, another study (Fernandes et al., 2017a) concludes that developers often perceive anomalous metric values for complexity and size as indicators of unclear code. Shortly, unclear code is a part of the source code structure that is potentially hard to understand and change (Fernandes et al., 2017a).

As discussed in Section 2.1, each metric aims at capturing a particular internal attribute (see Table 2.1 for details). Thus, one could assume the aforementioned study results provide hints on how relevant the critical attributes are for software evolution in industry. Still, to the best of our knowledge, no past work has investigated how much critical attribute – e.g., low cohesion, high complexity, high coupling, large inheritance, and large size – are relevant for developers while evolving features. We introduce a quantitative study aimed at addressing this literature gap in Chapter 4.

Similarly, previous studies (Palomba et al., 2014; Taibi et al., 2017; Yamashita and Moonen, 2013) investigated whether developers perceive design smells as useful symptoms of degraded code and design. Again, insights on critical attributes appear because the definition of many design smells depend on one or more critical attributes combined (Section 2.2). Contrary to expectations, only a few design smell types are potential threats to evolving features. For instance, results suggest that design smell types such as Duplicated Code may hinder code reuse (Taibi et al., 2017).

In addition, previous studies also agree that Large Class and Long Method make it hard for developers to understand and change code elements (Palomba et al., 2014; Taibi et al., 2017). Thus, design smell types like these represent threats to evolving features from the developer's perception. This result is particularly interesting because both design smell types are often detected by combining low cohesion, high complexity, and large size (Fernandes et al., 2017b; Vale et al., 2018). On the other hand, Lazy Class (Fowler, 2018), Long Parameter List (Fowler, 2018), and other design smell types are not harmful to software evolution after all (Palomba et al., 2014; Taibi et al., 2017; Yamashita and Moonen, 2013).

Our quantitative study introduced in Chapter 4 also explored the circumstances that make developers either mitigate or fully address critical attributes while evolving features. By doing that, we expect to understand each critical attribute in isolation rather than combined in design problems.

2.4 Refactorings and Re-refactorings

Refactoring means applying changes to improve the internal software quality (Fowler, 2018; Liu et al., 2011). Developers may purely intend to enhance code structures while refactoring, or they may refactor as a means to achieve other intents, such as adding or enhancing features (Fernandes, 2019a; Kim et al., 2014; Paixao et al., 2019; Silva et al., 2016a). Refactorings have been recommended to help manage different types of degradation symptoms, such as anomalous metric values (Bavota et al., 2015; Chaparro et al., 2014) and design smells (Bibiano et al., 2019; Szőke et al., 2015a). We discuss below three basic concepts associated with refactorings: refactoring tactics (Murphy-Hill et al., 2012), refactoring types (Fowler, 2018), and re-refactorings (Chávez et al., 2017).

Developers typically apply two different refactoring tactics: root-canal and floss refactoring (Murphy-Hill et al., 2012). While root-canal refactoring is the pure application of refactorings to code elements, floss refactoring means applying other changes in conjunction with refactorings. These other changes may range from fixing bugs to adding or enhancing features (Fernandes et al., 2020; Fernandes, 2019a; Murphy-Hill et al., 2012; Paixao et al., 2019). In this case, developers – consciously or not – enhance the code structure while applying other changes that realize particular developer intents. For instance, the developer adds new lines of code into an extracted method to implement new features.

In summary, root-canal refactoring means improving the structural quality of the code without further changes, while floss refactoring implies refactoring in conjunction with other changes. From the perspective of critical attributes, one could assume that different refactoring tactics have a different effect on each critical attributes in isolation. For instance, changes applied in conjunction with refactorings could lead to the worsening of internal attributes that a pure refactoring would improve. That assumption is often true and our quantitative study reported in Chapter 3 provides insights on this matter.

Each refactoring has a type targeting the enhancement of a particular code structure. Table 2.3 lists the set of 11 refactoring types analyzed in this doctoral thesis. We defined this set from Fowler's Refactoring book (Fowler, 2018). All refactoring types (second column) are grouped by granularity (first column), i.e. the scope of changes caused by each refactoring type. Examples of popular refactoring types in industry (Bibiano et al., 2019; Murphy-Hill et al., 2012; Silva et al., 2016a) are Extract Method, i.e. extracting a new method from an existing one, and Move Method, i.e. moving an existing method from one class to another class.

Table 2.3: Refactoring Types Grouped by Granularity

Granularity	**Refactoring Type**	**Definition**
Attribute	Move Attribute	Move an attribute from one class to another
	Pull Up Attribute	Move an attribute from one or more subclasses to the superclass
	Push Down Attribute	Move an attribute from the superclass to one or more subclasses
Class	Extract Interface	Extract a new general interface from two or more existing classes
	Extract Superclass	Extract a new superclass from two or more existing classes
Method	Extract Method	Extract a new method from an existing method
	Inline Method	Incorporate the body of an existing method into another method
	Move Method	Move an existing method across classes
	Pull Up Method	Move a method from one or more subclasses to the superclass
	Push Down Method	Move a method from the superclass to one or more subclasses
	Rename Method	Rename an existing method

The refactoring types are implicitly associated with one or more internal attribute. This is because each refactoring type should modify the code structure and its design in such a way it changes one or more metric values. Table 2.4 presents an attempt to associate the five internal attributes analyzed in this doctoral thesis (Section 2.1) with the full set of 11 refactoring types (Table 2.3). We derived this table for the analysis purposes of Chapter 3, based on our experience with software development in industry and insights provided by previous studies (Bibiano et al., 2019; Fowler, 2018).

Table 2.4: Internal Attributes Associated with Refactoring Types

Attribute	**Associated Refactoring Types**
Cohesion	Move Attribute, Move Method
Complexity	Move Method
Coupling	Extract Method, Inline Method, Move Attribute, Move Method
Inheritance	Extract Interface, Extract Superclass, Move Attribute, Move Method, Push Down Attribute, Push Down Method
Size	Extract Interface, Extract Method, Extract Superclass, Inline Method, Pull Up Attribute, Pull Up Method, Rename Method

As an example, the class cohesion may improve through Move Attribute and Move Method if the refactored code element (attribute or method) was at least partially the root-cause of the low cohesion. We highlight that this association takes into account the scope of each internal attribute as described in Section 2.1. Consequently, some reasonable associations are missing, such as complexity versus Extract Method. In this case, once we target the class complexity rather than the method complexity, we assume that Extract Method has not direct effect on the class complexity.

Re-refactoring: Refactorings are not restricted to the application of each single refactoring in isolation. Re-refactoring occurs whenever refactorings are applied to a code element refactored any time in the past (Chávez et al., 2017; Jiau et al., 2013). This phenomenon is quite frequent in practice (Bibiano et al., 2019; Chávez et al., 2017; Murphy-Hill et al., 2012). Re-refactoring is supposedly applied to fully achieve one of the aforementioned developer intents with refactorings, e.g. purely enhancing the code structure or evolving features (Fernandes, 2019a; Paixao et al., 2019).

Re-refactoring as a theoretical concept is still evolving in its formal definition and mechanisms for computing re-refactoring instances. This may have been due to the lack of studies, until recently (Bibiano et al., 2019; Chávez et al., 2017; Fernandes et al., 2019b; Murphy-Hill et al., 2012), on understanding how developers combine different refactorings for enhancing the code structure and its design. In this work, decide for choosing the re-refactoring definition proposed by a recent work (Chávez, 2017) Although this past definition is quite loose, we explain below how each refactoring is considered as either a re-refactoring instance or a single refactoring.

Consider the three granularities of refactoring types in the first column of Table 2.3: attribute, method, and class. Consider also that developers applied a refactoring `r` (e.g. Move Method) to a given code element (e.g., a method `m()`) at a certain granularity (in this case, at the method granularity). Thus, `r` is a re-refactoring instance only if developers previously applied another refactoring `r'` (e.g., another Move Method or an Extract Method etc.) to the same code element (in this case, `m()`) at the same granularity. That is, for a refactoring `r` to be a re-refactoring instance, another refactoring `r'` must have occurred previously during software evolution and share the same granularity.

Composite refactoring: There is empirical evidence that about a half of the tool-automated refactorings are applied in conjunction rather than in isolation (Murphy-Hill et al., 2012). Furthermore, developers can combine two or more refactorings to achieve their most varied intents (Silva et al., 2016a), e.g. to mitigate or fully address a degradation symptom (Bibiano et al., 2019; Fowler, 2018; Lin et al., 2016; Szőke et al., 2015a). Composite refactoring, formerly known as batch refactoring, is how we have been calling the application of two or more refactorings in conjunction (Fernandes et al., 2019b; Fernandes, 2019a). Although frequent in real systems, composite refactoring has only recently being empirically assessed (Bibiano et al., 2019; Bibiano et al., 2020).

2.5 On Addressing Degradation Symptoms through Refactoring

Some studies, e.g. (Bavota et al., 2015; Bibiano et al., 2019; Chaparro et al., 2014) investigated the role of refactoring in either mitigating or fully addressing degradation symptoms while evolving features. These studies targeted different degradation symptoms, including anomalous metric values (Bavota et al., 2015; Chaparro et al., 2014; Veerappa and Harrison, 2013) and design smells (Bibiano et al., 2019; Yoshida et al., 2016).

Although refactoring was originally assumed to enhance the code structure and its design (Fowler, 2018), thereby leveraging the internal software quality, past work does not support this assumption. For instance, a non-ignorable percentage of refactorings worsens, rather than improves, anomalous metric values (Chaparro et al., 2014; Du Bois and Mens, 2003). Furthermore, the literature (Bibiano et al., 2019; Bibiano et al., 2020; Tufano et al., 2017) suggests that refactoring rarely suffices to fully remove design smells.

Applying each single refactoring to the code structure only removes from 9% to 9.7% of design smells (Tufano et al., 2017). This results is not exactly unexpected, once design smell types such as Duplicated Code and Large Class (cf. Table 2.2) involve multiple code elements and are quite hard to fully address. On the other hand, applying composite refactorings also falls short in removing design smells (Bibiano et al., 2019; Bibiano et al., 2020) – even when applying composite refactorings suggested in Fowler's Refactoring book (Fowler, 2018). Composites remove only 11% of design smells (Bibiano et al., 2019), probably due to the lack of assistance in combining refactorings (Bibiano et al., 2020; Fernandes et al., 2019b).

Several techniques aim at assisting developers in mitigating or fully addressing degradation symptoms in real settings. Many tools aim at automating the detection of design smells (Fernandes et al., 2016b). Systematic procedures were introduced for driving the validation of degradation symptoms that really concern developers while evolving features (Liu et al., 2011; Oliveira et al., 2020b). Finally, tools were designed for removing design smells from systems (Lin et al., 2016; Szőke et al., 2015a; Tsantalis et al., 2018).

To the best of our knowledge, none of the existing tools and techniques mentioned above have been shaped for addressing particular needs of adding or enhancing features. We acknowledge that the current tooling support may be quite handful while performing refactorings, especially for preventing an overall decay of the internal software quality. Still, existing tools and techniques typically ignores that certain degradation symptoms – e.g., specific critical attributes – are more relevant than others are while evolving features.

Aimed at partially addressing this literature gap, our qualitative study reported in Chapter 3 summarizes a preliminary set of refactoring recommendations. Our major goals are i) assist developers with refactorings to help manage critical attributes, thereby supporting software evolution, and ii) inspire the design of novel tooling support for assisted refactorings.

2.6 Chapter Summary

This chapter summarized key theoretical concepts explored through the doctoral thesis. We discussed two of the most recurrently investigated degradation symptoms in the literature: critical attributes and design smells. As a complement, we summarized findings of previous studies on the developer's perception of degradation symptoms – with an emphasis on the lack of knowledge of relevant critical attributes. Also in this chapter, we discussed some basic concepts regarding refactorings, including refactoring tactics and types. Finally, we summarized the existing strategies for either mitigating or fully addressing degradation symptoms through refactorings – with an emphasis on the lack of refactoring support to help manage critical attributes.

In order to dig deeper into the degradation symptoms to evolving features, the next chapter introduces the first study that composes this doctoral thesis. In a large quantitative study, we investigate the relationship between refactoring and internal attributes. We are particularly concerned on addressing major literature limitations, such as the lack of empirical studies on re-refactoring.

3
Relationship between Refactorings and Internal Attributes

Several software changes are daily applied along with software evolution (Lehman, 1980). Each change addresses a particular demand for changing software features or operating environment constraints (Mens et al., 2010). Refactoring is typically applied to enhance the code structure and its design (Kim et al., 2014; Murphy-Hill et al., 2012; Silva et al., 2016a). In principle, each single refactoring targets the improvement of a particular code structure (Fowler, 2018; Kataoka et al., 2002; Liu et al., 2011). Mitigating or fully addressing a degraded code structure often requires the application of two or more refactorings on the same code element (Bibiano et al., 2019; Murphy-Hill et al., 2012). We refer to this phenomenon as re-refactoring (Fernandes et al., 2020; Jiau et al., 2013).

There are two refactoring tactics (Murphy-Hill et al., 2012): root-canal refactoring and floss refactoring. Developers apply root-canal refactoring when they aim at exclusively improving the internal software quality. Thus, changes applied along with root-canal refactoring are restricted to refactorings *à la* Fowler (Fowler, 2018). Developers apply floss refactoring aimed at achieving other intents rather than the pure code structure and design enhancement, e.g. evolving features (Fernandes, 2019a; Murphy-Hill et al., 2012). Thus, developers (consciously or not) enhance the code structure and its design in order to enable a feature addition or enhancement. This observation could partially explain why about 92% of the code review issues target refactorings together with either feature addition or enhancement (Paixao et al., 2019).

Internal quality attributes target internal quality properties of a system. Cohesion and complexity are examples of these attributes (Chidamber and Kemerer, 1994; Lanza and Marinescu, 2006; Lorenz and Kidd, 1994). One could expect a different effect on internal attributes after applying each refactoring type. For instance, Move Method could reduce coupling of the methods' source class, while Extract Method reduces method size (Fowler, 2018). Analyzing the refactoring effect on internal attributes could drive the enhancement of code structures while facilitating other intents, such as evolving features. One could also expect a different effect on attributes per refactoring tactic. For instance, floss refactoring may be more likely to worsen attributes due to the changes applied in conjunction with refactorings.

A few studies (Bavota et al., 2015; Yoshida et al., 2016) have investigated the refactoring effect on degraded code structures spotted by design smells (Fowler, 2018). These studies suggest that design smells potentially harm software evolution, once developers apply several refactorings on smelly code while evolving software features. Contrary to expectations, these studies show that single refactorings only remove from 9% to 9.7% of design smells. Design smells are coarse-grained degradation symptoms, once each design smell is often captured from two or more critical internal attributes (Fowler, 2018; Lanza and Marinescu, 2006). By targeting design smells only, past work may have overlooked a fine-grained improvement of particular critical attributes (Chávez et al., 2017), even if design smells were not removed.

It is assumed that re-refactoring may help addressing more complex degradation symptoms when compared to each single refactoring (Chávez et al., 2017; Jiau et al., 2013). Unfortunately, the current knowledge on the relationship between refactorings and internal attributes is insufficient to stand this assumption. Indeed, previous studies (Bavota et al., 2015; Chaparro et al., 2014; Du Bois and Mens, 2003; Veerappa and Harrison, 2013) assess a strict scope of internal attributes. While most studies target cohesion and coupling only, there are other potentially relevant attributes, e.g. complexity and size. Additionally, those studies analyzed only a few metrics per attribute. Finally, there is still no evidence that re-refactoring is more frequent in code elements affected by complex degradation symptoms, i.e. two or more attributes combined, when compared to single refactoring.

Critical attributes are quantifiable via software metrics as much as design smells. Nevertheless, we acknowledge the importance of investigating critical attributes in specific. Indeed, degradation symptoms are often interrelated (Fernandes et al., 2017b). For instance, critical attributes targeting the code structure and its design, such as low cohesion and high coupling, are closely associated with architecture degradation (Samarthyam et al., 2016). Thus, understanding how refactorings affect each internal attribute may leverage software evolution practices.

This chapter extends a recent work (Chávez et al., 2017) on the relationship between refactorings and internal attributes. Our study was originally designed to investigate five internal attributes (Fowler, 2018; Lanza and Marinescu, 2006; Lorenz and Kidd, 1994): cohesion, complexity, coupling, inheritance, and size. We mined 23 Java open source systems with 29,303 refactorings in total. We analyzed 11 popular refactoring types (Kim et al., 2014; Murphy-Hill et al., 2012).

In this chapter, we incorporate complementary study findings grounded on statistical testing, in order to address a major literature limitation reported by a recent work (Al Dallal and Abdin, 2017). Moreover, we introduce an unprecedented analysis of the re-refactoring effect on internal attributes, thereby addressing another major literature gap. Finally, we systematically compare the results obtained for refactorings in general with those achieved for re-refactorings.

Similar to our previous work (Chávez et al., 2017), we defined critical attribute as an attribute whose metrics used for capturing it assume anomalous values (Vale et al., 2018). We aimed to reveal if re-refactorings may at least partially address critical attributes, as refactorings in general (Chávez et al., 2017). Hereinafter, we refer to both refactoring and re-refactoring as *(re-)refactoring*. Our results suggest that:

– Surprisingly, 94% of (re-)refactorings occur in code elements with at least one critical attribute. This result contradicts a previous study (Bavota et al., 2015) by showing that refactorings and internal attributes are indeed interrelated. Such a high refactoring rate also indicates that critical attributes are possibly relevant for evolving features.

– 73% of refactorings constitute floss refactorings, i.e. refactorings applied in conjunction with other changes often intended at evolving features. Combined with the aforementioned finding, this result suggests that mitigating or fully addressing critical attributes via refactorings is important in practice. Otherwise, developers would not refactor code while performing feature additions and enhancements.

– Refactorings tend to either improve or keep unaffected the internal attributes, regardless of the refactoring types being associated with a particular attribute. In 64% of the cases, attributes were improved or kept unaffected when considering our strictest analysis approach. Nevertheless, it is possible that i) refactorings in general are insufficient to overcome degraded code structure and design spotted by critical attributes, or ii) developers only fully address those critical attributes that really concern developers while evolving features. Our qualitative study reported in Chapter 4 further investigates these assumptions.

– Re-refactorings tend to improve or keep unaffected the internal attributes, just as refactorings in general. However, the rates of attribute improvement are slightly greater than the rates of attribute worsening: up to 26% greater in the case of pure refactorings that changed attributes they should change. This result suggests re-refactoring aims at gradually addressing complex degradation symptoms, e.g. combinations of two or more critical attributes in the form of design smells.

We organized the remainder of this chapter as follows. Section 3.1 introduces the study goal and research questions. Section 3.2 describes the study steps and procedures. Section 3.3 presents the study results on the relationship between (re-)refactorings and critical attributes. Sections 3.4 and 3.5 provide the study results on the refactoring and re-refactoring effect, respectively. Section 3.6 compares the refactoring and re-refactoring effect results. Section 3.7 compares our study with past work at the levels of study design and results. Section 3.8 discusses threats to the study validity. Finally, Section 3.9 concludes this chapter and introduces the next one.

3.1 Goal and Research Questions

Our study goal is: *analyze* how refactoring in general and re-refactoring affect internal attributes; *for the purpose of* understanding when and how (re-)refactoring affect metrics that quantify cohesion, complexity, coupling, inheritance, and size; *with respect to* i) the frequency of (re-)refactorings applied to code elements with critical attributes, and ii) if these refactorings are more likely to improve, keep unaffected, or worsen internal attributes; *from the viewpoint of* software engineering researchers; *in the context of* Java open source systems and refactoring types popularly adopted in industry. We describe below our four research questions (RQs).

RQ$_1$: *Is (re-)refactoring often applied to code elements with critical attributes?* – Anomalous metric values are shown useful for spotting code elements that are worth refactoring (Fowler, 2018; Lanza and Marinescu, 2006; Lorenz and Kidd, 1994). In our past work (Chávez et al., 2017), we investigated if developers are more likely to apply refactorings to code elements whose anomalous metric values suggest the occurrence of critical attributes.

Through RQ$_1$, we extend our past work by investigating the application of both refactoring in general and re-refactoring. We refer to critical attributes those internal attributes whose metric values are anomalous when compared to a reference value. If code elements with critical attributes are typically (re-)refactored, it is worth investigating the (re-)refactoring effect on internal attributes. Section 3.2 describes our strategy for computing critical attributes.

From the perspective of software evolution, RQ$_1$ is essential to validate the practical relevance of critical attributes while evolving features. If developers often perform (re-)refactorings on code elements with at least one critical attribute, the degraded code structure and design realized by these code elements are worth mitigating or fully addressing. Past work (Bibiano et al., 2019; Chávez et al., 2017) showed that floss refactorings, i.e. refactorings combined with intents such as evolving features (Murphy-Hill et al., 2012), are more frequent than root-canal refactorings. Thus, the more critical attributes affect refactored code, the higher may be the relevance of critical attributes to either adding or enhancing features.

RQ$_2$: *How does refactoring affect internal attributes?* – Once we know how frequently developers apply refactorings to code elements with critical attributes (RQ$_1$), it becomes important to observe the side effect of these refactorings to the code structural quality. Via RQ$_2$, we investigate the refactoring effect on the five internal attributes selected for the study (Section 2.1). Our major goal is to understand if refactorings are more likely to improve, keep unaffected, or worsen these internal attributes, regardless of being critical.

Table 2.4 suggests that only a few internal attributes have a close relationship with each refactoring type. Thus, our study makes a clear distinction of associated and non-associated refactoring types by attribute. We also define two approaches for analyzing the refactoring effect on internal attributes: At Least One Metric and Most Metrics. Please refer to Section 3.2 for further information on each analysis approach.

In the RQ$_2$ case, analyzing the refactoring effect on internal attributes may help in understanding the importance of mitigating and fully addressing these degradation symptoms. If (re-)refactorings often worsen or keep unaffected the internal attributes, we could state that (re-)refactorings have been insufficient for fully addressing internal attributes. Thus, developers need assistance while adding or enhancing features, thereby preventing that internal attributes become critical. However, it is possible that refactorings and re-refactorings often improve internal attributes. Thus, we could conclude that these changes, even when combined with evolving features, are effective in mitigating or fully addressing internal attributes, including critical attributes.

RQ$_3$: *How does re-refactoring affect internal attributes?* – Developers may have to apply more than one refactoring to the same code element, e.g. to mitigate the harmfulness of a degradation symptom to software evolution. This phenomenon is called re-refactoring (Jiau et al., 2013). Contrary to refactorings in general (Chávez et al., 2017), i.e., regardless of constituting re-refactorings or not, little is known about the re-refactoring effect on internal attributes.

It may be true that, although developers typically perform re-refactoring, the internal software quality worsens rather than improves. If so, developers need guidance while applying re-refactorings. RQ$_3$ aims at revealing the extent in which re-refactoring has succeeded in improving code structures as assumed by previous work (Jiau et al., 2013). Section 3.2 describes two approaches for analyzing the re-refactoring effect on internal attributes.

Through RQ$_3$, we may partially address the aforementioned assumption that re-refactorings aim at mitigating or fully addressing complex degradation symptoms while evolving features. It is possible that re-refactorings very often improve internal attributes. Thus, developers may have been re-refactoring for gradually addressing critical attributes.

RQ$_4$: *To what extent the refactoring effect differs from the re-refactoring effect on internal attributes?* – In RQ$_2$ and RQ$_3$, we explore the effect of refactoring in general and re-refactoring on internal attributes separately. Thus, it may be hard to understand whether the effect of applying new refactorings to a previously refactored code element increases (or decreases) the likelihood of improving, keeping unaffected, or worsening internal attributes. RQ$_4$ aims at fairly comparing the results obtained by the separate analyses of refactoring in general and re-refactoring.

Maybe developers tend to worsen, rather than improve, internal attributes while performing re-refactorings. This result could serve as a warning for preventing an unexpected introduction of critical attributes along with re-refactorings. From a software evolution perspective, RQ$_4$ may validate our assumption that re-refactorings are more effective in addressing complex degradation symptoms when compared to refactorings in general.

3.2 Steps and Procedures

Step 1: *Selecting systems* – Table 3.1 lists the 23 analyzed systems sorted by the number of refactorings detected (third column). We relied on the same data set explored by our previous work (Chávez et al., 2017), retrieved from GitHub. The number of systems by number of refactorings is balanced: six systems have many refactorings ($> 1,000$) and six systems have less than 100 refactorings; the other 11 systems range from 100 to 1,000 refactorings each. Our system selection criteria were: Java as the main programming language, because Java is largely adopted and allows the detection of most refactoring types defined in Fowler's book (Fowler, 2018); open source systems to support the study replication; and popular systems based on the GitHub stars count.

Table 3.1: Systems Selected for Analysis

System	GitHub Repository	Refactorings	Commits
Elasticsearch	elastic/elasticsearch	9,507	23,597
Spring Framework	spring-projects/spring-framework	5,320	12,974
ArgoUML	argouml-tigris-org/argouml	2,588	17,654
Presto	prestodb/presto	2,068	8,056
Ant	apache/ant	2,063	13,331
Realm	realm/realm-java	1,699	5,916
Spring Boot	spring-projects/spring-boot	1,386	8,529
OkHttp	square/okhttp	855	2,645
Xerces2 Java	apache/xerces2-j	853	5,456
J2ObjC	google/j2objc	713	2,823
MPAndroidChart	PhilJay/MPAndroidChart	398	1,737
Facebook SDK for Android	facebook/facebook-android-sdk	341	601
Junit 4	junit-team/junit4	309	2,113
Dubbo	alibaba/dubbo	280	1,836
Hystrix	Netflix/Hystrix	266	1,847
Retrofit	square/retrofit	232	1,349
Fresco	facebook/fresco	161	744
Dagger 1	square/dagger	96	696
Google I/O Android App	google/iosched	73	129
Simian Army	Netflix/SimianArmy	55	710
Logger	orhanobut/logger	20	68
LeakCanary	square/leakcanary	12	265
Android Bootstrap	AndroidBootstrap/android-bootstrap	8	230
Total		**29,303**	**113,306**

Step 2: *Computing refactorings* – We adopted RefactoringMiner (Tsantalis et al., 2013) for detecting the 11 refactoring types analyzed in this work and listed in Table 2.3. This tool has presented a high accuracy of 95% of precision and recall in the detection of refactorings (Chávez et al., 2017; Tsantalis et al., 2013). We chose Version 0.2.0, retrieved online[1], due to an existing validation of detection results (Tsantalis et al., 2013). The 11 refactoring types covered by RefactoringMiner have been shown popular in industry (Murphy-Hill et al., 2012; Silva et al., 2016a). We detected 29,303 refactorings in total.

Step 3: *Classifying refactorings by tactic* – We manually classified a random set of 2,119 refactorings, i.e., about 7% out of the total, by refactoring tactic. We arbitrarily chose the set size for enabling its manual classification by three researchers. We annotated each refactoring regardless of constituting a single refactoring or a re-refactoring. If we observed that a refactoring was followed by refactoring-unrelated changes, e.g., the addition of new code statements, we annotated the refactoring as floss refactoring; otherwise, we annotated the refactoring as root-canal refactoring. We obtained 1,543 floss refactorings against 576 root-canal refactorings. We reused the same data set for analyzing the re-refactoring effect on internal attributes.

Step 4: *Computing metric values by internal attribute* – We used the non-commercial license of the Understand tool[2] for detecting the 25 metrics that capture internal attributes listed in Table 2.1. We have successfully

[1]https://github.com/tsantalis/RefactoringMiner
[2]https://scitools.com/features/

adopted this tool to detect metrics in large data sets (Bibiano et al., 2019; Chávez et al., 2017). Understand is compatible with various languages including Java, thereby allowing us to compute metrics for the 23 systems under analysis.

Procedure 1: *Computing critical attributes* – We computed the criticality of attributes for each (re-)refactored code element based on metrics. A metric may indicate a critical attribute in two scenarios. Scenario 1: if the metric typically becomes critical when its value decreases, then a critical attribute is characterized by values *below a lower threshold*. Scenario 2: if the metric typically becomes critical when its value increases, then a critical attribute is characterized by values *above an upper threshold*.

Table 2.1 discriminates the typical scenario for each analyzed metric. Thresholds can be computed through the distribution of a metric, especially quartiles that segregate highest and lowest values in a distribution (Vale et al., 2018). Inspired by our previous work (Chávez et al., 2017), we used the first (25%) and third (75%) quartiles as the lower and upper thresholds. We reused the criticality computation algorithm described in our past work (Chávez et al., 2017).

Procedure 2: *Computing the relationship of (re-)refactoring and critical attributes* – With Procedure 1, we computed the frequency of (re-)refactorings affecting code elements with zero or more critical attributes. To understand the differences among refactoring in general and re-refactoring, we computed the Fisher's exact test with a 99% confidence interval, i.e., p-value < 0.01. The test was employed to compute the relationship of (re-)refactorings and the number of critical attributes affecting the (re-)refactored code element (Fernandes et al., 2017b). Fisher's test computes the probability of a certain property – i.e., the application of either a single refactoring or a re-refactoring – to co-occur with another property – i.e., the affected code element to have either zero or more than one critical attribute. The test results in the Odds Ratio that indicates how many times the first property co-occurs with the second one (Fernandes et al., 2017b).

Procedure 3: *Computing the (re-)refactoring effect on internal attributes* – We define two analysis approaches for computing the behavior of internal attributes after (re-)refactorings. In the *At Least One Metric approach*, we assume that an internal attribute has improved if one or more associated metrics have improved after the (re-)refactoring application. Therefore, improving this attribute was at least a little relevant for evolving features. This is the less strict analysis approach because it considers any improvement, which is useful to capture the refactoring effect at a very fine-grained level.

In the *Most Metrics approach*, we assume that an internal attribute has improved if most of the associated metrics, i.e., 50% of metrics plus one metric, have improved. Therefore, improving this attribute was probably very relevant for evolving features. This is the strictest analysis approach, once the (re-)refactoring has to be significantly positive by means of the number of metrics that quantify an internal attribute. For attributes with many metrics, e.g., size, the improvement means a highly significant improvement of metrics.

Procedure 4: *Comparing the (re-)refactoring effect per analysis approach* – In order to compare the results obtained for each analysis approach defined in Procedure 3, we have applied a correlation analysis. The last column presents the ρ coefficient computed from the Pearson's correlation test for the results by type. ρ indicates the correlation between Most and At Least One frequencies. Aimed at a fair correlation computation, we computed correlation only for those attributes whose effect remained the same. We adopted the following criteria (Hinkle et al., 2002) to classify each ρ coefficient by the correlation strength: very strong ($\rho \geq 0.9$), strong ($0.7 \leq \rho < 0.9$), moderate ($0.5 \leq \rho < 0.7$), weak ($0.3 \leq \rho < 0.5$), and very weak ($\rho < 0.3$).

3.3 (Re-)Refactoring and Critical Attributes (RQ$_1$)

3.3.1 Frequency Regardless of Refactoring Type

Table 3.2 presents the frequency of (re-)refactorings applied to code elements with critical attributes. The first column lists three sets of refactorings. The *all refactorings* row includes the 29,303 refactorings detected for all 23 systems. The *single refactoring* row covers those transformations that do not constitute re-refactorings. The *re-refactoring* row is composed of all re-refactorings. Finally, the *random sample* rows include the 2,119 refactorings manually classified by refactoring tactic (Step 3 of Section 3.2).

Table 3.2: Frequency of Refactorings Applied to Elements with Critical Attributes

			Nº of Critical Attributes					
Refactoring Set	**Tactic**	**Total**	**No**		**Single**		**Multiple**	
			Abs.	**%**	**Abs.**	**%**	**Abs.**	**%**
All refactorings	All tactics	29,303	1,570	5.36	4,458	15.21	23,275	79.43
Single refactoring	All tactics	14,782	1,550	10.49	4,425	29.94	8,807	59.58
Re-refactoring	All tactics	14,521	20	0.14	33	0.23	14,468	99.64
Random sample	Root-canal	576	7	1.22	17	2.95	552	95.83
	Floss	1,543	4	0.26	13	0.84	1,526	98.90

Note: Percentage values may not total 100.00 due to the two-decimals rounding

The second column of Table 3.2 lists the refactoring tactics. The third column presents the total of refactorings by tactic. The remaining columns inform the frequency of refactorings applied to code elements that have: zero critical attributes, i.e., the *no* category; one critical attribute, i.e., the *single* category; and more than one critical attribute, i.e., the *multiple* category. Frequency values are given in absolute number (Abs.) and percentage (%).

Developers apply most (re-)refactorings to code elements with at least one critical attribute. By summing the two last categories, i.e., single and multiple, we have at least 94% of refactorings regardless of being either a single refactoring or a re-refactoring. There is a particular novelty of our results here in comparison with our previous work (Chávez et al., 2017), whose data are shown in Table 3.2, except for the fifth row. This time we have found that, even if a refactoring has been applied to a given code element in the past, this code element tends to remain with at least one critical attribute after the re-refactoring application.

To compare the results for single refactoring and re-refactoring, we computed the Fisher's exact test on our data with a 99% confidence interval. The test resulted in a statistically significant Odds Ratio equals 84.90. In other words, single refactorings are about 85 times more likely to be applied to code elements with zero critical attributes when compared to re-refactorings. This result suggests that developers decide to re-refactor only those degraded code structures that actually threaten software evolution.

Regarding refactoring tactics, floss refactoring is slightly more frequent in code elements with two or more critical attributes (98.90%) when compared to root-canal refactoring (95.83%). This result is particularly interesting because developers do not necessarily apply floss refactoring with code structure improvement in mind. Thus, although the intent behind changes extrapolates the pure enhancement of code structures, developers should be aware of the refactoring effect on internal attributes. Otherwise, developers may unexpectedly degrade the code structure and its design.

On the Relevance of Critical Attributes for Evolving Features: Surprisingly, about 94% of (re-)refactorings occur in code elements with at least one critical attribute. Such a high refactoring rate, summed up with a higher incidence of floss refactoring (73%) when compared to root-canal refactoring (27%), leads us to an interesting conclusion. That is, addressing critical attributes seems very relevant for evolving features. Otherwise, developers would not have spent such an effort with refactoring code elements with a degraded code structure or design.

Additionally, we found that single refactoring is up to 85 times more likely to occur in code elements without critical attributes when compared to re-refactoring. Thus, we partially confirm our assumption on the re-refactoring use to mitigate or fully address complex degradation symptoms, e.g., two or more critical attributes co-occurring in a code element.

3.3.2 Frequency by Refactoring Type

Table 3.3 presents the frequency of (re-)refactorings applied to code elements with different numbers of critical attributes, i.e., zero (*no*), one (*single*), or more than one (*multiple*). In this case, we did not split single refactoring and re-refactoring. We present the frequency in absolute number (Abs.) and percentage (%) by refactoring type. The 11 refactoring types listed in the second column are organized by means of the code element that each type typically affects, i.e., class (and interface), attribute, and method. The remaining columns are similar to Table 3.2.

Table 3.3: Frequency of Refactorings Applied to Elements with Critical Attributes per Type

			Nº of Critical Attributes					
Element	**Refactoring Type**	**Total**	**No**		**Single**		**Multiple**	
			Abs.	**%**	**Abs.**	**%**	**Abs.**	**%**
Class	Extract Interface	133	13	9.78	12	9.02	108	81.20
	Extract Superclass	341	16	4.69	10	2.93	315	92.38
Attribute	Move Attribute	4,355	0	0.00	6	0.14	4,349	99.86
	Pull Up Attribute	465	2	0.43	0	0.00	463	99.57
	Push Down Attribute	78	0	0.00	0	0.00	78	100.00
Method	Extract Method	7,513	14	0.19	14	0.19	7,485	99.62
	Inline Method	1,525	1	0.07	1	0.07	1,523	99.86
	Move Method	1,404	4	0.28	10	0.71	1,390	99.01
	Pull Up Method	629	2	0.32	5	0.79	622	98.89
	Push Down Method	114	0	0.00	1	0.88	113	99.12
	Rename Method	12,746	1,518	11.91	4,399	34.51	6,829	53.58

Note: Percentage values may not total 100.00 due to the two-decimals rounding

Data of Table 3.3 supports some key insights on the (re-)refactoring effect by refactoring type. Only Extract Interface, Extract Superclass, and Rename Method were applied more than 1% of the times to code elements without critical attributes. Additionally, for all refactoring types, the observation remains: developers ultimately apply refactorings to code elements with multiple critical attributes rather one or less attributes. Curiously, Rename Method had the highest percentage of refactorings in the *no* category. We are aware that Rename Method does not directly affect the method's structure. Nevertheless, renaming can be a complement to other refactorings previously applied for gradually enhancing a code structure (Bibiano et al., 2019; Bibiano et al., 2020; Fowler, 2018).

Comparison with a Closely Related Work: A previous work (Bavota et al., 2015) explored the relationship of metrics and refactorings applied to open source systems. Similar to our previous work (Chávez et al., 2017), the authors investigated how often refactorings are applied to code elements with anomalous metric values. Contrary to our results, they did not find a clear relationship of metrics and refactorings. Instead, the authors observed that developers typically apply refactorings to code elements whose metrics do not indicate a need for refactoring. Aimed at tracking the root cause for such a different result, we compared the designs of the previous work and ours.

Regarding the refactoring detection tool, we used RefactoringMiner (Tsantalis et al., 2013), while the previous work used Ref-Finder (Prete et al., 2010). The former was empirically validated with high accuracy, i.e., about 95% of average precision (Tsantalis et al., 2013), besides being successfully explored by us in recent studies (Bibiano et al., 2019; Bibiano et al., 2020; Chávez et al., 2017). The latter was also empirically validated with high accuracy, i.e., 79% of overall precision (Prete et al., 2010), but we decided to use the most accurate tool. We hypothesize that the tool choice of the previous work (Bavota et al., 2015) biased the results with too many false positives.

The data sets explored by our work and the previous one are quite different as well. Although both studies analyzed open source systems only, the previous work mined only three systems, i.e., ArgoUML, Ant, and Xerces2 Java, against the 23 systems mined by us, including those three systems. We included 20 more systems to diversify our database by means of the number of refactorings and commits. Furthermore, while our work relied on the full commit history of each system, the previous work analyzed only major system releases. Thus, previous work possibly overlooked refactorings applied in between release-related commits.

At least implicitly, both studies explored the same five internal attributes. Nevertheless, the metric sets differ a lot. We explored 25 metrics against 11 metrics by the previous work. Five metrics, namely CBO, DIT, LOC, NOC, and WMC, were explored by both studies (see Table 2.1 for metric definitions); we replaced the much criticized original Lack of Cohesion (LCOM) with alternative metrics (Chidamber and Kemerer, 1994; Li and Henry, 1993). We hypothesize that our larger metrics set allowed us a more comprehensive understanding of how refactorings and metrics inter-relate from the perspective of internal attributes.

3.3.3
Summary of RQ$_1$

Our quantitative data suggests that 94% of (re-)refactorings are applied to code elements with critical attributes. By contradicting a previous work (Bavota et al., 2015), we confirm that refactorings and internal attributes are indeed interrelated. Such a high rate of refactored code elements with at least one critical attribute suggests that these attributes are relevant degradation symptoms to evolving features. Moreover, 73% of refactorings constitute floss refactorings, i.e. refactorings applied in conjunction with other changes often intended at evolving features. This result suggest that addressing critical attributes is important to the success of feature additions and enhancements. Otherwise, developers would not refactor those code elements while evolving features.

3.4 Refactoring Effect on Internal Attributes (RQ$_2$)

3.4.1 Improvement of Internal Attributes

Table 3.4 summarizes the refactoring effect for each internal attribute (columns) and refactoring type (rows). For facilitating discussion, the refactoring types are grouped by the type of refactored code element: class (and interface), attribute, and method (Table 2.3). The first column lists all 11 refactoring types. The second column discriminates the study results by analysis approach: Most correspond to the Most Metrics approach, while One denotes the At Least One Metric approach. For each attribute, we present the refactoring effect on the attribute in the columns *E* – of *Effect* – and the percentage of refactorings that caused such effect is in the columns labeled with %.

Table 3.4: Refactoring Effect on Internal Attributes by Refactoring Type

Type	Appr.	Cohesion		Complexity		Coupling		Inheritance		Size		ρ
		E	%	E	%	E	%	E	%	E	%	
Extract	Most	–	83.46	–	94.74	–	69.92	↑	**64.66**	–	**42.11**	0.98
Interface	One	–	83.46	–	94.74	–	63.16	↑	**68.42**	–	**39.85**	
Extract	Most	↑	51.32	–	60.12	–	53.67	–	**54.25**	↑	**73.02**	0.99
Superclass	One	↑	51.32	–	60.12	↑	48.68	↑	**92.67**	↑	**81.52**	
Move	Most	↑	**63.31**	–	88.40	–	**55.57**	–	**90.13**	–	50.95	0.99
Attribute	One	↑	**63.31**	–	88.40	–	**49.00**	–	**87.83**	–	48.45	
Pull Up	Most	↓	59.35	–	68.60	↓	66.45	–	81.94	↓	**63.87**	0.89
Attribute	One	↓	59.35	–	68.60	↓	66.45	–	70.97	↑	**81.29**	
Push Down	Most	↓	47.44	–	87.18	↑	41.03	–	**97.44**	↑	46.15	0.82
Attribute	One	↓	47.44	–	87.18	↑	53.85	–	**94.87**	↑	79.49	
Extract	Most	↓	59.03	↑	44.95	↓	**45.77**	–	93.81	↓	**58.53**	1.00
Method	One	↓	59.03	↑	47.03	↑	**71.40**	–	93.09	↑	**85.81**	
Inline	Most	↑	58.30	–	48.92	↓	**39.87**	–	92.92	↑	**56.59**	0.68
Method	One	↑	58.30	–	48.07	↑	**77.64**	–	91.48	↑	**88.98**	
Move	Most	↓	**46.44**	–	**68.87**	↓	**41.17**	–	**81.05**	↑	43.73	0.23
Method	One	↓	**46.44**	–	**68.87**	↑	**48.01**	–	**74.29**	↑	82.55	
Pull Up	Most	↓	43.88	–	69.16	↓	67.89	–	85.06	↓	**52.94**	1.00
Method	One	↓	43.88	–	69.16	↓	67.89	–	83.47	↑	**82.99**	
Push Down	Most	↓	42.11	–	59.65	↑	44.74	–	**89.47**	↓	58.77	0.66
Method	One	↓	42.11	–	59.65	↑	77.19	–	**85.09**	↑	78.07	
Rename	Most	–	100.00	–	97.98	–	82.93	–	100.00	–	**86.11**	1.00
Method	One	–	100.00	–	97.98	–	80.63	–	100.00	–	**85.74**	

E stands for effect: improved (↑), worsened (↓), and kept unaffected (–)

Cells with **bold** font mean that the respective attribute should improve after the application of the respective refactoring type; Table 2.4 presents all the expected associations of attributes and refactoring types. The last column presents the ρ coefficient computed from the Pearson's correlation test for the results by type. ρ denotes the correlation between Most and One frequencies. Aimed at a fair correlation computation, we computed correlation only for those attributes whose effect remained the same. Taking Extract Superclass as an example, cohesion kept unaffected for both Most and One approaches, while coupling had a different effect per analysis approach.

Most Metrics analysis approach. Pull Up Attribute, Pull Up Method, and Rename Method did not improve any of the five internal attributes. This result is quite reasonable because the changes underlying Pull Up Attribute and Rename Method refactoring types are too little; thus, we conclude that they have small importance when it comes to avoid code quality decay. Extract Interface, Extract Method, Move Attribute, Move Method, and Push Down Method improved exactly one attribute. Curiously, only Move Method and Extract Interface improved attributes that one would expect to improve through these types, according to Table 2.4.

Although developers often apply Extract Method and Move Method (Murphy-Hill et al., 2012), both types rarely suffice in fully addressing complex degradation symptoms, such as Large Class and Feature Envy (Bibiano et al., 2019). Our study provides complementary insights by showing that both types rarely improve more than just one attribute. Thus, one should not expect both types are capable of fully addressing complex degradation symptoms, which require the improvement of multiple attributes, such as cohesion and coupling, to be removed. We then conclude that these refactoring types in especial should be combined with others in order to support the effective enhancement of code structures.

Extract Superclass, Inline Method, and Push Down Attribute improved exactly two attributes. All three types improved size, Push Down Attribute also improved coupling, while Extract Superclass and Inline Method improved cohesion. It was quite surprising that Push Down Attribute improve many more attributes than Pull Up Attribute. This observation may result from the particularities of class coupling in the analyzed systems; perhaps a recurring coupling issue in these systems would be simply resolved by moving attributes from the parent to the child classes. Fowler's Refactoring book (Fowler, 2018) recommends these types for fully addressing design smells, such as Duplicated Code and Large Class, that may harm software evolution. Our study confirms that these types are indeed effective in improving internal software quality, and matches recent observations that Pull Up refactorings can fully address various degraded code structures (Bibiano et al., 2019).

At Least One Metric analysis approach. Once this analysis approach is less strict, an overall improvement was expected. Indeed, the number of refactoring types that did not improve any of the five attributes has dropped from three to one. The only type that kept without improving attributes is Rename Method; this observation is fair when considering that Rename Method has no direct effect on the code structure. Extract Interface, Move Attribute, Pull Up Attribute, and Pull Up Method improved exactly one attribute. All these four types improved attributes that one would expect to improve.

Extract Method, Extract Superclass, Inline Method, Move Method, Push Down Attribute, and Push Down Method improved from two to four attributes. Especially, Extract Superclass improved all attributes but complexity. The variety of levels in which these refactoring types affect the source code is quite impressive. This result somehow reinforces previous assumptions that combining different refactoring types can succeed in fully addressing complex degradation symptoms (Bibiano et al., 2019; Fowler, 2018).

3.4.2 Worsening of Internal Attributes

Most Metrics analysis approach. Extract Interface, Extract Superclass, Move Attribute, and Rename Method did not worsen any of the five internal attributes. This result surprises because Extract Interface and Extract Superclass imply substantially changing both coupling and hierarchy of classes. The results encourage the application of both types if necessary.

Conversely, the results for Rename Method are reasonable, thereby confirming this type as harmless to code structures. Nevertheless, worsening was surprisingly more frequent than improvement in general. Inline Method and Push Down Attribute worsened exactly one attribute. Particularly, Inline Method worsened rather than improved coupling. This result suggests that developers should be careful while applying Inline Method across different classes; otherwise, they may increase the coupling between classes (Chidamber and Kemerer, 1994; Lanza and Marinescu, 2006).

Extract Method, Move Method, Pull Up Attribute, Pull Up Method, and Push Down Method worsened either two or three attributes. It is worth mentioning that four out of the five types apply to the method level. We conclude that developers are more likely to worsen the code structural quality while performing method-level refactoring. This result may derive from the high occurrence of these types in floss refactoring; thus, developers are more likely to overlook the code quality decay that results of applying these types. This result emphasizes the importance of the automated support for recommending refactoring (Tsantalis et al., 2013), especially at the method level.

At Least One Metric analysis approach. Our less strict analysis approach revealed different insights on the refactoring effect for each type. This analysis approach increased, from four to five, the number of refactoring types. In fact, Extract Interface, Extract Superclass, Move Attribute, Inline Method, and Rename Method that did not worsen any of the five internal attributes. This result reinforces that Extract Superclass and Inline Method, which are two of the types that improve many internal attributes (Section 3.4.1), can be applied without major concerns with code quality decay. These types will probably improve rather than worsen the code structural quality of systems.

Extract Method, Move Method, Push Down Attribute, and Push Down Method worsened exactly one attribute. All four types worsened cohesion. This result is quite unexpected because Pull Up and Move refactoring types are typically recommended for moving code elements across classes (Fowler, 2018); thus, the class cohesion should improve rather than worsen. Pull Up Attribute and Pull Up Method worsened exactly two attributes. This result is also unexpected once Pull Up refactorings suggest an improvement of class cohesion.

3.4.3 Most Metrics versus At Least One Metric

We rely on the thirteenth column of Table 3.4 to analyze the statistical difference of results obtained from the two analysis approaches, i.e., Most Metrics and At Least One Metric. We adopted the criteria discussed in Section 3.2 to classify the correlation strength indicated by each ρ coefficient. We discuss below the main insights obtained from the correlation analysis.

Except for Inline Method, Move Method, and Push Down Method, the overall results changed little according to the analysis approach. In the case of Inline Method (moderate correlation) and Move Method (very weak correlation), the size attribute was the most affected one. As the analysis became more strict, i.e., from the At Least One Metric to Most Metrics, the frequency in which the two refactoring types improve size have decayed in at least 32%. This result is quite expected because, as shown in Table 2.1, size is computed by the largest set of metrics, i.e., eight metrics in total.

In the particular case of Push Down Method (moderate correlation), coupling was the most affected attribute. The use of a more strict analysis approach decreased in about 32% the frequency of coupling improvement. Coupling is computed from five different metrics (Table 2.1), which is a lot to sustain the coupling improvement when the Most Metrics approach is considered. In summary, our data reveal that the analysis results are ultimately consistent regardless of the analysis approach.

3.4.4 Root-canal versus Floss Refactoring

Table 3.5 summarizes the refactoring effect on attributes by tactic, i.e., root-canal and floss refactoring. The first column lists the three refactoring effects: improve, keeps unaffected, and worsen. The second column discriminates the two analysis approaches used in this work: Most and One. From Table 3.4, we counted how many times each of the 11 refactoring types were expected to improve any of the five internal attributes – these are the so-called associated attributes. The resulting count equals 20 for each analysis approach and 40 regardless of the analysis approach. We used this count to compute the general refactoring effect specifically on the associated attributes.

Table 3.5: General Refactoring Effect on Internal Attributes by Refactoring Tactic

Refactoring		**Associated Attributes**				**All Attributes**			
Effect on	**Appr.**	**Root-canal**		**Floss**		**Root-canal**		**Floss**	
Attributes		**Abs.**	**%**	**Abs.**	**%**	**Abs.**	**%**	**Abs.**	**%**
Improve	Most	8	**40.00**	5	25.00	14	25.45	10	18.18
	One	13	**65.00**	11	**55.00**	23	41.82	23	**41.82**
Keeps Unaffected	Most	8	**40.00**	8	**40.00**	30	**54.55**	25	**45.45**
	One	7	35.00	7	35.00	27	**49.09**	21	38.18
Worsen	Most	4	20.00	7	35.00	11	20.00	20	36.36
	One	0	0.00	2	10.00	5	9.09	11	20.00

Note: Percentage values may not total 100.00 due to the two-decimals rounding

The third to sixth columns provide the absolute number (Abs.) and percentage (%) of times in which associated attributes have either improved, kept unaffected, or worsened. In total, we have 11 refactoring types and five attributes, totaling 11 * 5 = 55 pairs of type and attribute. We used this count to compute the general refactoring effect for all attributes; the results are presented in the remaining columns in both Abs. and %. We split the results by refactoring tactic aimed at capturing any differences. Cells with **bold** font indicate the highest values obtained through each analysis approach.

Results for Associated Attributes only. Regardless of refactoring tactic, refactorings are more likely to improve and keep unaffected rather than worsen internal attributes. In the Most Metrics approach, i.e., the strictest, attributes that either improved or kept unaffected summed 80% for root-canal refactoring against 65% for floss refactoring. In the At Least One Metric approach, i.e., the less strict one, the sum of attributes that improved and kept unaffected impressively reached 100% for root-canal refactoring against 90% for floss refactoring. Nevertheless, the considerable frequency of attribute worsening due to floss refactoring, which is less expressive for root-canal refactoring, is quite revealing. It seems that developers actually need guidance to enhance code structures while performing refactoring accompanied by other changes, e.g., changes intended at evolving features (Fernandes, 2019a).

Results for All Attributes. Our results are slightly different when considering all attributes regardless of being associated with specific refactoring types. In the Most Metrics approach, i.e., the strictest, attributes that either improved or kept unaffected summed 80% for root-canal refactoring against 63.63% for floss refactoring. In the At Least One Metric approach, i.e., the less strict one, the sum of attributes that improved and kept unaffected impressively reached 90.91% for root-canal refactoring against 80% for floss refactoring. The sums decreased in up to 10% but they still surpass the frequency of refactorings that worsened attributes. Comparing results for All and Associated Attributes suggests that refactorings affect associated and unassociated attributes similarly, for the good or for the bad.

3.4.5 Summary of RQ_2

Refactorings tend to either improve or keep unaffected the internal attributes, regardless of the refactoring types being associated with a particular attribute. Our results pointed out at least 64% of attributes improved or unaffected for the strictest analysis approach, i.e. Most Metrics, against at least 80% for the less strict approach, i.e. At Least One Metric. There are two way of interpreting the high rate of attributes kept unaffected.

One the one hand, it may be the case in which refactorings in general are insufficient to overcome such a complex degradation symptom while evolving features. On the other hand, it is possible that developers only fully address degradation symptoms that really harm software evolution. As a result, developers end up postponing the correction or ignoring less relevant symptoms. Our qualitative study (Chapter 4) further investigates this issue.

Finally, the rates of attribute improvement in isolation only slightly surpass the rates of attribute worsening. Thus, developers need further assistance for addressing critical attributes while evolving features.

3.5 Re-Refactoring Effect on Internal Attributes (RQ_3)

3.5.1 Improvement of Internal Attributes

Table 3.6 summarizes the re-refactoring effect by internal attribute (columns) and refactoring type (rows). The table structure is the same of

Table 3.4. Due to trace loss of methods that remained the same except for the name, we did not compute re-refactorings of the Rename Method type. Thus, we discarded this refactoring type from our analysis.

Table 3.6: Re-refactoring Effect on Internal Attributes by Refactoring Type

Type	**Appr.**	**Cohesion**		**Complexity**		**Coupling**		**Inheritance**		**Size**		ρ
		E	**%**	**E**	**%**	**E**	**%**	**E**	**%**	**E**	**%**	
Extract	Most	–	85.71	–	100.00	–	76.19	↑	**57.14**	–	**54.76**	0.98
Interface	One	–	85.71	–	100.00	–	69.05	↑	**59.52**	–	**50.00**	
Extract	Most	↑	54.39	–	57.89	↑	55.26	↑	**95.61**	↑	**86.84**	1.00
Superclass	One	↑	54.39	–	57.89	–	61.40	–	**50**	↑	**71.93**	
Move	Most	↑	**65.70**	–	90.24	–	**59.74**	–	**91.37**	–	56.86	0.99
Attribute	One	↑	**65.70**	–	90.24	–	**54.34**	–	**89.46**	–	54.73	
Pull Up	Most	↓	47.88	–	61.00	↓	59.07	–	77.22	↓	**69.50**	0.99
Attribute	One	↓	46.98	–	61.00	↓	59.07	–	72.20	↑	**85.71**	
Push Down	Most	↑	40.00	–	85.71	↑	42.86	–	**97.14**	↓	54.29	0.98
Attribute	One	↑	40.00	–	85.71	↑	54.29	–	**97.14**	↑	82.86	
Extract	Most	↓	59.11	↑	44.93	↓	**45.77**	–	93.82	↓	**58.53**	1.00
Method	One	↓	59.11	↑	46.99	↑	**71.40**	–	93.10	↑	**85.85**	
Inline	Most	↑	58.32	–	48.98	↓	**39.91**	–	92.90	↑	**56.54**	0.68
Method	One	↑	58.32	–	48.13	↑	**77.65**	–	91.45	↑	**89.02**	
Move	Most	↓	**47.52**	–	**71.17**	↓	**38.29**	–	**84.23**	↑	44.26	0.25
Method	One	↓	**47.52**	–	**71.17**	↑	**51.01**	–	**77.59**	↑	85.25	
Pull Up	Most	↓	39.20	–	67.77	↓	72.43	–	91.36	↓	**59.47**	1.00
Method	One	↓	40.32	–	67.77	↓	72.99	–	91.03	↑	**83.72**	
Push Down	Most	↓	48.57	–	51.43	↑	42.86	–	**94.29**	↓	57.14	0.67
Method	One	↓	49.02	–	51.43	↑	80.00	–	**94.29**	↑	84.29	

E stands for effect: improved (↑), worsened (↓), and kept unaffected (–)

Most Metrics analysis approach. Pull Up Attribute and Pull Up Method did not improve any of the five internal attributes. This result is expected for Pull Up Attribute once its effect on code structures is quite small. Conversely, this result is counter-intuitive for Pull Up Method because applying Pull Up Method after other refactorings is often recommended for fully addressing complex degradation symptoms (Bibiano et al., 2019; Fowler, 2018).

Extract Interface, Extract Method, Move Attribute, Move Method, and Push Down Method improved exactly one attribute. This result is equivalent to the one obtained for refactorings in general (Section 3.4.1). Inline Method and Push Down Attribute improved exactly two attributes. In this case, the results obtained for refactorings in general has remained the same. Once these types have been recommended for fully addressing design smells (Bibiano et al., 2019; Fowler, 2018), our results further encourage their use.

Finally, Extract Superclass surprisingly improved four attributes except complexity, which differs from the previously discussed results; thus, our results strongly suggest applying this refactoring type.

At Least One Metric analysis approach. With this less strict analysis approach, all refactoring types have improved at least one attribute. Extract Interface, Move Attribute, Pull Up Attribute, and Pull Up Method improved exactly one attribute. Curiously, this result is shared by the analysis of refactorings in general (see Section 3.4.1 for details).

Extract Superclass, Move Method, and Push Down Method have improved exactly two attributes. This result is especially interesting in the case of Move Method and Push Down Method, which are often recommended to succeed other refactoring types, e.g., Extract Method and Inline Method, in order to fully address certain types of design smells (Bibiano et al., 2019; Fowler, 2018). These types include Large Class, Long Method, and others (Fowler, 2018; Lanza and Marinescu, 2006).

Finally, Push Down Attribute, Extract Method, and Inline Method improved three attributes. Again, we found evidence that applying these types after other refactorings can indeed enhance code structures.

3.5.2 Worsening of Internal Attributes

Most Metrics analysis approach. The re-refactoring effect on worsening attributes is exactly the same when compared to general refactorings (Section 3.4.1) – except by the exclusion of Rename Method from our analysis. Extract Interface, Extract Superclass, and Move Attribute did not worsen any of the five internal attributes. In other words, even after applying other refactorings to a particular code class, the application of these three refactoring types did not cause the code quality decay.

Extract Method, Move Method, Pull Up Attribute, Pull Up Method, and Push Down Method worsened either two or three attributes. This result at least partially explains why combining these types with others, or their successive application to the same code element, often fails to fully remove design smells (Bibiano et al., 2019). We recommend that tool designers take into account the possibly negative effect of applying these types while they provide developers with refactoring recommendations.

At Least One Metrics analysis approach. By excluding Rename Method, the re-refactoring effect on worsening attributes is quite similar to the one observed for general refactorings (Section 3.4.2). Extract Interface, Extract Superclass, Move Attribute, Push Down Attribute, and Inline Method did not worsen any of the five internal attributes. Extract Method, Move Method, and Push Down Attribute worsened exactly one attribute. This result emphasizes the need for carefully applying both Extract Method and Move Method, otherwise there is an increasing risk of code quality decay (Bibiano et al., 2019).

Finally, Pull Up Attribute and Pull Up Method worsened exactly two attributes. For both types, cohesion and coupling usually become critical. This observation conflicts with the common wisdom that Pull Up refactorings help to properly distribute the features implemented by each class (Fowler, 2018).

3.5.3
Most Metrics versus At Least One Metric

According to the thirteenth column of Table 3.6, the overall results obtained for refactorings in general (Section 3.4.3) occurred again for re-refactorings only. Except for Inline Method, Move Method, and Push Down Method, the results changed little according to the analysis approach.

In the case of Inline Method (moderate correlation) and Move Method (very weak correlation), size was the most affected attribute. As the analysis became more strict, i.e., from the At Least One Metric to Most Metrics, the frequency in which the two refactoring types improve size have decayed in at least 32%. Once again, this result is expected because we selected seven metrics in total to compute size (Table 2.1), which makes it harder to keep the results consistent when considering a more strict analysis approach.

In the case of Push Down Method (moderate correlation), coupling was the most affected attribute. The attribute improvement decreased in 37% when considering the strictest analysis approach, i.e., Most Metrics. Similar reasoning applies to this result when compared to size: many metrics capture coupling, five in total. In summary, our results are consistent regardless of the analysis approach, even for re-refactoring.

3.5.4
Root-canal versus Floss Refactoring

Table 3.7 summarizes the re-refactoring effect on internal attributes by tactic, i.e., root-canal and floss refactoring. The first column lists the three alternatives for the re-refactoring effect: improve, keeps unaffected, and worsen. The second column discriminates our two analysis approaches: Most and One. We relied on Table 3.6 to compute how many times each out of the 10 refactoring types, except Rename Method, were expected to improve any of the five internal attributes – the so-called associated attributes. The resulting count equals 19 for each analysis approach and 38 regardless of the analysis approach. We used this count to compute the general re-refactoring effect specifically on the associated attributes.

The third to sixth columns provide the absolute number (Abs.) and percentage (%) of times in which associated attributes have either improved, kept unaffected, or worsened. In total, we have 10 refactoring types and five attributes, totaling 10 * 5 = 50 pairs of type and attribute. We used this count to compute the general re-refactoring effect for all attributes; the results are presented in the remaining columns in both Abs. and %. We split the results

Table 3.7: General Re-refactoring Effect on Internal Attributes by Refactoring Tactic

Refactoring Effect on Attributes	**Appr.**	**Associated Attributes**				**All Attributes**			
		Root-canal		**Floss**		**Root-canal**		**Floss**	
		Abs.	**%**	**Abs.**	**%**	**Abs.**	**%**	**Abs.**	**%**
Improve	Most	9	**47.37**	16	32.00	5	26.32	9	18.00
	One	13	**68.42**	26	**52.00**	12	**63.16**	22	**44.00**
Keeps Unaffected	Most	6	31.58	23	**46.00**	7	**36.84**	22	**44.00**
	One	6	31.58	19	38.00	6	31.58	17	34.00
Worsen	Most	4	21.05	11	22.00	7	**36.84**	19	38.00
	One	0	0.00	5	10.00	1	5.26	11	22.00

Note: Percentage values may not total 100.00 due to the two-decimals rounding

by refactoring tactic aimed at capturing any differences. Cells with **bold** font indicate the highest values obtained by analysis approach.

Results for Associated Attributes only. Re-refactorings rarely worsen internal attributes, regardless of refactoring tactic. In the Most Metrics approach, i.e., the strictest, attributes that either improved or kept unaffected summed 78.95% for root-canal refactoring against 78% for floss refactoring. This result considerably differs from the one observed for refactorings in general (Table 3.5). Thus, the frequency did not change between refactoring tactics.

In the At Least One Metric approach, i.e., the less strict one, the sum of attributes that improved and kept unaffected impressively reached 100% for root-canal refactoring against 90% for floss refactoring. These results are equivalent to those observed previously, which is quite expected due to the loose analysis approach we have defined.

Results for All Attributes. Our results are slightly different when considering all attributes regardless of being associated with specific refactoring types. In the Most Metrics approach, i.e., the strictest, attributes that either improved or kept unaffected summed 63.16% for root-canal refactoring against 62% for floss refactoring.

In the At Least One Metric approach, i.e., the less strict one, the sum of attributes that improved and kept unaffected impressively reached 94.74% for root-canal refactoring against 78% for floss refactoring. The sums changed little when compared to the analysis of Associated Attributes.

In summary, regardless of the analysis of associated attributes or all attributes, root-canal refactoring outperformed floss refactoring in improving the internal attributes. To the current knowledge, this result is quite new because it reveals that developers are more likely to enhance code structures when they perform "pure" refactorings, i.e., refactorings without any other changes. Still, we are aware that developers frequently have to perform refactorings while performing other changes, e.g. aimed at evolving features (Fernandes, 2019a; Murphy-Hill et al., 2012; Paixao et al., 2019; Silva et al., 2016a). Thus, our results emphasize the need for supporting developers in performing floss refactoring via refactoring recommendations.

3.5.5 Summary of RQ$_3$

Our quantitative data suggests that re-refactorings tend to improve or keep unaffected the internal attributes, just as refactorings in general. However, the rates of attribute improvement are slightly greater than the rates of worsening: up to 26% greater in the case of root-canal refactoring for the associated attributes. Thus, developers may re-refactor for gradually addressing more complex degradation symptoms while evolving features. Otherwise, developers would not refactor the same code element successively, or even improve rather than worsen the code structure and its design.

3.6 Comparison of Refactoring and Re-refactoring Effect (RQ$_4$)

3.6.1 Quantitative Results

Table 3.8 summarizes the Pearson's correlation coefficients, i.e., the ρ coefficients, obtained from a comparison of study results obtained for refactoring in general and re-refactoring. The first column discriminates two natures of study results: the (re-)refactoring effect by refactoring type, which is reported by Tables 3.4 and 3.6, and the general (re-)refactoring effect regardless of refactoring type, which is reported in Tables 3.5 and 3.7. The second column lists the alternatives analyzed across our study: rows three to 12 list the ten refactoring types whose refactoring effect we analyzed in both scenarios: refactoring effect and re-refactoring effect. Rows 13 to 15 list the variants of (re-)refactoring effect analyzed regardless of refactoring type.

Table 3.8: Correlation of Effect Observed for Refactoring in General and Re-refactoring

Analysis	Alternatives	ρ Coefficients	
		Most	At Least One
Effect by refactoring type	Extract Interface	0.93	0.94
	Extract Superclass	0.95	1.00
	Move Attribute	1.00	1.00
	Pull Up Attribute	0.81	0.98
	Push Down Attribute	1.00	0.99
	Extract Method	1.00	1.00
	Inline Method	1.00	1.00
	Move Method	1.00	1.00
	Pull Up Method	0.97	0.98
	Push Down Method	0.96	0.95
General effect regardless of type	Improve	0.98	0.55
	Keeps Unaffected	-0.12	0.71
	Worsen	0.81	0.97

(Re-)refactoring effect for each refactoring type. Using the Most Metrics analysis approach, the frequency results obtained for refactoring in general and re-refactoring changed very little for all refactoring types. We draw this observation from the third column, which points out either a strong correlation or a very strong correlation for all types. The same observation is valid for the At Least One Metric approach, for which all refactoring types obtained a very strong correlation. Thus, when not considering each internal attribute in isolation, we concluded that both refactoring in general and re-refactoring presented the same effect on attributes.

(Re-)refactoring effect regardless of the refactoring type. Using the Most Metrics analysis approach, we observed a strong or very strong correlation of results regarding the improvement and the worsening of internal attributes. Thus, both refactoring in general and re-refactoring ultimately improved the same attributes, while there was a slight difference in the worsened attributes. When comparing Tables 3.4 and 3.6, we observe that re-refactorings worsened a few less attributes when compared to refactorings. This result is surprising because we expected that re-refactorings would imply on further quality improvements, which our analysis revealed to be very little in practice. Nevertheless, the benefit of performing re-refactoring was not significantly superior to the one obtained by performing simple refactorings.

This result is consisted with our most recent achievements regarding the ineffectiveness of composite refactoring (Section 2.4) in fully removing design smells (Bibiano et al., 2019). Finally, we observed a very weak correlation of results with respect to keeping unaffected the internal attributes. We discussed in Section 3.5.4 that re-refactorings are more likely to either improve or worsen attributes, probably because re-refactoring a code element has a cumulative effect on the attributes. For instance, the more code extractions (e.g., via Extract Method) you apply on the same code element, the greater the chances of substantially changing the size of a code element. Thus, this weak correlation is reasonable.

Using the At Least One Metric approach, we observed a different effect on internal attributes. We observed a very strongly correlation regarding the attribute worsening, while a strong correlation was observed with respect to keeping unaffected the attributes. In other words, both refactoring in general and re-refactoring worsened and kept unaffected the same attributes. Conversely, the attributes improved by one or another changed considerably, which is confirmed by the moderate correlation observed for attribute improvement. This result is reasonable because, when considering the At Least One Method approach, the improvement of any metric leads to the overall improvement of the respective internal attribute.

In summary, regardless of the adopted analysis approach, we observed a consistency of results obtained for both refactoring in general and re-refactoring. Such consistency contrasts with our assumptions that our results would be sensitive to the different analysis approaches (Section 3.2). The Most Metrics approach did not outperform At Least One Metric.

3.6.2 Summary of RQ$_4$

When considering each refactoring type in isolation, re-refactoring affects the internal attributes similarly to refactoring in general (RQ$_2$). This study result contradicts assumptions that re-refactoring boosts the code quality improvement significantly.

However, when considering all data regardless of the refactoring type, the results change considerably for attributes improved and kept unaffected. Particularly, developers seem to evolve features in the same code elements constantly, so that a slightly greater attribute improvement is observed in comparison with attribute worsening. Developer may have been gradually addressing degradation symptoms as they hinder feature additions or enhancements.

Finally, it is possible that changes applied in conjunction with refactorings, in floss refactoring, prevent a more significant code quality improvement. Further studies are required to confirm whether these changes introduce degradation symptoms to evolving features.

3.7 Our Study Versus Related Work

3.7.1 Study Comparison at the Design Level

While searching for studies on (re-)refactoring, we found a systematic literature review (Al Dallal and Abdin, 2017) regarding effect on internal and external quality attributes. We discarded all restricted to external attributes, as those studies published before 2010, towards a comprehensive and updated list. We complemented the six remaining papers (Fontana and Spinelli, 2011; Källén et al., 2014; Kim et al., 2014; Murgia et al., 2011; Szőke et al., 2015b; Veerappa and Harrison, 2013) with other three studies (Bavota et al., 2015; Chaparro et al., 2014; Soetens and Demeyer, 2010).

Table 3.9 compares our study with the others by characteristic as follows. *Projects*: total number, nature (open or closed source), and analysis made by project (oriented to commits or major releases). *Refactoring*: detection (manual or tool-supported) and types. *Metrics*: total number and internal quality attributes. We draw below some conclusions from the table data, thereby highlighting the differences among studies.

Table 3.9: Design-level comparison of our study with previous work

Study	**Projects**			**Refactoring**		**Metrics**	
	Total	**Nature**	**Analysis**	**Detection**	**Types**	**Total**	**Attributes**
Ours	23	Open source	Commits	Refactoring-Miner	11	25	Cohesion, complexity, coupling, inheritance, size
(Bavota et al., 2015)	3	Open source	Major releases	Ref-Finder	63	11	Cohesion, complexity, coupling, inheritance, size
(Szőke et al., 2015b)	1	Closed source	Commits	n/a	n/a*	10	Complexity, coupling, inheritance, size
(Chaparro et al., 2014)	15	Open source	Commits	Unnamed Tool	12	11	Cohesion, complexity, coupling, inheritance, size
(Källén et al., 2014)	1	n/a	n/a	n/a	n/a*	8	Cohesion, complexity, coupling, inheritance, size
(Kim et al., 2014)	1	Closed source	Commits	Manual	n/a*	16	Complexity, coupling (aka dependency), size
(Veerappa and Harrison, 2013)	8	Open source	n/a	n/a	n/a*	4	Coupling
(Fontana and Spinelli, 2011)	1	Open source	n/a	n/a	3	6	Cohesion, complexity, coupling, size
(Murgia et al., 2011)	5	Open source	Major releases	Ref-Finder	39	2	Coupling
(Soetens and Demeyer, 2010)	1	Open source	Commits	Manual	3*	1	Complexity

n/a: not applicable or unspecified by the paper authors
*Refactoring effect evaluated regardless of type

We have targeted the largest data set composed of 23 projects against 15 for the second largest set. Additionally, our work is among the 70% targeting open source rather than closed source projects. By analyzing open source projects, we achieved a certain diversity that certainly constrained studies such as (Kim et al., 2014; Veerappa and Harrison, 2013).

As discussed in Section 3.3.2, analyzing major releases can overlook the behavior of code metrics between a pair of releases, which may comprise dozens of interesting commits. Although only 20% of studies performed such analysis, 30% did not make it clear the employed analysis approach (Fontana and Spinelli, 2011; Källén et al., 2014; Veerappa and Harrison, 2013). Different of past work, we not just performed a commit-by-commit analysis, but also mitigated threats by deriving metric thresholds for each commit (Section 3.2).

Only a half of the studies made the refactoring detection technique explicit. The authors of four out of the ten papers have reportedly used automated tools. The authors of two papers used Ref-Finder, whose literature criticism we reported in Section 3.3.2; the authors of two other papers performed a manual detection based on refactoring-related branches and commit logs. We adopted a state-of-the-art tool with high accuracy (Tsantalis et al., 2013), which not just dispensed a manual validation, but also helped in detecting a large and varied refactoring set.

Only 50% of the papers computed the refactoring effect by types as we did in this work. We know that each refactoring operation is fine-grained and varies in granularity. Thus, we encourage the data analysis per type to unveil the nuances of the refactoring effect.

More critically, only 50% of the papers analyzed all five attributes covered by our work. Nevertheless, we have evaluated the largest set of 25 traditional code metrics. We highlight our careful selection of alternative metrics to the traditional LCOM (Section 3.3.2) for boosting the reliability of our study.

In this work, we employed a *tour de force* to carefully address many threats to validity of past research reported by the aforementioned literature review (Al Dallal and Abdin, 2017). Aimed at providing a large-scale study, we analyzed 23 open source projects with thousands of refactoring operations and commits. We also analyzed 15 metrics to expand the scope of previous studies and derive wide insights on the (re-)refactoring effect. We overcame a lack of statistical tests via computing and Fisher's test computation (Section 3.2).

3.7.2
Study Comparison at the Results Level

Most existing studies regarding the refactoring effect lack an analysis of either each refactoring type or each attribute in isolation (Al Dallal and Abdin, 2017). Hence, it is hard to compare results across different studies. From the studies listed in Table 3.9, only one paper (Bavota et al., 2015) is fairly comparable to ours, especially due to the similar study design. We already compared our work with this particular study (Section 3.3.2). Thus, this section compares our findings with those summarized by the literature review (Al Dallal and Abdin, 2017) – rather than with each previous work separately.

Table 3.10 presents the comparison in terms of three refactoring types covered by both studies: *Extract Method*, *Move Field*, and *Move Method*. The third column informs which analysis approach, i.e., Most Metrics and At Least One Metric, supported the refactoring effect analysis in our work. *Matches* counts how many times the refactoring effect summarized by the literature review (Al Dallal and Abdin, 2017) matches our empirical observations. The remaining columns present the refactoring effect observed on each internal quality attribute.

Extract Method. Our results matched previous work observations for complexity – i.e., complexity usually improves after the refactoring application

Table 3.10: Design-level comparison of our study with previous work

Type	Study	Approach	Internal Quality Attribute				
			Cohesion	**Complexity**	**Coupling**	**Inheritance**	**Size**
Extract Method	Past work	n/a	Improves	Improves	Unaffected	Unaffected	Improves
	Ours	Most	Worsens	Improves	Worsens	Unaffected	Worsens
		One	Worsens	Improves	Improves	Unaffected	Improves
	Both	**Matches**	**0**	**2**	**0**	**2**	**1**
Move Field	Past work	n/a	Improves	n/a	Worsens	n/a	n/a
	Ours	Most	Improves	Unaffected	Unaffected	Unaffected	Unaffected
		One	Improves	Unaffected	Unaffected	Unaffected	Unaffected
	Both	**Matches**	**2**	**n/a**	**0**	**n/a**	**n/a**
Move Method	Past work	n/a	Improves	Worsens	Worsens	n/a	n/a
	Ours	Most	Worsens	Unaffected	Worsens	Unaffected	Improves
		One	Worsens	Unaffected	Improves	Unaffected	Improves
	Both	**Matches**	**0**	**0**	**1**	**n/a**	**n/a**

Past work: Al Dallal and Abdin (2017)
n/a: not applicable or unspecified by the paper authors

– and inheritance – i.e., inheritance keeps unaffected regardless of the analysis approach. Our results also matched the previous ones for size. Size improved according to the At Least One Metric analysis approach, but such improvement did not resist to a more strict analysis approach. Finally, there was a complete mismatch between our results and the previous ones for cohesion – i.e., cohesion worsens rather than improves after the refactoring application – and coupling – i.e., the more strict the analysis approach, the worst was the observed effect.

Move Field. Our study results matched previous work observations for cohesion. Indeed, cohesion typically improves after the refactoring application. However, our results did not match those observed by the literature with respect to coupling. Contrary to previous work that observed the coupling worsening caused by the *Move Field* application, we observed that the attribute kept unaffected most of the times. Our results were consistent for both the less strict and the most strict one analysis approach. Note that we quantified coupling through five metrics (Table 2.1), which gives our results a certain robustness.

Move Method. Our study matched previous observations for coupling, i.e., cohesion worsens rather than improves after the refactoring application, using our strictest analysis approach. Although a mismatch occurred with the less strict approach, the coupling worsening is the most reliable finding. Conversely, our results mismatched the previous ones for cohesion and complexity, regardless of analysis approach. Cohesion worsens rather than improves, while complexity keeps unaffected rather than worsens. We highlight that our analysis covered three cohesion metrics and five complexity metrics. Once our results were not sensitive to the analysis approach, we conclude that cohesion typically worsens while complexity typically keeps unaffected after applying *Move Method*.

3.8 Threats to Validity

Construct Validity. We followed strict procedures for selecting systems to conduct the analysis. We collected many systems from GitHub open repositories. After that, we applied filters to select Java systems, all open source and with many refactorings to be analyzed. We relied on our previous work (Chávez et al., 2017) to define fair criteria, thereby avoiding the system selection bias and subjectivity. With respect to the refactoring detection, we looked for a detection tool with high accuracy rates and able to detect different refactoring types. Unlike the previous work closest to ours (Bavota et al., 2015), we relied on a highly accurate tool called RefactoringMiner. Although the number of detected refactoring types dropped from 63 to 11, the detection accuracy has increased substantially. Thus, we expected to reduce the rates of false positives, thereby reaching a reliable set of refactorings by system.

We investigated the literature on software evolution to cherry-pick metrics for analysis. We relied on 25 metrics either proposed or exploited by previous work, e.g., (Chidamber and Kemerer, 1994; Lanza and Marinescu, 2006; Lorenz and Kidd, 1994; McCabe, 1976). Especially, we replaced a traditional cohesion metric with others that are more reliable, in order to avoid data noise. Finally, this work inherited a random set of manually classified refactorings by tactic explored in a previous work (Chávez et al., 2017). Three refactoring specialists participated in the manual classification aimed to avoid biases; one of them has helped in solving ties and reaching a consensus.

Internal Validity. Section 2.4 describes how we computed re-refactoring instances for each system analyzed. For a refactoring `r` to be a re-refactoring instance, another refactoring `r'` must have occurred previously during software evolution and share the same granularity (attribute, method or class). We acknowledge that using this definition implies a threat to the study validity, especially regarding the set of re-refactorings whose effect on internal attributes we assessed. In particular, we may have overlooked cases where refactorings applied at different granularities affect a common subset of code elements that constitute part of the code structure and its design.

An example of possibly overlooked case is this: if developers applied a refactoring `r'` (e.g. Move Method) to a method `m()` and, after, a refactoring `r` (e.g. Move Class) to class `C` which implements `m()`. Although we did not address this threat in our current work, we explored other mechanisms for assessing the successive applications of refactorings to a (subset of) code elements in other studies (Bibiano et al., 2019; Bibiano et al., 2020; Fernandes et al., 2019b).

Examples of these mechanisms are composite refactoring (Bibiano et al., 2020) and batch refactoring (Bibiano et al., 2019; Fernandes et al., 2019b) – both briefly mentioned in Section 2.4 once they are not explored in the thesis.

Another threat to validity of the re-refactoring computation may be the interval of application between refactorings `r'` and `r`. We did not restrict the interval of application to a particular time span of number of commits in between refactorings. This decision may have led to the computation of too distant re-refactorings, which may relate little with the code structure enhancement intended for the same code elements through the original refactoring. At least in the context of batch refactoring (Section 2.4), it seems to rarely takes more than a few weeks (or up to three months) for developers to re-refactor the same (subset) of code elements (Ferreira, 2018). Thus, this threat may have been minimal, though further analyses are required to confirm this in the particular context of re-refactoring.

Finally, we payed special attention to the data collection and tabulation. Once our quantitative study relies on large amounts of data, we used spreadsheets and databases for storing and manipulating all (re-)refactorings, metrics, and internal attributes data. At least two researchers engaged in the data tabulation. By doing that, we expected to avoid missing and duplicated data as much as possible.

Conclusion Validity. We carefully planned the data analysis procedures documented in Section 3.2 prior to the analysis execution. Again, at least two researchers contributed to the descriptive analysis aimed at answering our research questions (Sections 3.3, 3.4, and 3.5). Our descriptive analysis consisted of presenting both absolute numbers and percentages with respect to the (re-)refactoring effect on attributes; our goal was providing a detailed view on the phenomenon by refactoring type and tactic.

As a complement to our previous work (Chávez et al., 2017), and in accordance with suggestions of a recent work (Al Dallal and Abdin, 2017), we applied statistical tests. We opted for the Fisher's exact test to understand the application of (re-)refactorings to critical code elements (Section 3.3). We also computed correlation to understand how different were the results obtained through each analysis approach (Sections 3.4 and 3.5), besides comparing the results obtained for refactoring in general and re-refactoring. Statistical testing aimed to leverage our data analysis, thereby allowing us to understand the (re-)refactoring effect on attributes.

We highlight that our quantitative study reported in this chapter targets a correlation analysis rather a causality analysis. In particular, our quantitative data extracted from the 23 systems analyzed does not reveal much of the developer's intents behind refactorings. There is also little to conclude, based on purely quantitative data, about the extent to which the refactoring effect on internal attributes is expected, planned or not by developers while performing refactorings. Due to this inherent limitation of our study design, we propose a qualitative study approach in Chapter 4 based on the developer's perception of critical attributes. Thus, we expect to better understand the role of refactorings in managing critical attributes during software evolution.

External Validity. As observed by a recent literature review on the study of refactorings (Al Dallal and Abdin, 2017), previous studies explored the refactoring effect in very limited scopes. More critically, we did not find studies on the re-refactoring effect. In general, a limited scope leads to the threat of little study generality. We addressed this threat by performing a large quantitative study with 23 systems, which expands a lot the previous studies scopes. We discussed that the systems selected for analysis (Table 3.1) are diversified in terms of commit and refactoring count.

Nevertheless, we are aware that our focus on Java open source systems also represents a scope limitation. We encourage researchers to address this limitation by exploring the (re-)refactoring effect in trending programming languages, such as JavaScript and Python. Similar reasoning applies to the explored sets of 11 refactoring types and 25 metrics. We explored metrics listed in well-known catalogs, e.g., (Chidamber and Kemerer, 1994; Lanza and Marinescu, 2006; Lorenz and Kidd, 1994), and refactoring types often applied by developers in practice (Murphy-Hill et al., 2012). Still, we encourage researchers to investigate other metrics and refactoring types.

3.9 Chapter Summary

In this study, we presented and discussed the outcomes of a quantitative study on the relationship between (re-)refactorings and internal attributes. Particularly, we assessed the effect of either refactorings in general or re-refactorings on improving and worsening five attributes: cohesion, complexity, coupling, inheritance, and size. The analysis for each attribute contrasted, to some extent, with the evidence that refactorings often fall short in removing design smells (Bibiano et al., 2019; Yoshida et al., 2016). This is because, while refactorings may not fully remove design smells, they can sometimes benefit certain internal attributes. In summary, through our study, we provided a more fined-grained view of the (re-)refactoring effect towards an enhanced code structure.

Answering RQ$_1$ revealed that critical attributes are potentially relevant degradation symptoms in practical settings. RQ_2 showed that most refactorings either improve or keep unaffected internal attributes, regardless of being critical. Still, our results did not clarify whether i) refactorings are insufficient to overcome critical attributes or ii) developers only mitigate or fully address critical attributes that are actually relevant for evolving features. RQ_3 and RQ_4 show that both refactorings and re-refactorings perform similarly with respect to enhancing code structure and its design.

We address the majority of aforementioned issues in the second study that constitutes this doctoral thesis. The next chapter introduces a case study (Runeson and Höst, 2009) based on focus group sessions (Kontio et al., 2004). By selecting two industry cases where software evolution is a major task, we promote discussion on how much (and why) critical attributes are relevant for evolving features. Chapter 4 describes the study goal and presents our study results, including refactoring recommendations to help manage critical attributes while evolving features.

4
On the Relevance of Critical Attributes for Evolving Features

Development teams typically have to apply hundreds of changes to their software systems along with the software evolution (Burke, 2014; Kim et al., 2014; Paixao et al., 2019). A major side effect of applying changes while adding or enhancing features is unexpectedly degrading the source code structure and its design (Le et al., 2016; Tufano et al., 2017). This is because, consciously or not, developers sometimes introduce degradation symptoms (Bavota et al., 2013; Tufano et al., 2017). Each degradation symptom is a potential threat to understanding and changing code elements, e.g. classes (Fowler, 2018; Yamashita and Moonen, 2013). Hence, periodically monitoring the occurrence of degradation symptoms may help enhancing the internal software quality, as the longevity of systems (Le et al., 2016).

Several techniques have been proposed for monitoring the occurrence of degradation symptoms in a system (Chávez et al., 2017; Fernandes et al., 2016b; Lanza and Marinescu, 2006; Lorenz and Kidd, 1994). The majority of existing techniques rely on measuring the code structure aimed at spotting degraded code structures and design (Fernandes et al., 2016b). Using internal quality attribute is a major technique (Chávez et al., 2017). Each internal attribute captures a particular aspect of internal software quality (Fernandes et al., 2020). Two examples of popular internal attributes are cohesion (Chidamber and Kemerer, 1994) and complexity (McCabe, 1976). While cohesion captures the interrelation degree of attributes and methods within a class (Chidamber and Kemerer, 1994; Lanza and Marinescu, 2006), complexity captures the cognitive difficulty of understanding code elements (Nejmeh, 1988).

Monitoring internal attributes usually requires measuring code structures through traditional software metrics (Fernandes et al., 2020). Lack of Cohesion (LCOM2) is an example of metric aimed at capturing the degree of class cohesion (Chidamber and Kemerer, 1994). In turn, Cyclomatic Complexity (CC) is a metric designed for capturing the complexity degree of a method (McCabe, 1976). Developers may compare metric values to a reference value (Vale et al., 2018). Thus, a development team may reason on how adequate the metric values are based on the developer's expertise, metrics distribution analysis, or other factors (Bavota et al., 2013; Pantiuchina et al., 2018).

Assessing metric values allows to draw strategies for managing critical internal attributes, e.g. low cohesion and high complexity. A critical attribute is an internal attribute whose metrics used for capturing it assume anomalous values in comparison to the reference value (Fernandes et al., 2020; Vale et al., 2018). Section 3.2 discusses two scenarios where critical attributes emerge. However, in this chapter, we are particularly concerned on metrics that become critical as their values increase. This scenario applies to metrics like LCOM2 and CC, whose high values suggest classes with non-cohesive features or a high complexity, respectively (Chidamber and Kemerer, 1994; McCabe, 1976). In summary, critical attributes characterize degradation symptoms that developers should consider managing – either mitigating or fully addressing – for the sake of software evolution (Chávez et al., 2017; Fernandes et al., 2020).

Refactorings are largely advertised as effective means to help manage different types of degradation symptoms (Kim et al., 2014; Murphy-Hill et al., 2012), including critical attributes (Chávez et al., 2017). For instance, Extract Method and Move Method can help manage the large size and high complexity of a method, respectively (Bibiano et al., 2019; Fowler, 2018). However, until recently, there was little empirical evidence on how refactorings affect internal attributes, regardless of being critical. We addressed this literature gap in Chapter 3 through a large quantitative study targeting five internal attributes: cohesion, complexity, coupling, inheritance, and size.

Although refactorings may enhance source code structure and its design (Fowler, 2018; Murphy-Hill et al., 2012), Chapter 3 suggests the refactoring effect on internal attributes is diverse. Only a few refactoring types, such as Extract Superclass and Push Down Attribute (Fowler, 2018), often improve many attributes together (Section 3.4.1). Thus, developers are encouraged to apply these refactorings. However, refactoring types like Extract Method and Move Method improve one or another attribute while worsening others (Section 3.4.2). Thus, developers should carefully apply these refactorings in order to avoid the unexpected worsening of internal attributes that may possibly become critical for software evolution.

The design of our quantitative study (Chapter 3) limited our findings to the frequency in which refactorings improve, worsen, or keep unaffected each internal attribute. Contrary to a previous work (Bavota et al., 2013), we have found that most refactorings (94%) affect code elements with at least on critical attribute. Moreover, we characterized those refactoring types more likely to improve each internal attribute. These types are the most recommended for either mitigating or fully addressing critical attributes. Still, the purely quantitative nature of our previous study led to a major question: how much (and why) are critical attributes perceived as relevant by developers while evolving features?

Answering this question could significantly contribute to how we assist developers in refactoring while performing software evolution. For instance, let us suppose that developers find little relevance in an internal attribute that refactorings often worsen, e.g. coupling (see Table 3.4 for details). That is, worsening this internal attributes does not necessarily hinder the task of evolving features. Thus, applying those refactorings would be less threatening than it appeared based on what was observed in our quantitative study where the perceived relevance was not considered. Conversely, there may be internal attributes often worsened by refactorings that developers find very relevant in practice. In this case, we should further assist refactorings in order to prevent these internal attributes from worsening and becoming critical.

This chapter presents a qualitative study aimed at filling this literature gap. We carefully designed and conducted an industrial case study based on strict research guidelines (Runeson and Höst, 2009). Our study goal is three-fold. First, we investigate the relevance degree reported by developers on critical attributes spotted by the five internal attributes assessed in Chapter 3 – cohesion, complexity, coupling, inheritance, and size. Second, we elicit reasons why developers find each critical attribute relevant (or irrelevant) while evolving features. Third, we recommend refactorings to help manage critical attributes from a data crossing between Chapter 3 – refactoring effect on internal attributes – and this chapter – relevance of critical attributes.

We recruited two development teams of an industry-academy joint initiative for Research and Development (R&D) in Brazil called ExACTa[1]. Each team engaged in a particular focus group session (Kontio et al., 2004). Developers discussed the relevance of six critical attributes: low class cohesion, high class complexity, high class coupling, large class hierarchy depth, large class hierarchy breadth, and large class size. We refer to *relevance* as the need for either mitigating or fully addressing critical attributes while evolving features.

We targeted critical attributes at the class level, thereby discarding others, such as high method complexity and large method. We did that because classes are key code elements constituted of many others, including attributes and methods. In addition, we did that for fitting all discussions among developers in up to two hour focus group sessions, so we discarded

[1]http://www.exacta.inf.puc-rio.br/

other critical attributes, e.g. at the method level – which are encompassed by class level attributes to some extent.

We aimed at promoting healthy discussions among developers, where each one would be able to express his perception of the critical attributes completely. Therefore, we recruited only three active developers per team. We discuss data on the relevance degree of critical attributes through descriptive analysis. We also applied thematic synthesis (Cruzes and Dyba, 2011) procedures for aggregating and extracting major lessons on why each critical attribute is critical while evolving features. Finally, we crossed data of Chapter 3 and this chapter. Our study results suggest that:

– Two critical attributes are ultimately perceived as relevant for developers while evolving features: low class cohesion and high class complexity. Developers pointed out that both critical attributes potentially increase the difficulty for understanding and changing classes in different occasions. These occasions include software testing, fault tracking, feature additions. This is particularly interesting because popular refactorings (Murphy-Hill et al., 2012; Silva et al., 2016a), e.g. Extract Method, Move Method, and Pull Up Method, often worsen cohesion (e.g. see Table 3.4). Conversely, complexity is rarely sensitive to any of the 11 refactoring types investigated in our qualitative study.

– The other four critical attributes are not necessarily perceived as relevant by developers while evolving features: high class coupling, large class hierarchy depth, large class hierarchy breadth, and large size. Developers argued that using design patterns like Object Pool, implementing particularly complicated features, and reusing code via class hierarchy might lead to degraded code structure and design. Although these critical attributes have their impact on the internal software quality, they are acceptable in those circumstances. This results is curious because refactorings rarely worsen inheritance, and only a few refactoring types worsen coupling and size – e.g., Pull Up Attribute and Pull Up Method.

– Based on the reasons reported for considering each critical attributes as relevant or irrelevant, we found out that certain critical attributes are closely related. For instance, i) large class size may cause low class cohesion, ii) high class complexity may lead to high class coupling, iii) large class hierarchy breadth may increase class complexity, and so forth. This result could support the design of novel refactoring tools for enhancing code structures, especially those based on the optimization of critical attributes that depend on one another.

- We crossed our quantitative data of Chapter 3, regarding the refactoring effect on internal attributes, with those discussed in this chapter, regarding the developer's perception of critical attributes as relevant for evolving features. Thus, based on our empirical observations, we were able to derive a catalog of refactoring recommendations to help manage critical attributes during software evolution. These recommendations partially rely on data regarding critical attributes at the class level. Thus, generalizing our study results to critical attributes at other system levels requires further investigation.

We organized the remainder of this chapter as follows. Section 4.1 provides the study characterization. We describe the research problem, our research objectives, and the context of each case study. Section 4.2 describes the case studies design in detail. We justify each research question and explain our procedures for collecting and analyzing data. Section 4.3 summarizes our study results in order to address each research question. Section 4.4 discusses threats to the study validity. Section 4.5 concludes this chapter.

4.1 Study Characterization

This section introduces the reader to the overall study characterization. As aforementioned, we proposed an industry case study (Runeson and Höst, 2009) based on focus group sessions (Kontio et al., 2004) aimed at investigating the developer's perception of critical attributes that are relevant for evolving features. The remainder of this section presents our research problem statement (Section 4.1.1), objectives (Section 4.1.2), and context (Section 4.1.3).

4.1.1 Problem Statement

Many studies, e.g. (Bavota et al., 2015; Chaparro et al., 2014; Du Bois and Mens, 2003; Veerappa and Harrison, 2013), investigated the refactoring effect on internal attributes. These studies helped, to some extent, validate the benefits and drawbacks of refactoring application in terms of internal software quality. Unfortunately, the current knowledge in this topic was quite scarce and limited due to the strict design of past work. In fact, these studies analyzed only a few refactoring types, internal attributes, and systems under software evolution. Aiming at addressing this literature gap, we have introduced the large quantitative study (Fernandes et al., 2020) described in Chapter 3.

By extending the current knowledge on the refactoring effect of internal attributes, while comparing study results of the literature, we acquired a broader understanding on this topic. In particular, we revealed that 94% of (re-)refactorings are applied to code elements with critical attributes. Once 73% of refactorings constitute floss refactorings, i.e. they co-occur with feature additions and enhancements, we concluded that critical attributes are potentially relevant for evolving features. In other words, developers seem to be concerned with mitigating or fully addressing critical attributes in order to successfully add and enhance features.

Nevertheless, our results regarding the refactoring effect on each internal attribute are diverse. On the one hand, refactorings quite often enhance code structure and its design, thereby helping in managing critical attributes, regardless of the applied refactoring type (Sections 3.4.1 and 3.5.1). Thus, we confirm certain assumptions (Fowler, 2018) on the benefits of applying refactorings to the internal structure of the source code. On the other hand, in the case of floss refactorings – which often co-occur with feature additions and enhancements – 35-55% of refactorings keep unaffected the critical attributes (Table 3.5). More critically, also in the case of floss refactorings, 9-35% of refactorings worsen these attributes (Table 3.5).

The design of our quantitative study did not allow us to investigate the reasons behind such a high rate of refactorings that either keep unaffected or worsen internal attributes. We had two major hypotheses on these reasons. Our **first hypothesis** was that refactoring is generally insufficient to overcome complex degradation symptoms while evolving features. Recent studies (Bibiano et al., 2019; Bibiano et al., 2020; Yoshida et al., 2016) reinforce this hypothesis to some extent. Indeed, these studies suggest that refactorings rarely suffice for fully removing design smells – degradation symptoms whose detection may depend on combining two or more critical attributes.

Our **second hypothesis** was that developers only remove those degradation symptoms that really matter for evolving features. Thus, developers tend to postpone or discard the removal of other, less relevant, degradation symptoms. Unfortunately, empirical evidence that supports this assumption is scarce. Drawing a parallel among different degradation symptoms, a considerable amount of studies such as (Palomba et al., 2014; Taibi et al., 2017; Yamashita and Moonen, 2013) investigated the developer's perception of design smells relevance for evolving features. Still, we did not find similar studies in the context of critical attributes.

This chapter targets the research problem of characterizing how developers perceive a well-defined set of critical attributes as relevant (or irrelevant).

We investigate critical attributes associated with the five internal attributes largely debated in our quantitative study (Chapter 3). The set of internal attributes is composed of cohesion, complexity, coupling, inheritance and, size. The set of critical attributes is exclusively limited to the class level: low class cohesion, high class complexity, high class coupling, large class hierarchy depth, large class hierarchy breadth, and large class size.

We opted for discarding critical attributes at other system levels (e.g., methods) due to time constraints for performing case studies in the industry. Still, we acknowledge that classes are a fundamental component of the code structure of any system. Moreover, developers should constantly monitor and enhance the internal quality at the class level (Fowler, 2018; Lanza and Marinescu, 2006).

4.1.2 Research Objectives

We structured our research objectives based on the Goal Question Metric methodology (Basili and Rombach, 1988) as follows: *analyze* the perception of software developers; *for the purpose of* understanding how much (and why) critical attributes are relevant while evolving features; *with respect to* the relevance of critical attributes individually and relatively, reasons why each critical attribute becomes relevant, and interrelations of critical attributes; *from the viewpoint of* developers engaged in software evolution tasks; *in the context of* a Brazilian industry-academy initiative for R&D called ExACTa, two development teams, and two systems partially implemented in Java.

We designed and performed industry case studies based on focus group sessions. Case study is a research methodology for observing phenomena in their natural context (Runeson and Höst, 2009). It has been used to observe practitioners while performing various tasks, e.g. code review (Sadowski et al., 2018). Thus, case study is suitable for contextualized discussions of developers on evolving features. Focus group is a qualitative research method for extracting experiences from the participants (Kontio et al., 2004). This method was designed for promoting discussions and knowledge sharing among participants. Thus, focus group is suitable for understanding the developer's perception of critical attributes in the context of evolving features.

We also expect to provide developers with refactoring recommendations to help manage critical attributes that they typically find relevant for evolving features. Thus, developers may improve critical attributes of interest while preserving other internal attributes that may become critical over time. In addition, we expect to inspire study replications in other industry contexts – once generality of case studies tends to be limited.

4.1.3 Context

We decided to conduct our case study in the context of the Experimentation-based Agile Co-creation initiative for digital Transformation (ExACTa). ExACTa is an industry-academia R&D initiative launched in September 2019 by the Pontifical Catholic University of Rio de Janeiro (PUC-Rio) in cooperation with Petrobras. The latter is the largest Brazilian company operating in the energy, natural gas, and oil industries (Kalinowski et al., 2020). ExACTa developed and delivered dozens of software-based solutions, each targeting a particular business need. The selection of the development teams at ExACTa was by convenience, because the author of this doctoral thesis had a direct contact with one of the development project managers at that institution.

Our case study aims at investigating the relevance of critical attributes based on metrics computed for Java systems in our previous study (Chapter 3). As a result, we opted for selecting only teams using the Java programming language. We ended up selecting two cases for study: ADN and Smart Frete – see Section 4.2.2 for details. Due to the subjective nature of the qualitative data analyzed and discussed throughout this work (e.g., the developer's perception on their own source code), we kept the developers associated with the development of each system anonymous. Thus, we expect to preserve participants from any personal judgment on their perception.

The two cases followed a Lean R&D development approach (Kalinowski et al., 2020). Concerning the implementation, both cases use Git[2] for performing code review through pull requests, as well as Azure DevOps[3] for managing software development tasks. Finally, both systems rely on the Spring MVC Framework[4] for implementing Web systems using the Model-View-Controller (MVC) architectural pattern.

4.2 Case Study Design

This section introduces the case study design. Section 4.2.1 introduces our research questions. Section 4.2.2 describes each case and its respective subjects. Finally, Section 4.2.3 presents our data collection procedures.

[2]https://git-scm.com/systems
[3]https://azure.microsoft.com/pt-br/services/devops/
[4]https://spring.io/

4.2.1 Research Questions

We defined the three research questions below.

RQ$_1$: *What is the relevance degree of each critical attribute for evolving features from the developer's perception?* – Monitoring the internal software quality is essential to assure the longevity of systems (Lanubile and Visaggio, 1995; Mens et al., 2010). Thus, developers have largely adopted techniques for performing such monitoring (Chidamber and Kemerer, 1994; Fernandes et al., 2016b; Lanza and Marinescu, 2006). Degradation symptoms are usually an effective mechanism for spotting degraded code structure and design (Pantiuchina et al., 2018; Yamashita and Moonen, 2013). Among several types of degradation symptoms proposed so far, critical attributes emerge as means for capturing different aspects of internal software quality (Chávez et al., 2017; Fernandes et al., 2020). They also contribute to detect broader symptoms, e.g. design smells (Fowler, 2018; Lanza and Marinescu, 2006).

Each critical attribute in isolation is advertised as capable of revealing a particular extent of quality decay, such as low cohesion, high coupling, and large size (Chávez et al., 2017). Nevertheless, the current empirical knowledge on how relevant these critical attributes are for developers while evolving features is scarce if not non-existent. It may be true that developers tend to either mitigate or fully address only those critical attributes that hinder features from being added or enhanced in the system. As a result, knowing what critical attributes usually represent threats to evolving features could help recommending refactorings to help manage critical attributes properly.

Through RQ$_1$, we investigate two aspects of the relevance of critical attribute. First, we investigate the relevance degree of each critical attribute in isolation. In this case, we aim at understanding how important it is for developers to either mitigating or fully address a critical attribute to favor software evolution. Second, we investigate the relative relevance of the six critical attributes altogether: low class cohesion, high class complexity, high class coupling, large class hierarchy depth, large class hierarchy breadth, and large class size. Thus, we expect to understand what critical attributes have the highest priority when it comes to evolving features.

RQ$_2$: *What are the reasons behind considering a critical attribute as relevant for evolving features?* – Each critical attribute in isolation reflects a particular aspect of the internal software quality. Thus, one could hypothesize that some critical attributes are more relevant than others are while performing software evolution. That is, developers may easily decide to mitigate or fully address certain critical attributes, once they hamper feature additions or enhancements. On the other hand, there may be critical attributes that developers learned to live with, i.e. whose occurrence is acceptable given a potential benefit to evolving features – or the lack of time for addressing them.

For a critical attribute to become relevant strongly depends on how each developer perceives its advantages, drawbacks, and inherent conditions to software evolution. For instance, let us consider a development team responsible for evolving classes whose features are naturally complex due to intricate business rules. Thus, developers may expect – and accept – certain classes to be little cohesive, highly complex, or even too large. Still, developers may prioritize delivering new features rather than addressing critical attributes that have little impact on software evolution after all.

Via RQ$_2$, we aim at exploring what circumstances make certain critical attributes actually relevant for evolving features. In other words, we ask participants of each focus session group to argue why each critical attributes deserves special attention while either adding or enhancing features. This knowledge is particularly useful for supporting decision-making in development teams with too short time to delivering features. Indeed, there is usually hundreds of stakeholders' demands and operating environment systems for developers to worry about while performing software evolution (Lehman, 1980; Mens et al., 2010). Consequently, they may find certain critical attributes worth managing in the detriment of other critical attributes.

RQ$_3$: *How to assist developers with refactoring to help manage critical attributes while evolving features?* – An undisciplined application of changes eventually degrades the source code structure and its design. Developers often introduce degradation symptoms that may (or may not) make it harder to performing software evolution. Previous studies regarding other types of degradation symptoms (Palomba et al., 2014; Taibi et al., 2017; Yamashita and Moonen, 2013) provide us with sufficient evidence for speculating that certain critical attributes are less relevant than others while evolving features. Still, the empirical investigations driven by RQ$_1$ and RQ$_2$ may confirm this speculation in practical settings.

Understanding the refactoring effect on internal attributes is the first step towards assisting developers in managing critical attributes. We achieved this understanding through our quantitative study reported in Chapter 3. Characterizing how much (and why) critical attributes are relevant for evolving features is another step towards the same purpose (RQ$_1$ and RQ$_2$). With RQ$_3$, we aim at completing the cycle with refactoring recommendations to help manage critical attributes from the developer's perception when evolving software features, preserving other internal attributes from becoming critical.

We discuss in Section 2.5 that several techniques have been proposed to help manage degradation symptoms such as design smells (Bibiano et al., 2019; Chidamber and Kemerer, 1994; Fernandes et al., 2016b; Lanza and Marinescu, 2006), which are strongly associated with critical attributes (Section 2.2). A few of these techniques (Lin et al., 2016; Szőke et al., 2015a) rely on Search-Based Software Engineering (SBSE) techniques (Simons et al., 2015) for optimizing the improvement of anomalous metric values. Unfortunately, these tools lack the developer perspective on which critical attributes really concern developers while evolving features. Thus, they provide generic support rather than a support shaped to particular needs of software evolution.

4.2.2 Case and Subject Selection

As discussed in Section 4.1.3, we selected two cases from the ExACTa initiative for analysis. We describe below the particularities of each case.

Case A: *ADN* – The ADN case consists of designing, implementing, and testing a software solution for shipping logistics management purposes at Petrobras. Such management occurs through an integrated network of multiple systems, which allow practitioners to identify and cope with situations in which a chartered ship is not available for delivering a service. The proposed system heavily depends on calculus, e.g. with respect to payments and available fuel computation. Four Petrobras practitioners often participate with feedback and orientations during the software development process. They support the ExACTa team in SCRUM planning and review cycles whenever possible.

Case B: *Smart Frete* – The Smart Frete case consists of designing, implementing, and testing a software solution for handling with freight calculation. It is responsible for assisting practitioners at Petrobras in predicting freight prices to transport materials via road transport. The system relies on a large data set, including information of truck size, axes, weight of materials, distance between cities, and so forth. The proposed system also estimates fair prices to transport materials between refineries, Petrobras units, etc. Thus, this system is also heavily grounded in calculus and data processing.

Table 4.1 provides a general view on the participant background for each case. We collected the table data from an online Background Form, which we sent to participants minutes before the start of his respective focus group session. Appendix A presents the full form. Regarding the highest instruction degree, we observe that both cases have similar results. Although Case A has a PhD, both cases counted on the participation of one Master and on Specialist in knowledge areas related to Computer Science. Although participants of Case A have 4.33 years of experience in average against 8.33 years for Case B, participants in both cases have participated in the development of 5 systems in average. Thus, we assume the development expertise is balanced across cases.

Table 4.1: Participant Background Collected via Background Form

Question (Q)	**Case A**			**Case B**		
	A1	**A2**	**A3**	**B1**	**B2**	**B3**
Q1: Highest instruction degree	MSc	Specialist	PhD	Specialist	MSc	BS
Q2: Years of industry experience	5	2	6	5	10	10
Q3: Number of software projects	7	4	4	2	6	7
Q4: Familiarity with software metrics	Q4.c	Q4.c	Q4.b	Q4.d	Q4.c	Q4.d
Q5: Concerned with improving quality	Agree	Agree	Indifferent	Strongly agree	Agree	Strongly agree
Q6: Familiarity with internal attributes	Q6.e	Q6.c	Q6.b	Q6.d	Q6.d	Q6.d

Q4.b: I have heard about them but I am not so sure what they are
Q4.c: I have a general understanding, but do not use them in my software projects
Q4.d: I have a good understanding, and use them in my software projects sometimes
Q6.b: I have heard about them but I am not so sure what they are
Q6.c: I have a general understanding, but do not analyze them in my software projects
Q6.d: I have a good understanding, and analyze them in my software projects sometimes
Q6.e: I have a strong understanding, and analyze them in my software projects frequently

In our Background Form, participants were asked to report on their familiarity with software metrics (Q4). According to the data of Table 4.1, two participants of Case A said they have heard of metrics but are not really sure about what metrics mean (Q4.b), while one participant said to have a general understanding but does not use metrics in his projects (Q4.c). Case B participants claimed to be slightly more familiar with metrics: while one participant said to have a general understanding but not to use metrics (Q4.c), two participants said they have a good understanding and use metrics sometimes (Q4.d).

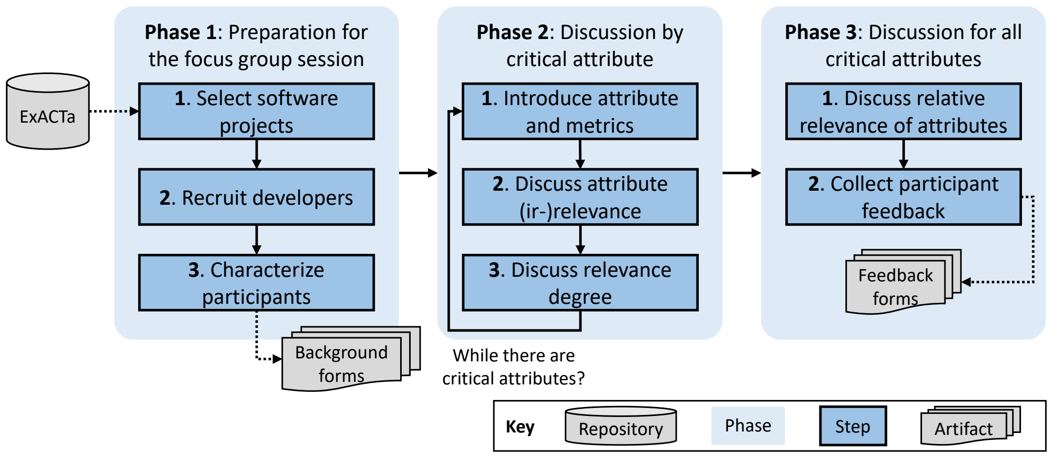
We also asked participants on how much they are concerned with improving the quality of source code in their systems (Q6). While participants of Case A simply agreed with this statement, participants of Case B showed to be significantly more concerned about this matter. Judging by the lower familiarity of participants in Case A with metrics, this result is reasonable.

Finally, we asked in the Background Form about the familiarity of participants with the concept of internal attributes. Once this term is rather academic than recurring in industry settings, we presented as example the five internal attributes assessed through this doctoral thesis: cohesion, complexity, coupling, inheritance, and size. Thus, we supported the participants by slightly driving their responses to the context of our work. Data of Table 4.1 showed some interesting results. Participants of Case A oscillated a lot in terms of expertise in internal attributes: one of them only heard about it (Q6.b), another one said to have a general understanding but not analyze internal attributes in his projects (Q6.c), and the last one said to have a strong understanding and analyze internal attributes frequently (Q6.e). One could say these participants know at least a little about internal attributes because of the metrics they are familiar with. Conversely, all participants of Case B said to have a good understanding and analyze internal attributes occasionally (Q6.d).

The bottom line is that participants of Case A are less familiar with internal quality and its management when compared to participants of Case B. Still, all participants are considerably experienced in software development.

4.2.3 Data Collection Procedures

Figure 4.1 depicts the procedures adopted for collecting data throughout the case study. We organized these procedures in three major phases: *Preparation for the focus group session*, *Discussion by critical attribute*, and *Discussion of all critical attributes*. We describe below each phase and procedure.



Phase 1: Preparation for the focus group session
ExACTa
1. Select software projects
2. Recruit developers
3. Characterize participants
Background forms
Phase 2: Discussion by critical attribute
1. Introduce attribute and metrics
2. Discuss attribute (ir-)relevance
3. Discuss relevance degree
While there are critical attributes?
Phase 3: Discussion for all critical attributes
1. Discuss relative relevance of attributes
2. Collect participant feedback
Feedback forms
Key
Repository
Phase
Step
Artifact

Figure 4.1: Data Collection Procedures

Phase 1: *Preparation for the focus group session* – This phase aimed at driving the collection of preliminary resources for supporting the execution of each focus group session. Besides defining what cases would be assessed, and recruiting participants to engage in discussions, we collected the background information. This phase is composed of the three procedures below.

1. *Select software projects* – As discussed in Section 4.1.3, we contacted a development project manager at ExACTa aimed at asking for software projects whose systems are implemented using (at least partially) the

Java programming language. This decision was taken to assure that participants are minimally familiar with concepts of object-oriented languages, which permeate all this work (e.g., the refactoring types and metrics explored here are mostly applicable to these languages). We ended up selecting two projects (Case A and Case B).

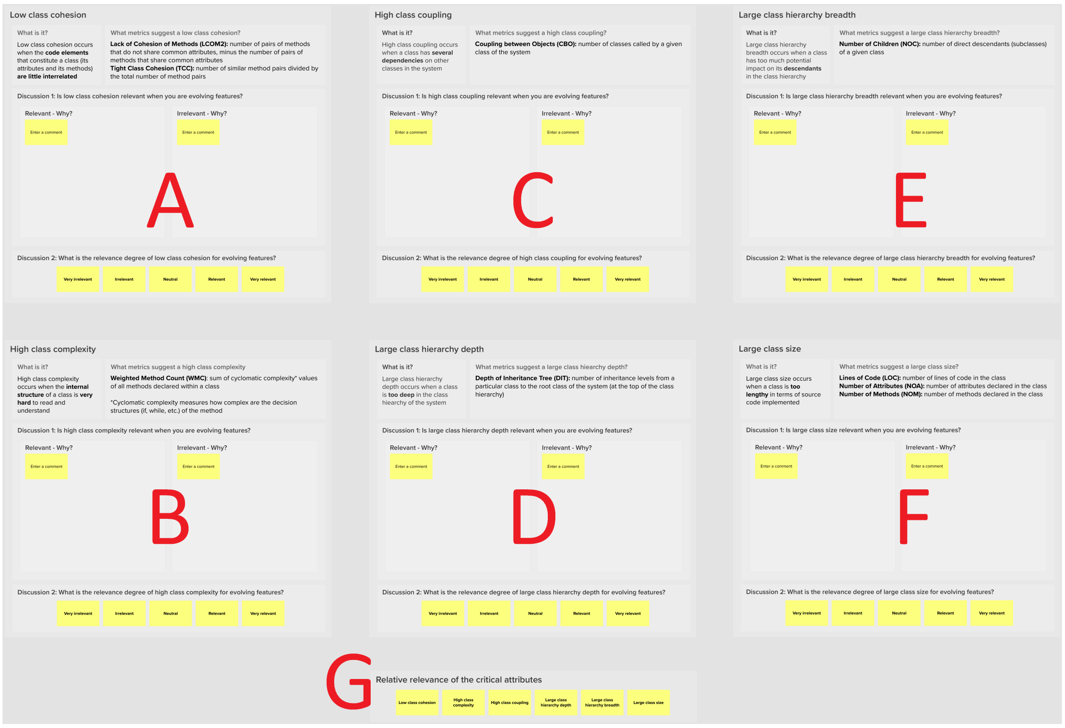
2. *Recruit developers* – For each project, we asked for permission to invite developers for participating in our study. Due to the intensively collaborative nature of ExACTa projects, both cases share developers, which contribute in the development of multiple systems. We then opted for selecting two independent sets of participants, one per system. No participant engaged in two focus group sessions, even though they may have contributed to the development of both systems. The project manager played an essential role in defining what developers are more active in the development of each system. We recruited three participants per case.

3. *Characterize participants* – We carefully designed and revised our Background Form aimed at collecting basic information on the participant expertise. Our major goal was profiling each case so we could better interpret our study results. We opted for a short and simple form in order to prevent participants from being tired or discouraged to participate in discussions right after filling the form. As shown in Table 4.1, we collected data on the participant instruction (Q1), experience with software development in industry (Q2 and Q3), familiarity with two key concepts of this work, i.e. software metrics (Q4) and internal attributes (Q6), and the concern degree of developers in improving code quality (Q5).

Before discussing **Phase 2** and **Phase 3**, we illustrate the online environment used for promoting discussions on critical attributes. Figure 4.2 depicts the virtual template that we carefully designed using the MURAL online tool[5]. The MURAL team kindly granted us with a free workspace at MURAL for Education program. Each session started with a mural as the template depicted in the figure. This mural has seven well-defined sections. Sections A to F aimed at driving the discussion regarding each critical attribute: low class cohesion, high class complexity, high class coupling, large class hierarchy depth, large class hierarchy breadth, and large class size. Section G aimed at driving the discussion on the relative relevance of all six critical attributes.

Figure 4.3 depicts only Section A, which was designed for aggregating all discussions on low class cohesion. Section A1 contains the short de-

[5]https://www.mural.co/



Low class cohesion

What is it?
Low class cohesion occurs when the **code elements** that constitute a class (its attributes and its methods) **are little interrelated**

What metrics suggest a low class cohesion?
Lack of Cohesion of Methods (LCOM2): number of pairs of methods that do not share common attributes, minus the number of pairs of methods that share common attributes
Tight Class Cohesion (TCC): number of similar method pairs divided by the total number of method pairs

Discussion 1: Is low class cohesion relevant when you are evolving features?
Relevant - Why? Enter a comment
Irrelevant - Why? Enter a comment

Discussion 2: What is the relevance degree of low class cohesion for evolving features?
Very irrelevant | Irrelevant | Neutral | Relevant | Very relevant

High class coupling

What is it?
High class coupling occurs when a class has **several dependencies** on other classes in the system

What metrics suggest a high class coupling?
Coupling between Objects (CBO): number of classes called by a given class of the system

Discussion 1: Is high class coupling relevant when you are evolving features?
Relevant - Why? Enter a comment
Irrelevant - Why? Enter a comment

Discussion 2: What is the relevance degree of high class coupling for evolving features?
Very irrelevant | Irrelevant | Neutral | Relevant | Very relevant

Large class hierarchy breadth

What is it?
Large class hierarchy breadth occurs when a class has too much potential impact on its **descendants** in the class hierarchy

What metrics suggest a large class hierarchy breadth?
Number of Children (NOC): number of direct descendants (subclasses) of a given class

Discussion 1: Is large class hierarchy breadth relevant when you are evolving features?
Relevant - Why? Enter a comment
Irrelevant - Why? Enter a comment

Discussion 2: What is the relevance degree of large class hierarchy breadth for evolving features?
Very irrelevant | Irrelevant | Neutral | Relevant | Very relevant

High class complexity

What is it?
High class complexity occurs when the **internal structure** of a class is **very hard** to read and understand

What metrics suggest a high class complexity
Weighted Method Count (WMC): sum of cyclomatic complexity* values of all methods declared within a class

*Cyclomatic complexity measures how complex are the decision structures (if, while, etc.) of the method

Discussion 1: Is high class complexity relevant when you are evolving features?
Relevant - Why? Enter a comment
Irrelevant - Why? Enter a comment

Discussion 2: What is the relevance degree of high class complexity for evolving features?
Very irrelevant | Irrelevant | Neutral | Relevant | Very relevant

Large class hierarchy depth

What is it?
Large class hierarchy depth occurs when a class is **too deep** in the class hiearchy of the system

What metrics suggest a large class hiearchy depth?
Depth of Inheritance Tree (DIT): number of inheritance levels from a particular class to the root class of the system (at the top of the class hierarchy)

Discussion 1: Is large class hierarchy depth relevant when you are evolving features?
Relevant - Why? Enter a comment
Irrelevant - Why? Enter a comment

Discussion 2: What is the relevance degree of large class hierarchy depth for evolving features?
Very irrelevant | Irrelevant | Neutral | Relevant | Very relevant

Large class size

What is it?
Large class size occurs when a class is **too lengthy** in terms of source code implemented

What metrics suggest a large class size?
Lines of Code (LOC): number of lines of code in the class
Number of Attributes (NOA): number of attributes declared in the class
Number of Methods (NOM): number of methods declared in the class

Discussion 1: Is large class size relevant when you are evolving features?
Relevant - Why? Enter a comment
Irrelevant - Why? Enter a comment

Discussion 2: What is the relevance degree of large class size for evolving features?
Very irrelevant | Irrelevant | Neutral | Relevant | Very relevant

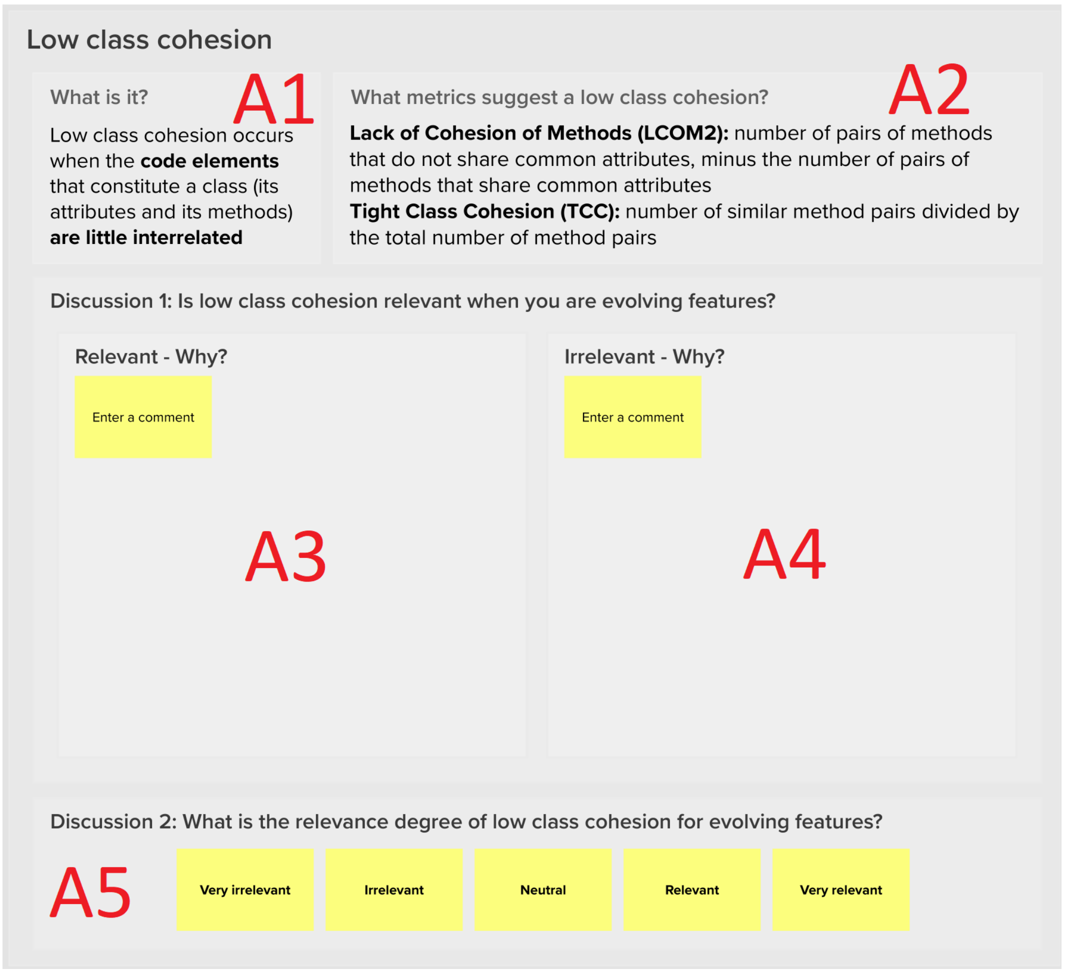
Relative relevance of the critical attributes
Low class cohesion | High class complexity | High class coupling | Large class hierarchy depth | Large class hierarchy breadth | Large class size

Figure 4.2: Template of Focus Group Session Defined at MURAL

scription of the critical attribute based on the literature (Chávez et al., 2017; Fernandes et al., 2020; Fowler, 2018; Lanza and Marinescu, 2006). Section A2 provides the participants with examples of metrics aimed at capturing the respective critical attribute. Section A3 was designed for developers to add notes on why the critical attribute is relevant for evolving features. Section A4 is similar but focused on why critical attributes may be irrelevant for evolving features. Finally, Section A5 is designed for capturing the relevance degree of the critical attribute based on a five-point scale: very irrelevant, irrelevant, neutral, relevant, and very relevant.

Phase 2: *Discussion by critical attribute* – This is the first phase associated with focus group session itself. During this phase, we collected all data regarding the developer's perception of critical attributes as relevant (or irrelevant) for evolving features. However, in this phase each critical attribute is discussed in isolation. We defined the three procedures below.

1. *Introduce attribute and metrics* – The discussion on each critical attribute starts with a brief theoretical introduction. We provided the participants with a short definition of the critical attribute based on the literature (Chávez et al., 2017; Fernandes et al., 2020; Fowler, 2018; Lanza and Marinescu, 2006). After that, we provided them with a few examples of metrics designed for capturing the respective internal attribute. We sample metrics arbitrarily based on our notion of metrics that



Low class cohesion

What is it?

Low class cohesion occurs when the **code elements** that constitute a class (its attributes and its methods) **are little interrelated**

What metrics suggest a low class cohesion?

Lack of Cohesion of Methods (LCOM2): number of pairs of methods that do not share common attributes, minus the number of pairs of methods that share common attributes
Tight Class Cohesion (TCC): number of similar method pairs divided by the total number of method pairs

Discussion 1: Is low class cohesion relevant when you are evolving features?

Relevant - Why?

Irrelevant - Why?

Discussion 2: What is the relevance degree of low class cohesion for evolving features?

Figure 4.3: Section Dedicated to Discussing Low Class Cohesion

developers could understand more easily. For instance, we opted for exemplifying Lack of Cohesion (LCOM2) (Chidamber and Kemerer, 1994) rather than LCOM3 (Li and Henry, 1993) because the latter implies explaining concepts like disjoint components in a graph.

2. *Discuss attribute (ir-)relevance* – We asked the participants to elicit reasons why each critical attribute is relevant (or irrelevant) for evolving features. Each reason should be documented as a note in the appropriate section: Section A3 for relevant and Section A4 for relevant (cf. Figure 4.3). From time to time, we reminded participants that evolving features include adding new features as much as enhancing existing features of the system. In addition, we constantly recommended participants to share knowledge and experiences surrounding each critical attribute, especially when discussions lost intensity. All participants we asked to collaborate with circumstances where mitigating or fully addressing a critical attribute is important for facilitating software evolution.

3. *Discuss relevance degree* – After discussing why a critical attribute is relevant (or irrelevant) for evolving features, the instructor of the focus session group promoted a recap. Each note on the (ir-)relevance of the critical attribute was read out loud. Whenever the instructor felt that a note is poorly written, he asked the participants to provide further considerations on the note. At the end of this procedure, we asked each participant to assign one vote to the relevance degree of the critical attribute according with the five-point scale depicted in Figure 4.3. Each participant voted individually, without knowing the votes of his colleagues until the discussion by critical attribute has finished.

Phase 3: *Discussion for all critical attributes* – After discussing each critical attribute in isolation, the focus session group ended with a discussion about the relative relevance of critical attributes. This phase has two procedures described below.

1. *Discuss relative relevance of attributes* – We asked participants to rank those critical attributes that matter the most while evolving features. Each participant received five votes in total. These votes were meant to be distributed throughout the six critical attributes: low class cohesion, high class complexity, high class coupling, large class hierarchy depth, large class hierarchy breadth, and large class size. We arbitrarily chose to assign five votes per participant in order to prevent them from assigning one vote for each critical attribute, thereby making it hard to conclude anything on the relative relevance.

2. *Collect participant feedback* – By the end of each focus group session, we asked participants to fill out a Feedback Form. We aimed at assuring that each developer felt confident and comfortable to discuss critical attributes together with their colleagues. We further discuss the results of this form in Section 4.6. Appendix B presents the full form.

Each focus group session was conducted online via a Zoom Meeting[6] chat. We kept video and audio records of all sessions to support the analysis of data provided by participants through our virtual workspace at MURAL. We often accessed the video and audio records for understanding what developers meant with each note. In addition, the records were very helpful in translating the notes from Brazilian Portuguese to English, once part of the participant's intent with a note may be lost in translation. The focus group sessions for Case A and Case B occurred in October 16, 2020 and October 20, 2020, respectively. Each session lasted no longer than two and a half hours.

[6]https://zoom.us/pt-pt/meetings.html

4.3 Results and Discussion

This section discusses our study results as follows. Section 4.3.1 elaborates on how developers perceive each critical attribute as relevant for evolving features. Sections 4.3.2 and 4.3.3 discuss the reasons why each critical attribute is pointed out as relevant or irrelevant by case study. Section 4.3.4 provides refactoring recommendations to help manage critical attributes while evolving features, based the crossing of quantitative (Chapter 3) and qualitative data. Section 4.3.5 summarizes participant feedback on the focus group sessions.

4.3.1 Relevance of Critical Attributes for Evolving Features (RQ$_1$)

Figure 4.4 depicts how many participants voted for a certain degree of relevance with respect to each critical attribute under investigation. We grouped the results by case: Case A data in the left and Case B in the right.

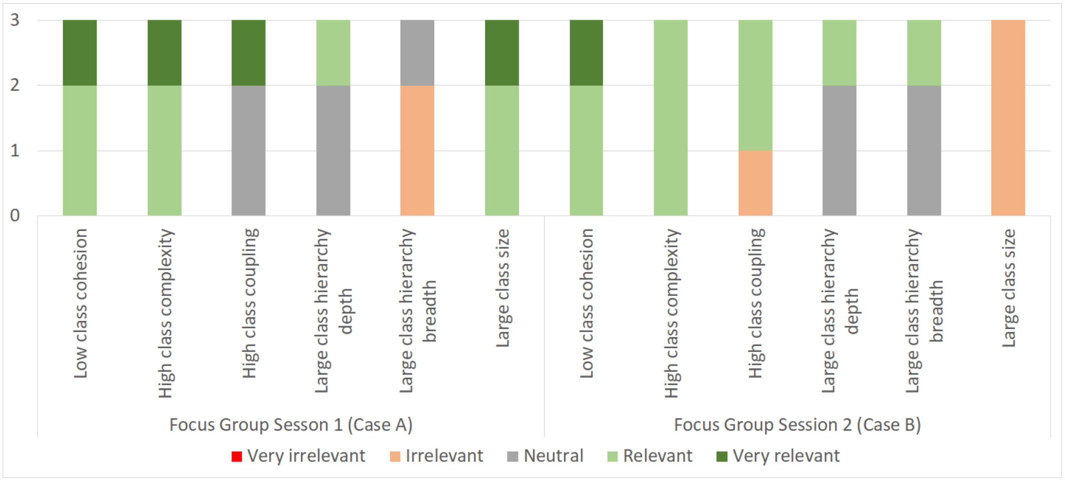


Figure 4.4: Relevance of Critical Attributes per Case

Regarding Case A, three critical attributes are ultimately perceived as relevant by the developers while evolving features: low class cohesion, high class complexity, and large class size. These are the only critical attributes for which no participant reported perceptions as either neutral or (very) irrelevant. According to background data of Table 4.1, we assume that participants of Case A are the less familiar with metrics and internal attributes. Thus, one could speculate that low class cohesion, high class complexity, and large class size are intuitive degradation symptoms – and potentially harmful to evolving features.

With respect to Case B, our study results changed considerably. First, only low class cohesion and large class complexity are reportedly relevant for evolving features. Curiously, in the opposite way of Case A, large class size was ultimately considered irrelevant while performing software evolution. None of the participants assigned the very irrelevant degree, but they also did not assign any relevant degree. This result may be associated with the fact that participants of Case B are the more familiar with metrics and internal attributes (Table 4.1). Maybe they are experienced enough to acknowledge that certain large classes are acceptable depending on factors such as the system domain and the inherent difficulty of a business rule.

Figure 4.5 depicts the overall developer's perception on how relevant each critical attribute is, regardless of the case. Two critical attributes are ultimately relevant for developers: low class cohesion and high class complexity. The other four critical attributes are not necessarily relevant for developers while evolving features: high class coupling, large class hierarchy depth, large class hierarchy breadth, and large size. Curiously, the aggregated data shows that developers are not exactly sure whether large class hierarchy depth. The high rate of neutral votes suggest that this critical attribute is not even an issue when debating and performing software evolution.

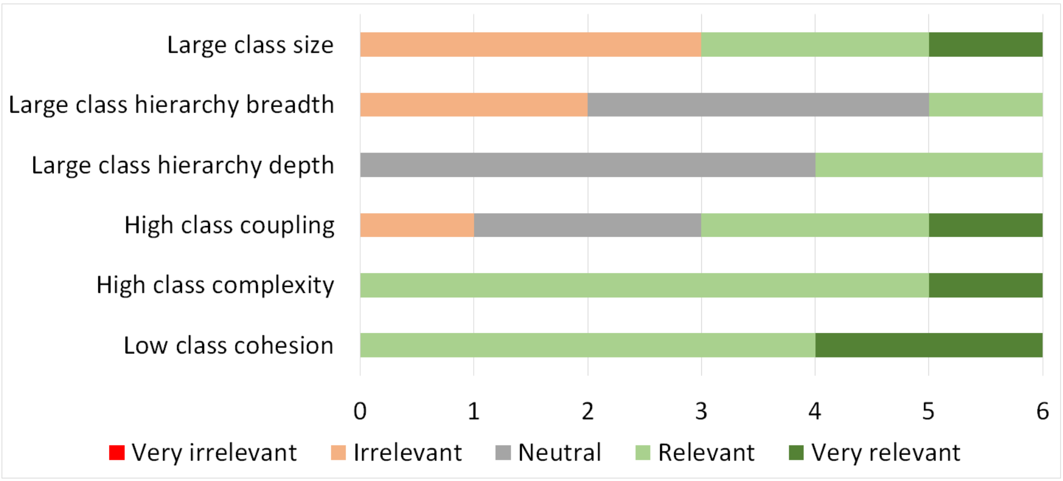


Figure 4.5: Relevance of Critical Attributes for Both Cases

4.3.2 Reasons Behind the (Ir-)relevance of Critical Attributes (RQ$_2$) – Case A

As discussed in Section 4.2.3 about **Phase 2** (*Discussion by critical attribute*), we asked participants of each focus session to provides us with reasons why each critical attribute is relevant (or irrelevant) for evolving features. We have collected these reasons through notes posted by the participants in the session's virtual mural (as depicted in Figure 4.3). Appendices E and F present prints of the mural with all notes provided by the participants (in Brazilian Portuguese) via MURAL platform.

Aimed at analyzing these reasons, we first transcribed all notes exactly as they were written by the participants into a spreadsheet. Based on our impressions as instructors of the focus group sessions, and after watching the video and audio records, we rewrote the notes aimed at fixing typos, filling communication gaps (e.g. omitted words), and making it explicit what is the target critical attributes of each note. Finally, we translated the notes from Brazilian Portuguese to English. Appendix C presents the full video transcription for the Focus Group Session 1 (Case A).

Overall Results for Case A: Table 4.2 lists the notes provided by participants on why each critical attribute is either relevant or irrelevant for evolving features. The first column refers to each critical attribute, while the second column distinguishes notes about the attribute relevance or irrelevance.

Large class size was the most discussed critical attribute in terms of number of notes, which is not surprising because it is a top-three most relevant attribute according to Figure 4.4. This is interesting the usefulness of size metrics for assessing the internal software quality is quite debatable (Briand et al., 2000; Fernandes et al., 2017a; Lanza and Marinescu, 2006; Li and Henry, 1993). On the one hand, participants reported, for instance that "large class size makes it hard to maintain source code" and "large class size increases error proneness of the source code," both topics discussed by previous work (Briand et al., 2000; Fernandes et al., 2017a). On the other hand, participants mentioned that "large class size is irrelevant when developers deal with urgency in program delivery" which, by the way, is a major issue presented in Chapter 1. Curiously, Participant C associated large class size with the system domain as suggested by the transcription quotes below:

"Does this comment mean that, sometimes, it is fine to keep large a class implementing several features which are hard to decouple?" – Applicant

"Yes! This scenario has something to do with program domain. It is harder to maintain two classes that are equivalent to a single class." – Participant C

The second most discussed critical attribute is low class cohesion, which is also a top-three most relevant attribute according to Figure 4.4. This is an attribute whose applicability in measuring internal quality has been shown in different development scenarios (Bieman and Kang, 1995; Chidamber and Kemerer, 1994). On the one hand, participants said that "high class cohesion facilitates source code reuse,", something that previous work have assessed (Bieman and Kang, 1995). Participants also said that "low class cohesion makes it hard to find errors," which is a recurring argument through the responses of both cases (Case A and Case B).

Table 4.2: Notes on (Ir-)relevance of Critical Attributes for Case A

Low class cohesion (9)	Rel. (7)	- High class cohesion facilitates source code reuse - High class cohesion helps optimizing the most used parts of the source code - High class cohesion may help understanding the source code - High class cohesion prevents code duplication - Low class cohesion makes it hard to find errors - Low class cohesion makes it harder to organize code than usual - Non-cohesive classes tend to larger than necessary
	Irrel. (2)	- Low class cohesion is irrelevant when the time to delivering code is short - Low class cohesion is irrelevant when you are fixing bugs in legacy code
High class complexity (6)	Rel. (5)	- High class complexity leads to high class coupling - High class complexity makes it hard to perform program testing - High class complexity makes it hard to understand and evolve code - High class complexity makes it hard to understand and maintain code - High class complexity may cause unnecessary slowness, thereby harming the class performance
	Irrel. (1)	- High class complexity is irrelevant when having a complex method is necessary
High class coupling (6)	Rel. (3)	- Class coupling makes the source code rigid and hard to evolve - High class coupling makes it dangerous to update method signatures - High class coupling may favor error propagation
	Irrel. (3)	- High class coupling is acceptable when coding a highly coupled entity of the Entity Relationship Diagram - High class coupling is irrelevant when adopting the Object Pool pattern - Keeping the high coupling of a class for a while may speed up programming
Large class hierarchy depth (7)	Rel. (4)	- Large depth is relevant because changes at the highest hierarchy levels may affect all classes at the lowest levels - Large depth is relevant when a subclass is unnecessarily specialized - Large depth is relevant when the class must implement several abstract methods inherited from other classes - Large depth makes it hard to understand what can be implemented at the subclass level
	Irrel. (3)	- Class hierarchy is an advantageus feature of object-orientation - Large depth is irrelevant when a subclass is necessarily specialized - Large depth is irrelevant when it allows reusing reuse located at the highest hierarchical levels
Large class hierarchy breadth (6)	Rel. (2)	- Large breadth is relevant because errors affecting the parent class impact a considerable part of the source code - Large breadth may increase the class complexity
	Irrel. (4)	- Large breadth allows defining generic code that benefits child classes - Large breadth helps distributing program features - Large breadth is irrelevant because several subclasses may reuse methods provided by the parent class - Large breadth is irrelevant when reusing properties used by all entities of the Entity Relationship Diagram
Large class size (10)	Rel. (6)	- Large class size increases error proneness of the source code - Large class size is relevant because classes should be small whenever possible - Large class size makes it hard to maintain source code - Large class size makes it hard to organize source code - Large class size makes it hard to perform program testing - Large class size makes it hard to understand source code
	Irrel. (4)	- Large class size is irrelevant when a class has several methods that are hard to decouple - Large class size is irrelevant when developers deal with urgency in program delivery - Large class size is irrelevant when it affects Proof of Concept (PoC) classes - Large class size is irrelevant when the methods are naturally very complex

On the other hand, participants discussed that "low class cohesion is irrelevant when the time to delivering code is short," which has been discussed with respect to large class size as well. Curiously, in the particular case of low class cohesion, participants seem to have more arguments on the relevance of this critical attributes for evolving features.

The remaining critical attributes were also significantly discussed through this focus group session. Large class hierarchy depth had many notes regarding its relevance for evolving features, as well as high class complexity, high class coupling, and large class hierarchy breadth. Curiously, high class complexity had the highest number of notes on its relevant while evolving features; this results is interesting because high class complexity is the last top-three most critical attribute according to Figure 4.4. Participants showed different perspectives on large class hierarchy breadth, as suggested by these quotes:

"Participant B, I disagree when you say it: 'large hierarchy breadth may increase class complexity'. What do you mean?" – Participant A

"What do I mean? I see no problem with large hierarchy depth, but with large hierarchy breadth. Large breadth may increase class complexity, because several features come from a single source. Right?" – Participant B

"Is not that good?" – Participant A

"This is polymorphism. I am not saying this is bad, but... Look, I have added a comment to 'Irrelevant – Why?' saying that large breadth may help decouple and split program features. Still, I see no negative effect of large hierarchy depth." – Participant B

Regarding the two critical attributes associated with inheritance (namely, large class hierarchy depth and large class hierarchy breadth), participants tended to discuss irrelevance by means of the practical usefulness of class hierarchies in a system. Examples of quotes that illustrate this issue are "large depth is irrelevant when it allows reusing code located at the highest hierarchical levels" and "large breadth is irrelevant when reusing properties used by all entities of the Entity Relationship Diagram." Quotes like these justify, at least in parts, why the majority of participants assigned a neutral or irrelevant degree for the relevance of both attributes (cf. Figure 4.4).

Is is worth reminding that, during each focus group session, we constantly stimulated participants to report as many aspects of either relevance or irrelevance by critical attribute. Judging by the considerable number of arguments favor and against the relevant of all attributes, we concluded that our effort in promoting a healthy and productive discussion among participants paid off.

Thematic Synthesis Results for Case A: As briefly discussed in Chapter 1, we have applied thematic synthesis (Cruzes and Dyba, 2011) procedures for aggregating and extracting major lessons on why each critical attribute is critical while evolving features. We performed these procedures for each case (Case A and Case B). Shortly, we have adopted the following procedures.

First, we separated all translated notes regarding the relevance of all critical attributes altogether from those regarding irrelevance. Second, for each set of notes (relevant and irrelevant), we arbitrarily grouped the notes according to their core theme, i.e. fine-grained themes discussed throughout each note. Third, we grouped these core themes into macro-themes, i.e. themes that are more comprehensive than the core themes. Fourth, we separated those macro-teams in two categories: *Code Structure and Design* regards aspects of the code structure and design of a system and *System Functionality* regards aspects of the features realized by a system.

Figure 4.6 is a tree that depicts our results for the thematic synthesis procedures applied to notes on why critical attributes are *relevant* for evolving features. The root note of the tree corresponds to the major theme, i.e. the relevance of critical attributes altogether. The first intermediate level corresponds to the two categories mentioned above (Code Structure and Design and System Functionality). The second intermediate level corresponds to the macro-themes. We assigned in brackets the critical attributes associated with each micro-theme or macro-theme whenever the node corresponds to a leaf from the tree – i.e. the node has no variants. We have found seven macro-themes, four of them associated with code structure and its design. The leaves correspond to the micro-themes, which are nine in total.

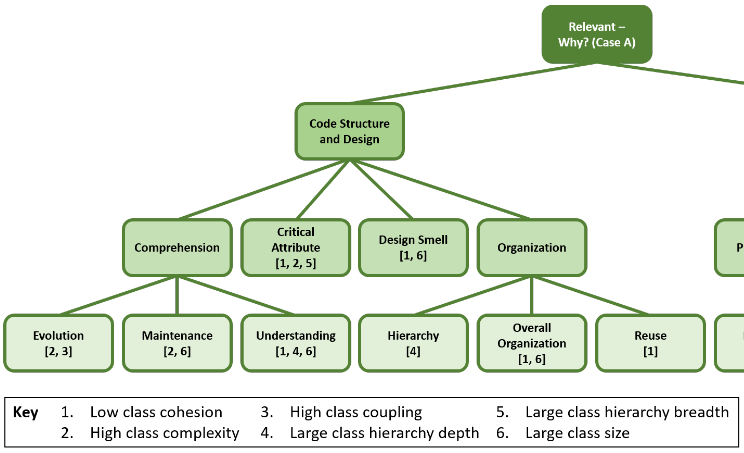


Figure 4.6: Themes on Why Attributes are Relevant for Case A

With respect to Code Structure and Design, participants mentioned that critical attributes in general are relevant for: *Comprehension*, i.e., the ability of reading and understanding the code elements; *Critical Attribute*, i.e. analyzing and reasoning about other critical attributes; *Design Smell* i.e. assessing or reasoning about the occurrence of Fowler-like design smells (Fowler, 2018); and *Organization*, i.e. the way how code elements are organized within the source code structure. The micro-themes have quite intuitive names, but it is worth saying *Overall Organization* includes those aspects not associated with *Hierarchy* and not closely associated with *Reuse*.

Regarding System Functionality, participants said that critical attributes are relevant for: *Change Propagation*, i.e. critical attributes may spot cases in which certain changes are unexpectedly or undesirably propagated throughout the system; *Failure*, i.e. the occurrence of bugs, faults, or failures (Ferreira et al., 2018); and *Performance*, i.e. aspects of the system performance such as the speed to respond to requests. All micro-themes have intuitive naming, so there is no need to discuss them.

Complementarily, Figure 4.7 is a tree of themes on why critical attributes are *irrelevant* while evolving features (cf. tree root). The first intermediate level corresponds to the categories Code Structure and Design and System Functionality. The second intermediate level corresponds to seven macro-themes identified, four of them associated with system functionality. The leaves are the five micro-themes identified. We assigned in brackets the critical attributes associated with each micro-theme or macro-theme whenever the node corresponds to a leaf from the tree – i.e. the node has no variants.

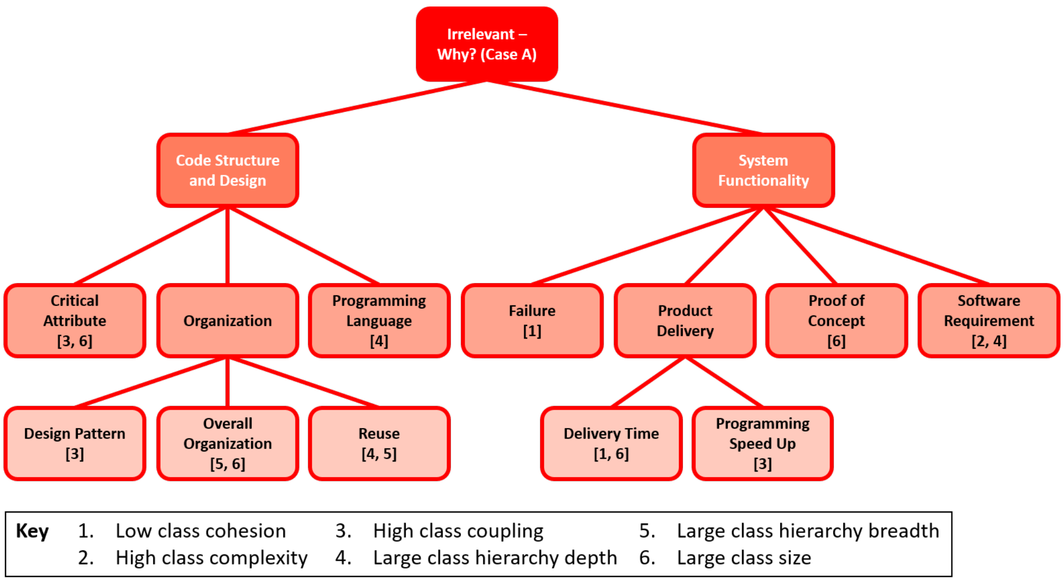


Figure 4.7: Themes on Why Attributes are Irrelevant for Case A

With respect to Code Structure and Design, participants mentioned that critical attributes in general are irrelevant for: *Critical Attribute*, i.e. analyzing and reasoning about other critical attributes; *Organization*, i.e. the way how code elements are organized within the source code structure; and *Programming Language*, i.e. aspects derived from the syntax, structure, and features provided by the programming used during software evolution. The micro-themes have quite intuitive names, but we highlight that *Design Pattern* refers to Gamma-like design patterns (Gamma et al., 1993). The participants argued that adopting certain patterns might lead to critical attributes. Despite of their theoretical or practical impact on the internal software quality, the critical attributes are either irrelevant or impossible/unfeasible to manage.

Regarding System Functionality, participants reported that critical attributes are irrelevant for: *Failure*, i.e. certain circumstances associated with bug fixing – in this case, when "fixing bugs in legacy code"; *Product Delivery*, i.e. aspects of delivering a system; *Proof of Concept*, i.e. aspects associated with implementation of source code particularly aimed at proving concepts to stakeholders during the iterative development cycles of agile processes like Lean R&D; and *Software Requirements*, i.e. requirements in general.

Via the **Critical Attribute** macro-attribute, we have found interesting insights on the interrelation between different critical attributes.

With respect to *relevance*, participants of Case A said that: i) "high class complexity leads to high class coupling," ii) "large breadth may increase the class complexity," and iii) "non-cohesive classes tend to be larger than necessary." In summary, we found three tuples of interrelations: i) (high class complexity, high class coupling), ii) (large class hierarchy breadth, high class complexity), and iii) (low class cohesion, large class size), respectively. This interesting result could support the design of novel refactoring tools for enhancing code structures, especially those based on the optimization of critical attributes that depend on one another.

Regarding *irrelevance*, participants of Case A said that: i) "high class coupling is acceptable when coding a highly coupled entity of the Entity Relationship Diagram" and ii) "large class size is irrelevant when the methods are naturally very complex." Thus, we found two tuples of interrelations: i) (high class coupling, high entity coupling) associating attributes at the levels of class and Entity Relationship (ER) model, and ii) (large class size, high method complexity) associating attributes at the levels of class and method, respectively. Again, these results could drive the design of novel refactoring tools for enhancing code structure and its design.

4.3.3 Reasons Behind the (Ir-)relevance of Critical Attributes (RQ$_2$) – Case B

Overall Results for Case B: For consultation purposes, Appendix D presents the full video transcription for the Focus Group Session 2 (Case B). Table 4.3 lists the notes on why each critical attribute is either relevant or irrelevant for evolving features.

Large class size tied with high class coupling as the critical attributes

Table 4.3: Notes on (Ir-)relevance of Critical Attributes for Case B

Low class cohesion (6)	Relev. (3)	- Low class cohesion makes it hard to maintain a program - Low class cohesion makes it hard to track errors - Low class cohesion makes it hard to evolve a program
	Irrel. (3)	- Low class cohesion is irrelevant if it affects an utility class - Low class cohesion is irrelevant in very small programs - Low class cohesion is irrelevant if the methods share concepts although they do not share attributes/parameters
High class complexity (4)	Relev. (3)	- High class complexity makes it hard to maintain a program - High class complexity makes it hard to understand program features - High class complexity makes it hard to implement new business rules
	Irrel. (1)	- High class complexity is irrelevant if affecting a class with a single author and recently maintained
High class coupling (7)	Relev. (4)	- High class coupling is relevant when implementing fault tolerance/error handling - High class coupling is relevant when CBO is high but the class is coupled with classes at different program levels - High class coupling is relevant when affecting non-cohesive classes - High class coupling is relevant when the class uses variable of concrete types rather than variables of interfaces
	Irrel. (3)	- High class coupling is irrelevant when CBO is high but the class is coupled with classes at the same program level - High class coupling may support source code reuse - High class coupling is irrelevant when the class is highly cohesive
Large class hierarchy depth (5)	Relev. (3)	- Large depth makes it hard to know where to implement a new program feature - Large depth makes it hard to understand the source code structure - Large depth may cause code duplication
	Irrel. (2)	- Large depth is irrelevant when DIT is inherited from libraries, especially stable libraries - Large depth is irrelevant because high DIT is rare
Large class hierarchy breadth (3)	Relev. (2)	- Large depth makes large breadth worse - Large breadth is relevant if child classes redefine the concrete behavior inherited from their parent class
	Irrel. (1)	- Large breadth is irrelevant if child classes do not redefine the concrete behavior inherited from their parent class
Large class size (7)	Relev. (2)	- Large size is relevant if the programming screen size is small - Large size rarely occurs in isolation
	Irrel. (5)	- Large size is irrelevant if the affected class is not complex - Large size rarely occurs in isolation - Large size is irrelevant if the methods are cohesive - Large size is irrelevant in utility classes - Large size is irrelevant if the integrated development environment (IDE) can collapse large source code blocks

with the highest number of notes. Curiously, these critical attributes were the only ones to have at least one vote for irrelevant (Figure 4.4). As previously discussed with respect to Case A, the usefulness of size metrics for assessing the internal software quality has been debated by previous work with mixed opinions (Briand et al., 2000; Fernandes et al., 2017a; Lanza and Marinescu, 2006; Li and Henry, 1993). This debate is reflected by the highest number of votes for irrelevant from the entire case study (Figure 4.5). On the one hand, participants said that "large size is relevant if the programming screen size is small" and because "large size rarely occurs in isolation" in terms of problems associated with internal software quality. On the other hand, participants also said that "large size is irrelevant in utility classes" and "large size is irrelevant if the integrated development environment (IDE) can collapse large source code blocks." These comments particularly suggest that large class size a problem of the development environment rather than a system problem.

Regarding high class coupling, participants also showed different perspectives on relevance and irrelevance. On the one hand, participants reported that "high class coupling is relevant when implementing fault tolerance/error handling." They also were very specific on the metrics used for computing this critical attribute, with statements like "high class coupling is relevant when CBO is high but the class is coupled with classes at different program levels." CBO is the Coupling between Objects metric of Table 2.1, which we displayed in our virtual mural during the focus session group for exemplification. By the way, with "program level" the participants clarified that they refer to a system package or module. On the other hand, participants said that "high class coupling [is irrelevant because it] may support source code reuse" and, in opposition to the previous case, "high class coupling is irrelevant when CBO is high but the class is coupled with classes at the same program level." I is worth mentioning that, although is has mostly seen as relevant for evolving features (Figure 4.4, this critical attribute receive one vote for irrelevant.

The third most discussed critical attributes is low class cohesion. Participants of Case B reported that "low class cohesion makes it hard to maintain a program" and "low class cohesion makes it hard to track errors." On the other and, from the developer's perception, "low class cohesion is irrelevant if it affects a utility class" and "low class cohesion is irrelevant in very small programs," for instance. The quotes below, extracted from the video transcription, suggest different perspectives for the same issue:

"Certain classes host features that fit no other part of the program – they are the so-called utility classes. Examples are classes responsible for handling objects, converting or editing data, counting... all useful in many part of the program. In these cases, low class cohesion is irrelevant." – Participant B

"I would easily find the best place to add features in a cohesive class. Web programs usually have a controller layer [...]. Adding the new feature to the controller layer is reasonable if the feature validates data forms, for instance. If classes within the controller layer are cohesive, I will know where to add the feature." – Participant B

"I have been thinking of small programs that evolve little. Maintaining this program will require little changes once the program has only a few features. The program will grow little and, then, low class cohesion is irrelevant." – Participant A

Comments like these suggest that the relevance of low class cohesion strongly depends on what the system implements. If the system is too simple or the class provides features to the whole system, low class cohesion is acceptable. Regardless of that, this critical attribute is curiously the one with most votes for relevant (with one vote for very relevant) – according to data of Figure 4.5.

The other three critical attributes – high class complexity, large class hierarchy depth, and large class hierarchy breadth – were less discussed in comparison with the same attributes in Case A. Curiously, the overall perception of developers in Case B are quite different and valuable. Regarding the attribute relevance, participants report that “high class complexity makes it hard to implement new business rules,” “large depth makes it hard to know where to implement a new program feature,” and “large breadth is relevant if child classes redefine the concrete behavior inherited from their parent class.” These notes add up as they confirm how different critical attributes may hinder feature additions and enhancements, which are the basis of software evolution.

We present below some quotes that illustrate the discussion above:

1) *“High class coupling is relevant while implementing fault tolerance. I mean, highly coupled classes depend on many others to realize their features and of those classes many have problems of different natures. It implies handling with different faults.”* – Participant B

2) *“Do you see cases in which deciding where in the class hierarchy to add a new feature is challenging? It may be necessary to duplicate code.”* – Applicant

“Yes, I may have to duplicate source code because of that.” – Participant B

“The most relevant comment so far is the one Participant B added about how hard is to know where to add a new feature.” – Participant A, later on

3) *“Large class hierarchy breadth is irrelevant if the child classes cannot change features inherited by the parent class, because the implementation focuses on the parent class. Otherwise, large breadth becomes relevant.”* – Participant B

We highlight each of the three aforementioned attributes (high class complexity, large class hierarchy depth, and large class hierarchy breadth) received at least one vote for relevant – see Figure 4.4. It is worth mentioning that the two critical attributes regarding inheritance received neutral votes. This results suggest that, for developers considerably concerned on internal software quality (Table 4.1), inheritance is not a major concern.

Thematic Synthesis Results for Case B: Figure 4.8 is a tree showing results for the thematic synthesis procedures applied to notes on why critical attributes are *relevant* during software evolution. The root note of the tree corresponds to the major theme, that is, the irrelevance of critical attributes altogether. The first intermediate level corresponds to the two major categories of themes: Code Structure and Design and System Functionality. The second intermediate level corresponds to the macro-themes. We have derived seven macro-themes where the majority (five of them) is associated with code structure and its design. The leaves are the seven micro-themes found in total. We assigned in brackets the critical attributes associated with each micro-theme or macro-theme whenever the node corresponds to a leaf from the tree – i.e. the node has no variants.

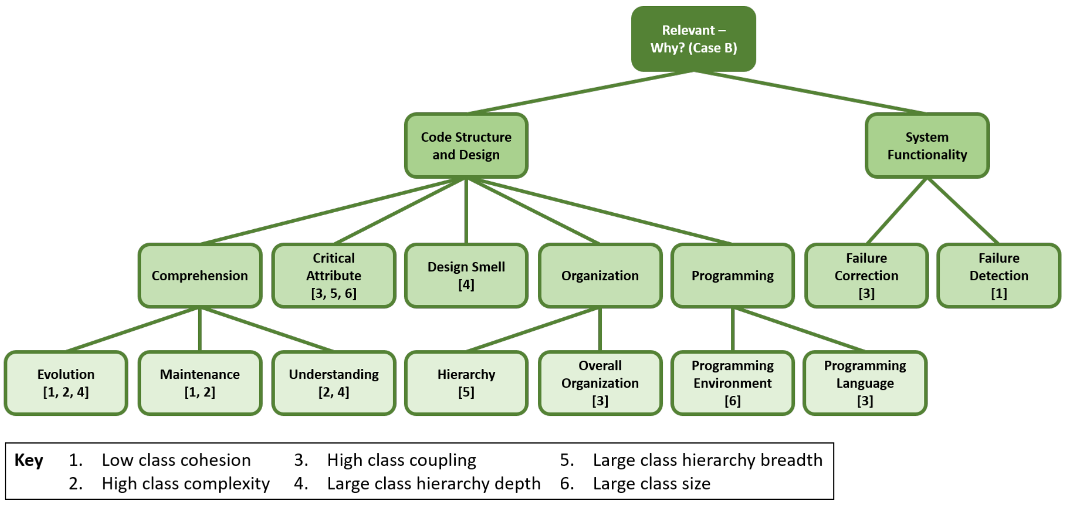


Figure 4.8: Themes on Why Attributes are Relevant for Case B

With respect to Code Structure and Design, participants mentioned that critical attributes in general are relevant for: *Comprehension*, i.e., the ability of reading and understanding the code elements; *Critical Attribute*, i.e. analyzing and reasoning about other critical attributes; *Design Smell* i.e. assessing or reasoning about the occurrence of Fowler-like design smells (Fowler, 2018); *Organization*, i.e. the way how code elements are organized within the source code structure; and *Programming*, i.e. aspects of programming a system that may affect how the internal software quality is perceived. All micro-themes received intuitive names with no need for further explanation.

Regarding System Functionality, participants reported only two macro-themes; *Failure Correction*, i.e. the ability to fix bugs, faults, or failures affecting the system behavior; and *Failure Detection*, i.e. the task of tracking unexpected system behaviors realized by bugs, faults, or failures in a system.

As a complement to the discussion above, Figure 4.9 depicts the themes derived the topic of why critical attributes are *irrelevant* while evolving features (see the tree root). The first intermediate level corresponds to the categories

Code Structure and Design and System Functionality. The second intermediate level corresponds to five macro-themes identified, three of them associated with code structure and its design. The leaves correspond to four micro-themes. We assigned in brackets the critical attributes associated with each micro-theme or macro-theme whenever the node corresponds to a leaf from the tree – i.e. the node has no variants.

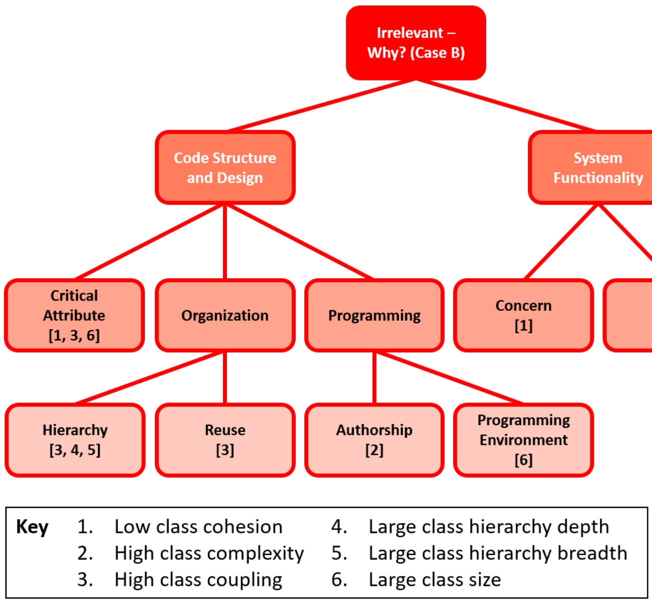


Figure 4.9: Themes on Why Attributes are Irrelevant for Case B

With respect to Code Structure and Design, participants reported that critical attributes in general are relevant for: *Critical Attribute*, i.e. analyzing and reasoning about other critical attributes; *Organization*, i.e. the way how code elements are organized within the source code structure; and *Programming*, i.e. aspects associated with activity of programming a system. All micro-themes have intuitive names. Still, it is worth mentioning that *Authorship* refers to the source code authorship as discussed in past work (Avelino et al., 2019).

Regarding System Functionality, participants reported that critical attributes are irrelevant in cases associated with: *Concern*, i.e. nature of the features realized by the system, pretty much as discussed in the field of software requirements (Chung and do Prado Leite, 2009); and *Utility*, i.e. classes of the system that serve as feature providers to the majority of the system – also known as utility classes.

By investigating the **Critical Attribute** macro-attribute, we derived interesting insights on the interrelation between different critical attributes.

With respect to *relevance*, participants of Case B said that: i) "high class coupling is relevant when affecting non-cohesive classes," ii) "large depth makes large breadth worse," and iii) "large size rarely occurs in isolation." Based on these quotes, we identified three tuples of interrelations: i) (high class coupling, low class cohesion), ii) (large class hierarchy depth, large class hierarchy breadth), and iii) (large class size, any critical attribute), respectively. We did not find any similarities with the interrelations of Case A and Case B.

Regarding *irrelevance*, participants of Case B said that: i) "high class coupling is irrelevant when the class is highly cohesive," ii) "large size is irrelevant if the affected class is not complex," iii) "large size is irrelevant if the methods are cohesive," iv) "large size rarely occurs in isolation," and v) "low class cohesion is irrelevant in very small programs." Thus, we found five tuples of interrelations: i) (high class coupling, low class cohesion), ii) (large class size, high class complexity), iii) (large class size, low method cohesion), iv) (large class size, any critical attribute), and v) (low class cohesion, high system size), respectively. Once again, we could not find any similarities with the interrelations of Case A and Case B.

4.3.4 Managing Critical Attributes through Refactorings (RQ$_3$)

We crossed our quantitative data of Chapter 3, regarding the refactoring effect on internal attributes, with those discussed in this chapter, regarding the developer's perception of critical attributes as relevant for evolving features. Thus, we were able to derive a simple, but empirically derived, catalog of refactoring recommendations to help manage critical attributes during software evolution. We discuss these recommendations below.

Table 4.4 summarizes data discussed in Sections 4.3.2 and 4.3.3 on the critical attributes that are interrelated from the developer's perception.

Table 4.4: Relevance and Interrelation of Critical Attributes from the Developer's Perception

Critical Attribute	**Relevant?**			**Interrelated Critical Attributes**					
	Case A	**Case B**	**All Cases**	**Low class cohesion**	**High class complexity**	**High class coupling**	**Large class hierarchy depth**	**Large class hierarchy breadth**	**Large class size**
Low class cohesion	X	X	X			X			X
High class complexity	X	X	X			X		X	X
High class coupling		X	X	X	X				X
Large class hierarchy depth								X	X
Large class hierarchy breadth					X		X		X
Large class size	X		X	X	X	X	X	X	

The first column lists all six critical attributes. The second to fourth columns mark those cases (Case A, Case B or both) where each critical attribute was voted as either relevant or very relevant by half of participants – three participants for isolated cases and six for all cases combined. We did not consider neutral votes. The remaining columns mark those critical attributes that interrelate with others. We considered interrelations only between critical attributes at the class level. Thus, we discarded any interrelations at system or method levels reported by the participants. This table will support the understanding of the following discussion.

Table 4.5 provides refactoring recommendations to help manage each of the six critical attributes. We derived this table by analyzing whether each refactoring type (columns two to twelve) tends to improve or worsen the internal attributes associated with each critical attribute. It is worth mentioning that we only considered data derived from the Most Metrics analysis approach, i.e. the strictest one. In addition, both large class hierarchy depth and breadth are associated with the same internal attribute, i.e. inheritance. The crossing of our quantitative and qualitative data is the key to understand whether developers could apply a particular refactoring type without major side effects to the internal software quality perceived as relevant by developers when evolving features. We discuss below some examples of how these recommendations based on the data crossing occurs.

Table 4.5: Refactoring Recommendations per Critical Attribute

	Is This Refactoring Type Recommended?										
Critical Attribute	**Extract Interface**	**Extract Method**	**Extract Superclass**	**Inline Method**	**Move Attribute**	**Move Method**	**Pull Up Attribute**	**Pull Up Method**	**Push Down Attribute**	**Push Down Method**	**Rename Method**
Low class cohesion		N	Y	Y	Y	N	N	N	N	N	
High class complexity		Y									
High class coupling		N		N		N	N	N	Y	Y	
Large class hierarchy depth	Y										
Large class hierarchy breadth	Y										
Large class size		N	Y	Y		Y	N	N	Y	N	

Y: Yes, because refactoring often improves the internal attribute
N: No, because refactoring often worsens the internal attribute

As discussed in Section 4.3.1, two critical attributes are ultimately perceived as relevant by developers while evolving features: low class cohesion and high class complexity. Developers pointed out that both critical attributes potentially increase the difficulty for understanding and changing classes in different occasions. These occasions include software testing, failure tracking, and feature additions. This is particularly interesting because popular refactorings (Murphy-Hill et al., 2012; Silva et al., 2016a), e.g. Extract Method, Move

Method, and Pull Up Method, often worsen cohesion (e.g. see Table 3.4). However, as our qualitative data suggests (Table 4.5), developers should take into account any side effects on high class coupling and large class size, which are possibly interrelated to low class cohesion.

Our quantitative data also suggested that only Extract Method often improves complexity and, therefore, could be used for addressing high class complexity. In addition, the data shows that complexity is rarely sensitive to any of the 11 refactoring types investigated in our qualitative study (Table 3.4). However, this critical attribute is closely related with three other attributes: high class coupling, large class hierarchy breadth, and large class size. Consequently, applying refactorings for addressing high class complexity may be non-trivial because developers must take into account any side effects on those three critical attributes.

The other four critical attributes tend to be irrelevant for developers (Section 4.3.1): high class coupling, large class hierarchy depth, large class hierarchy breadth, and large size. Developers argued that using design patterns like Object Pool, implementing particularly complicated features, and reusing code via class hierarchy might lead to degraded code structure and design. Although these critical attributes have their impact on the internal software quality, they are acceptable in those circumstances. This results is curious because refactorings rarely worsen inheritance, and only a few refactoring types worsen coupling and size – e.g., Pull Up Attribute and Pull Up Method.

It is worth noting that the recommendations above partially rely on data regarding critical attributes at the class level. Thus, generalizing our study results to critical attributes at other levels of the system may be unfeasible without further investigation. We strongly recommend researchers to replicate our work in other industrial contexts, especially those where evolving features is a major development task.

4.3.5 Participant Feedback

Table 4.6 summarizes the participant feedback collected from our Feedback Form. This form had three questions aimed at capturing the degree of confidence and comfort of developers in participating of the focus group sessions. All participants agree or strongly agree that they were able to discuss the relevant of critical attributes (Q1), rank these attributes (Q2) and, more importantly, share knowledge among their respective development teams (Q3).

The last form question (Q4), which does not appear in the table, asked participants about any other degradation symptoms they may find relevant

Table 4.6: Participant Feedback Collected via Feedback Form

Case	**Case A**			**Case**		
Participant	**A1**	**A2**	**A3**	**B1**	**B2**	**B3**
Q1: Confident to discuss relevance by attribute	Agree	Agree	Agree	Agree	Agree	Strongly agree
Q2: Confident to rank attributes by relevance	Strongly agree	Agree	Agree	Agree	Agree	Strongly agree
Q3: Comfortable to share opinion during discussion	Strongly agree	Agree	Strongly agree	Agree	Strongly agree	Agree

while evolving features. Only one participant of Case A replied, which is expected once Case A participants are the less concerned on improving the quality of source code. Participant A1 said that “bad names of attributes and methods” may hinder adding or enhancing features.

Differently, all participants of Case B had something to say about degradation symptoms that go beyond critical attributes. Participant B1 said that “lack of detailing of features” can make feature additions and enhancements harder to perform than usual. Participant B2 reported that both “a limited understanding of the business rules” and “a high complexity of using libraries and development environments” can also hamper evolving features. Finally, Participant P3 said that “an inconsistent mapping of business rules into source code implementation” can lead to difficulties in evolving features.

Curiously, issues that emerged from Q4 are closely associated with aspects of software development other than code structure and its design. This short feedback just reinforces our assumption that supporting software evolution depends on much more than simply enhancing code structures.

4.4 Threats to Validity

Construct Validity: We carefully defined the case study protocol and artifacts based on strict guidelines of research in empirical software engineering (Cruzes and Dyba, 2011; Runeson and Höst, 2009). We used previous studies (Chávez et al., 2017; Fernandes et al., 2020) as a reference for defining our study steps and procedures. Thus, we expected to support the proper data collection and analysis (Section 4.2.3. We performed both protocol and artifacts definition prior to the case study execution. As a result, we expected to prevent the study mischaracterization along with its execution.

Although we did not submit our study protocol to evaluation by the Ethics in Research Committee at PUC-Rio, we did our best to respect ethical issues in our work. First and foremost, we relied on the design of our most recent qualitative studies (Ferreira et al., 2020; Oliveira et al., 2020b;

Ralph et al., 2020) to assure all participants we comfortable with reporting on their perceptions. Second, two researchers contributed with insights on how to organize and conduct the focus group sessions. We discussed refinements in meetings before the study execution aimed at supporting a successful data collection. Third, Section 4.1.3 discusses we kept anonymous the identity of the focus group participants once we capture their perception of industry practices they apply in their jobs daily.

This study targeted six critical attributes: low class cohesion, high class complexity, high class coupling, large class hierarchy depth, large class hierarchy breadth, and large class size. These attributes have been typically used for monitoring the internal software quality (Bavota et al., 2013; Chávez et al., 2017; Fernandes et al., 2020). Thus, we expected to understand the developer's perception of critical attributes in the large, i.e., in an extensive scope. An immediate advantage of the large scope is deriving insights that go beyond the common wisdom on popular critical attributes, such as low class cohesion and large class size.

All six critical attributes investigated are derived from the five internal attributes of our quantitative study: cohesion, complexity, coupling, inheritance, and size (Chapter 3). We also analyzed the same set of 11 refactoring types introduced in Table 2.3. By deciding to focus on the same set of attributes and refactorings, we expected to support the comparison of both quantitative and qualitative studies – as we discuss in Section 4.3.4.

Internal Validity: Due to the COVID-19 pandemic, we could not perform face-to-face focus group sessions. We then created Zoom chats for hosting our discussions (Section 4.2.3). One could argue that the online environment may have discouraged developers to contribute and share knowledge. We did our best to promote a healthy environment so that everyone was comfortable – this can be confirmed by the data of Table 4.6. For instance, we asked each developer to share his perception of the attributes relevance. In addition, aimed at stimulated participants to engage in the focus group sessions, we informed our agreement in donating food to charity for each engaged participant.

We controlled the time spent by the development teams for discussing and reporting their perception of critical attributes. Each focus group session lasted no longer than two hours and a half. Thus, we aimed at preventing unnecessary spent of working in industry settings. The chosen time limit was sufficient for us to understand the relevance of each critical attribute and derive important insights on this matter.

During the focus groups sessions, we instructed participants on the formal definition of each critical attribute and metrics frequently used for capturing them. By doing this, we expected to normalize the participants' knowledge, thereby preventing certain participants from being unprepared for the discussions. However, one could argue that developers would have been biased in their responses, e.g. with respect to the samples set of metrics. To this regard we defend the need for introducing examples of metrics because some participants, especially in Case A, showed either little background on metrics or little concern on improving code quality (cf. Section 4.1).

We carefully kept video and audio records of each focus group session, with the permission of all participants, to support our posterior data analysis. This procedure was essential for either correcting the tabulated data or validating our discussions surrounding the reasons why each critical attribute is relevant for evolving features (details in Section 4.2.3). Thus, we expected to avoid missing and incorrect data while makes reliable our findings.

External Validity: We have conducted our focus group sessions in only one software development company and two development teams. The choice for this company was made by convenience (Section 4.1.3). In this particular case, we highlight that we opted for performing case studies, whose definition implies observing a phenomenon in its natural context (Runeson and Höst, 2009). Studies like this typically encompass only a small set of subjects, so that this is not a threat to our study. Nevertheless, we are aware that our study results, as their implications, may not apply to any companies and development teams.

We recruited only three participants for each focus group session. This may be a small number of participants. Besides, one could argue that such a limited set would have hindered the derivation of relevant study results on developer's perception of critical attributes. Although we partially agree with this argument, focus groups are not intended at large scale analyzes (Runeson and Höst, 2009). Rather, these studies aim at promoting discussions among a few participants in such a way controlling their participation becomes possible. Thus, it is reasonable that our study results are valid to a very limited scope.

Finally, the two systems analyzed in this work are at least partially implemented in the Java programming language. Although the development company is responsible for evolving dozens of systems, we opted for Java-based projects based on well-defined Section 4.1.3. We expect not just allow the study replication, but also make our qualitative data studies somehow comparable with those of our quantitative study (Chapter 3). Nevertheless, we observed some aspects of the developers' discussions during the focus group sessions that suggest other software domains could be further explored. For instance, the session transcriptions for Case B (Appendix D) suggest different perceptions of the relevance by critical attributes according to the programming paradigm – e.g. object-oriented versus functional programming. We encourage researchers to replicate our study to yet unexplored contexts like this.

Reliability: Aimed at bringing reliability to our qualitative study, we performed descriptive analysis similar to other studies on refactorings (Bibiano et al., 2019; Bibiano et al., 2020; Chávez et al., 2017). In addition, we performed thematic synthesis procedures (Cruzes and Dyba, 2011) with the purpose of systematically extracting themes of why critical attributes are relevant (or irrelevant) for evolving features. Thus, we expected at achieving a minimum degree of study replication as well as clarity in our data and discussions.

Regarding our qualitative study (Chapter 4), we discussed how critical attributes are perceived as interrelated by the participants of our two industry cases (see Section 4.3.4). We used this interrelation data for reasoning on how developers could perform refactorings for addressing critical attributes when evolving features (Table 4.4). However, one could argue that the developer's perception may not be the best way of establishing interrelations of critical attributes. Particularly, there may be other interrelations not covered by our study or neglected by the developers for some reason.

Still, we believe that our choice is quite reasonable for many reasons. For instance, developers argue that low class cohesion and high class coupling are interrelated, which is reflected in recurring strategies for detecting design smells such as Large Class (cf. Section 2.2). Similar reasoning applies to other interrelations, such as high class complexity and large class size (Fowler, 2018; Lanza and Marinescu, 2006). In summary, interrelations reported by the participants may not saturate all possibilities but, still, serve as a starting point for assisting refactorings to help in managing critical attributes.

Qualitative methods for empirical research in software engineering are powerful when employed by a team of researchers (Kontio et al., 2004; Runeson and Höst, 2009), each with different perceptions on the qualitative data under analysis. Thus, we acknowledge this threat to the validity of our study: only one research was responsible for analyzing most of the data, even when a second researcher provided support in double-validation or feedback. A implication of this is a possible decrease in reliability of the study results, especially because of the interpretative nature of the qualitative analysis.

Although we did not properly address this threat, it is worth mentioning that the researcher responsible for the qualitative analysis had previous experiences on this matter in different occasions (Fernandes et al., 2016a; Oliveira et al., 2020b; Uchôa et al., 2019). For instance, he has been involved with the application of thematic synthesis procedures in past work (Oliveira et al., 2020a; Oliveira et al., 2020b) and qualitative studies with practitioners, researchers, and students (Fernandes et al., 2016a; Ferreira et al., 2020; Uchôa et al., 2019).

4.5 Chapter Summary

This chapter discussed the outcomes of a qualitative case study (Runeson and Höst, 2009) based on focus groups (Kontio et al., 2004). We carefully designed and conducted two industrial case studies aimed at understanding the relevance of six critical attributes at the level of classes in the context of software evolution – all based on the five internal attributes analyzed in our previous quantitative study (Chapter 3). As a result, we derived many interesting insights that may help both practitioners and researchers in managing critical attributes while evolving features. For instance, low class cohesion and high class complexity are ultimately perceived as relevant when evolving features. Thus, popular refactorings (e.g. Extract Method and Move Method) should be carefully applied for preventing a major worsening of code structure and its design.

We also filled some critical literature gaps that prevented us from further understanding the refactoring effect on internal attributes. Particularly, we elicited factors that lead developers to consider certain critical attributes as relevant, e.g. the difficulty to perform testing and feature additions. Finally, we discuss how refactorings may help in managing critical attributes when evolving features through recommendations derived on crossing quantitative data (Chapter 3) and qualitative data (this chapter).

The next chapter concludes this doctoral thesis. In Chapter 5, we summarize the implications of the results obtained through our quantitative and qualitative studies. We also discuss key publications achieved during this PhD course – some of them directly derived from this doctoral thesis.

5
Conclusion

Software change is the basis of what we know as software evolution. Developers daily apply hundreds of changes to their systems (Kim et al., 2014; Murphy-Hill et al., 2012), aimed at meeting ever-changing stakeholders' demands and operating environment settings (Lehman, 1980; Mens et al., 2010). As a system evolves (Martin, 2002), developers must carefully apply changes (Elfatatry, 2007). Indeed, each change in isolation may affect the system in different granularities, from fined-grained code elements to the entire system architecture (Chávez et al., 2017; Paixao et al., 2019). Additionally, every change has one or more developer intents, which range from purely enhancing code structure and its design to fixing bugs and evolving software features (Gousios et al., 2015; Paixao et al., 2019; Silva et al., 2016a; Tao et al., 2012).

Although developers perform several changes in the regular basis, applying certain changes may be quite challenging in practice. Indeed, an undisciplined application of changes may eventually degrade, rather than improve, the code structure and its design (Fernandes et al., 2017b; Tufano et al., 2017). More critically, performing other changes with major underlying intents, especially evolving features, tends to becomes hard than usual or even unfeasible (Fernandes, 2019a). Evolving features means either adding new features or enhancing existing features (Burke, 2014). Therefore, both mitigating and fully addressing degradation symptoms is essential to enhance the internal software quality (Bavota et al., 2013; Fernandes et al., 2017b; Tufano et al., 2017).

The literature regarding degradation symptoms is extensive. Examples of degradation symptoms are anomalous metric values (Chidamber and Kemerer, 1994; Fernandes et al., 2017a; Pantiuchina et al., 2018), critical internal attributes (Chávez et al., 2017; Fernandes et al., 2020) and design smells (Fowler, 2018). Most of these degradation symptoms are quantifiable via traditional software metrics, e.g. Coupling between Objects (CBO) (Chidamber and Kemerer, 1994; Lanza and Marinescu, 2006). By definition, a degradation symptom is just a hint of degraded code structure and its design. A degradation symptom is relevant for evolving features when developers have the need of either mitigating or fully addressing it before performing feature additions and enhancements.

Previous studies (Bavota et al., 2015; Bibiano et al., 2019; Chaparro et al., 2014; Palomba et al., 2014; Pantiuchina et al., 2018), partially investigated the relationship between degradation symptoms and changes designed for enhancing code structures, i.e., refactorings (Fowler, 2018). Unfortunately, there is still scarce knowledge on what degradation symptoms are relevant for practitioners in the industry. In addition, most studies targeted either critical metric values (Bavota et al., 2015; Chaparro et al., 2014; Pantiuchina et al., 2018) or design smells (Bibiano et al., 2019; Palomba et al., 2014) rather than critical attributes. This is a major opportunity for addressing literature gaps and further assist developers in enhancing code structures, via refactorings, while evolving features.

This doctoral thesis discussed two complementary studies aimed at empirically addressing two issues, thereby filling literature gaps, such as limited analysis scopes (variety of systems, internal attributes and so forth). The first issue is the scarce and imprecise knowledge on how refactorings affect internal attributes used for capturing critical attributes such as low class cohesion and high class complexity. The second issue is the lack of empirical evidence on how much (and why) developers perceive critical attribute as relevant (or irrelevant) for evolving features.

Via a large quantitative study (Chapter 3), we explored the relationship between refactorings and internal attributes. We extended previous work (Chávez et al., 2017) on 23 open source systems with new analyses, statistical testing, and discussions. We analyzed 11 refactoring types, 25 metrics, and five internal attributes: cohesion, complexity, coupling, inheritance, and size. Besides revealing new insights on the refactoring effect, we contradict literature assumptions (Bavota et al., 2015; Fowler, 2018).

Through a case study (Runeson and Höst, 2009) based on focus group sessions (Kontio et al., 2004), we complemented the findings of the previous study and shed light on the relevance of critical attributes in industry, when evolving software features (Chapter 4). We investigated six (all the class level) critical attributes derived from the five internal attributes mentioned above. We asked participants of each session on the relevance of these critical attributes when evolving features: low class cohesion, high class complexity, high class coupling, large class hierarchy depth, large class hierarchy breadth, and large class size. Crossing the findings from both conducted studies, we discuss how critical attributes can be addressed through refactoring, when evolving features.

We structured this final chapter of the doctoral thesis as follows. Section 5.1 summarizes major implications of our quantitative study (Chapter 3) about the refactoring effect on internal attributes. We display insights for both researchers and practitioners, with an emphasis on mechanisms for assisting developers in improving internal attributes – especially during software evolution. Section 5.2 presents major implications of our qualitative study (Chapter 4) on the developer's perception of critical attributes for evolving features. We highlight the need for re-designing the current automated refactoring tools to help manage critical attributes based on this empirically validated perception. Section 5.3 lists some closely related publications achieved during the PhD course. Finally, Section 5.4 discusses peripheral yet relevant publications also achieved during the PhD course.

5.1 Quantitative Study Implications

Implication for Practitioners 1: *Developers should carefully plan the refactoring application, especially when combined with intents such as evolving features, in order to prevent software degradation.*

Some previous studies (Chávez et al., 2017; Murphy-Hill et al., 2012; Paixao et al., 2019; Silva et al., 2016a) showed that developers not very often intend to purely enhance the code structure and its design while refactoring systems. On the other hand, our study corroborates that developers often apply refactorings on those code elements that are more likely to represent relevant degradation symptom (Chapter 3.3). Indeed, the majority of refactorings and re-refactorings (above 90% of the total) occur in code elements with at least one critical attribute. This result suggest that developers are possibly aware of degraded code structure and design while performing refactorings in the regular basis.

We conclude it is equally crucial to raise awareness on the need for applying changes in a careful rather than an undisciplined fashion. Such awareness is particularly necessary in the case of floss refactorings, i.e. refactorings applied in conjunction with other changes. Floss refactorings represent about 73% of all refactorings, and their typically negative effect on internal attributes may be caused by those other changes, e.g. changes intended to evolving features. A careful refactoring planning and application could prevent developers from unexpectedly (and undesirably) degrading the code structure and its design.

Implication for Practitioners 2: *Applying certain refactoring types requires parsimony – and a pretty clear code structure enhancement intent.*

Our quantitative study results reveal that each refactoring type has its advantages and drawbacks with respect to code structure enhancement. Let us take Extract Method, which is a very popular refactoring type in industry (Murphy-Hill et al., 2012; Silva et al., 2016a), as an example. While Extract Method can improve many attributes together, especially when applied as a re-refactoring (Section 3.5), it can worsen other attributes as well (Section 3.4).

Further research is still required to understand the trade-off between improving and worsening different attributes for each refactoring type. However, our study suffices to raise awareness on the fact that *applying (re-)refactorings requires parsimony, but also a clear code enhancement intent.* That is, developers have to state clearly the internal attributes that really matter in a certain system, so that they can mitigate or fully address degradation symptoms without unnecessarily increasing software evolution costs.

Implication for Researchers 1: *Techniques aimed at either prioritizing and ranking of refactoring opportunities should strongly consider the refactoring effect on internal attributes.*

Our study results suggest that, consciously or not, *developers very often apply (re-)refactorings on code elements affected by one or more critical attributes.* Besides contradicting the literature (Bavota et al., 2015), this result reinforces that critical attributes are useful hints of degraded code structures that require refactoring. Consequently, the existing techniques for prioritizing and ranking refactoring opportunities, e.g. (Lin et al., 2016; Szőke et al., 2015a; Tsantalis et al., 2013), should strongly consider the (re-)refactoring effect on internal attributes.

Implication for Researchers 2: *Search-Based Software Engineering (SBSE) can support the refactoring composition towards a more significant enhancement of the code structure and its design.*

Integrated development environments (IDEs) provide limited support for supporting developers in tailoring their refactorings to achieve their intents (Kim et al., 2014), such as adjusting the outcome of each change. In particular, they provide a restrict support for customizing and composing refactorings in a way that is suitable to the working context of a developer. Our most recent studies (Bibiano et al., 2019; Bibiano et al., 2020; Fernandes et al., 2019b; Fernandes, 2019a) aims at understanding the process of composing two or more refactorings to mitigate or fully address degraded code structures, including those revealed by design smells (Fowler, 2018). Our major goal is defining strategies to *compose multiple refactorings in such a way they can overcome the limited benefits of applying isolated refactorings* (Yoshida et al., 2016).

There may be many alternatives for composing refactoring types (Fernandes et al., 2019b), each with a different effect on the critical attributes. Similar to a previous work (Lin et al., 2016), SBSE techniques could help in finding compositions that optimize the improvement of certain attributes while avoid the worsening of other attributes. In future work, we aim to explore the potential of supporting the customization and composition of refactorings (Fernandes et al., 2019b). The resulting customization and composition should be able to improve multiple internal attributes in conjunction with the achievement of other intents, e.g. evolving features.

5.2 Industry Case Study Implications

Implication for Practitioners 1: *Existing techniques could help in mitigating or fully addressing critical attributes while evolving features.*

Through our focus group sessions, we were able to reveal what critical attributes were perceived as relevant by developers while evolving features in industry. We aimed at confirming and complementing preliminary insights of previous studies (Palomba et al., 2014; Taibi et al., 2017; Yamashita and Moonen, 2013) on the importance of addressing critical attributes for facilitating software evolution. Our major results include the validation of low class cohesion and high class complexity as ultimately relevant from the developer's perspective. On the one hand, existing techniques for detecting designs smells – which are usually combinations of two or more critical attributes – could be handful for assisting developers in analyzing critical attributes in their systems. There is a myriad of options in the literature for this particular purpose (Fernandes et al., 2016b; Lin et al., 2016; Liu et al., 2011; Oliveira et al., 2020b; Szőke et al., 2015a).

In addition, the technical literature of anomalous metric values and critical attributes is quite diverse and comprehensive (Bavota et al., 2013; Bavota et al., 2015; Chaparro et al., 2014; Chávez et al., 2017; Fernandes et al., 2020; Lanza and Marinescu, 2006; Lorenz and Kidd, 1994). Still, our Background Form in particular (Section 4.2.2) reminded us that developers may not be sufficiently aware of the techniques already proposed for supporting the analysis of critical attributes in practical settings. Raising such awareness is fundamental for developers to discuss and manage critical attributes, especially in cases of short time for delivery products – a recurring issue (cf. Table 4.2 and Table 4.3).

Implication for Practitioners 2: *Our refactoring recommendations could be a starting point for development teams to discuss managing critical attributes while evolving features.*

As previously discussed, we systematically crossed our quantitative data of Chapter 3 – about the refactoring effect on five internal attributes – with our qualitative data of Chapter 4 – regarding the developer's perception of critical attributes as relevant for evolving features. After that, we discuss how developers may perform refactorings to help manage critical attributes along with software evolution. Although these recommendations are not extensive, they serve as a starting point for discussing the role of refactorings in enhancing code structure and its design.

For instance, we discussed that low class cohesion and high class complexity are ultimately relevant for developers while evolving features. In this case, refactoring types such as Extract Method, Move Method, and Pull Up Method could effectively help in managing critical attributes like low class cohesion (Table 3.4). Another example is that Extract Method could help addressing high class complexity without major side effects on other internal attributes that may become critical after performing the refactorings.

The value of our recommendations is in demonstrating to developers that enhancing code structures is not trivial and requires a careful planning and execution. Otherwise, an unexpected decay of internal software quality is expected. We believe our refactoring recommendations serve as a starting point for decision-makings on critical attributes during software evolution.

Implication for Researchers 1: *Existing techniques should be adapted to prioritize the management of critical attributes that really concern developers while evolving features.*

Our qualitative study revealed how much (and why) critical attributes may be relevant for developers during software evolution. It also supported refactoring recommendations that developers could incorporate to enhance code structure and its design. Nevertheless, such a simple set of recommendations is probably insufficient for assisting developers in refactoring in practice. Thus, automated refactoring assistance is beyond desired: it is necessary.

Additionally, each system may have several critical attributes to manage. In cases like this, automation becomes fundamental and a major feature should be prioritizing critical attributes that concern developers the most. We encourage researchers to re-design their tools for assisting software quality assessment by considering the developer's perception of critical attributes that concern them the most while evolving features.

Implication for Researchers 2: *Recommender systems should incorporate mechanisms for driving changes aimed at other intents than the pure enhancement of code structure and its design.*

A few recommender systems, e.g. (Lin et al., 2016; Szőke et al., 2015a) aim to assist developers in enhancing code structures (Szőke et al., 2015a) and evolving software architectures (Lin et al., 2016). In both cases, the traditional refactorings of Fowler's Refactoring book (Fowler, 2018) are employed towards a facilitated software development in general. These tools rely on the assumption that code structure and design free of degradation symptoms is favorable to adding or enhancing features. This assumption is not incorrect *per se*. However, the ideal code structure and design suggested by these tools may be either unnecessary or too costly in practice.

Our qualitative study results (Sections 4.3.2 and 4.3.3) suggest that, during software evolution, developers tend to concentrate effort in applying only those changes necessary to achieve their major intent. While existing recommender systems typically suggests at once dozens of changes, their practical adoption sounds unrealistic at times. This observation is especially valid when the developer wants to add or enhance features very locally in the code. Our study results could help researchers in re-designing tools for addressing degradation symptoms – including critical attributes with fewer changes and focused on problems that actually affect the tasks of evolving features. For instance, tool designers could try incorporating our refactoring recommendations to assist disciplined refactorings aimed at mitigating or fully addressing critical attributes.

5.3 Closely Related Publications

The **first publication** (Chávez et al., 2017) closely related to this doctoral thesis was a best paper at the 31st Brazilian Symposium on Software Engineering (SBES) in 2017. This paper empirically assessed the relationship between refactoring and critical attributes. Differently of previous studies quite limited in scope, we analyzed five internal attributes: cohesion, complexity, coupling, inheritance, size. Our study encompassed 11 popularly adopted refactorings (Murphy-Hill et al., 2012; Silva et al., 2016a), e.g. Extract Method and Move Method, and 25 traditional metrics. We have significantly contributed with the study design, writing, and the effect analysis. We also documented related work, *from which we tracked limitations of the automated refactoring support*.

Aimed at filling additional literature gaps and extending our past work (Chávez et al., 2017), we have led a **second publication** (Fernandes et al., 2020) at the Information and Software Technology (IST) journal. We have performed and documented an unprecedented study on a recently proposed concept called re-refactoring. Re-refactoring means refactoring a previously refactored code element, e.g. a class or a method. We addressed some major threats to the validity of our previous work by i) applying different statistical methods and ii) performing an extensive literature comparison. This study has helped us to realize that i) *managing critical attributes is insufficient to mitigate and fully address degradation symptoms*, and ii) *change recommendations should consider other intents such as partially improving attributes to evolve features.*

The **third publication** (Bibiano et al., 2019) closely related to this doctoral thesis was a large study published in the 13th International Symposium on Empirical Software Engineering and Measurement (ESEM) in 2019. We assessed the refactoring effect on both introducing and removing design smells. We have contributed with phases such as the study design, writing, and impact analysis. This was the first empirical study to assess composite refactorings (formerly batch refactorings). A composite refactoring is constituted of two or more refactorings combined and applied together with a shared intent (Bibiano et al., 2019), which rages as much as for each single refactoring (Fernandes et al., 2019b; Fernandes, 2019a). This study has helped us to confirm that i) *evolving features strongly depends on systematically applying composite refactoring*, and ii) *evolving features should be tool-aided.*

The **fourth publication** (Bibiano et al., 2020) is derived from the third one. It consists of a quantitative study accepted in the 28th International

Conference on Program Comprehension (ICPC) in early 2020. This study targeted a phenomenon we called incomplete composite refactoring. Shortly, incomplete composite is a composite that fell short in fully removing a design smell it was supposed to remove. Once again, we performed some major contributions, from the study design definition to the paper writing and review. This was the first study aimed at investigating internal attributes in the context of composite refactoring. We have contributed with phases such as the study design, writing, and text review. Contrary to expectations, our study revealed that, at the class level, incomplete composites have a slight or no effect on internal attributes after all. Thus, *coping with critical attributes via composite refactoring is much harder than one could expect.*

The first attempt to formalize this doctoral thesis, our **fifth publication** (Fernandes, 2019a), was presented in the Doctoral Symposium of the 41st International Conference on Software (ICSE) in 2019. At that time, we proposed investigating what we called obstacles to evolving features. Similarly to our definition of critical attributes that are relevant for evolving features (Section 1.1), these are obstacles that make it hard or impossible to evolve features. The paper was praised in both reviews and the face-to-face presentation, but our study required major refinements. Instead of focusing most of our effort in drawing refactoring recommendations to enable feature additions, *we should clarify what critical attributes are ultimately relevant for evolving features.* Thus, we shifted our study focus from designing a recommender system to understanding the relevance of critical attributes from the developer's perception.

Our **sixth publication** was an exercise aimed at discussing the possible alternatives for combining two or more refactorings within a composite refactoring. We searched for scenarios of refactoring composition in open code review repositories, specifically those powered by the Gerrit Code Review platform[1]. We analyzed some examples of discussions made by developers along with the code reviews. One of our goals was tracking scenarios in which the collaboration among developers could lead to the recommendation of effective composite refactorings. This publication was presented at the 3rd International Workshop on Refactoring (IWoR), co-located with the 41st ICSE (Fernandes et al., 2019b). Our exercise was fruitful in *revealing more limitations of the state of the art refactoring tools, especially when it comes to evolving features.*

[1]https://www.gerritcodereview.com/

5.4 Other Publications

We have contributed with other publications along with this PhD course. Some of them have significantly contributed with useful insights to this doctoral thesis, although they targeted quite different challenges in software engineering. We briefly introduce these studies below.

The **first publication** (Ferreira et al., 2018) was a poster paper presented at the 40th International Conference of Software Engineering (ICSE) in 2018. This study assessed the relationship between refactorings and bug introductions. As in the first closely related publication (Chávez et al., 2017), we substantially contributed with the study design, writing, and analysis. We summarized related work, which revealed additional limitations of the current refactoring tools. We also proposed means to compute the distance from refactorings to bug-introducing commits. However, our "turning point" was realizing that i) *change recommendations should consider potential bug introductions* and ii) *analyzes commit by commit often overshadow the changes applied within each commit.* Thus, researchers should carefully address threats to validity associated with a too coarse-grained analysis of the refactoring effect on the software quality.

The **second** (Fernandes et al., 2019c) and **third publications** (Uchôa et al., 2019), both published in 2019, are interrelated. The former we published in the 16th International Conference on Information Technology: New Generations (ITNG). The later appeared in the 1st International Workshop on Software Engineering for Healthcare (SEH), co-located with the 41st ICSE. These two publications derived from our practical experience with adding gamification features into a system. This experience, documented in details in our seventh related publication (Section 5.3) helped us to understand *how hard can be evolving features in a legacy source code and software architecture.*

Finally, the **fourth** (Oliveira et al., 2020a) and **fifth publications** (Oliveira et al., 2020b) were published in Empirical Software Engineering (EMSE) journal and the Information and Software Technology (IST) journal, respectively. In both studies, we have largely contributed with the study design, execution, and writing. These studies included interviews with practitioners in order to understand a particular development task: respectively, identifying highly productive developers and detecting design smells. In particular, *these studies were fruitful exercises to perform and discuss the outcomes of thematic synthesis procedures* (Cruzes and Dyba, 2011), which we also applied to our qualitative study (Chapter 4).

Bibliography

[Al Dallal and Abdin, 2017] 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.

[Apel and Kästner, 2009] APEL, S.; KÄSTNER, C.. **An overview of feature-oriented software development**. Journal of Object Technology, 8(5):49–84, 2009.

[Avelino et al., 2019] AVELINO, G.; PASSOS, L.; HORA, A. ; VALENTE, M. T.. **Measuring and analyzing code authorship in 1+ 118 open source projects**. Science of Computer Programming, 176:14–32, 2019.

[Basili and Rombach, 1988] BASILI, V.; ROMBACH, D.. **The TAME project: Towards improvement-oriented software environments**. IEEE Transactions on Software Engineering, 14(6):758–773, 1988.

[Bavota et al., 2013] BAVOTA, G.; DIT, B.; OLIVETO, R.; DI PENTA, M.; POSHYVANYK, D. ; DE LUCIA, A.. **An empirical study on the developers' perception of software coupling**. In: PROCEEDINGS OF THE 35TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE), p. 692–701, 2013.

[Bavota et al., 2015] 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.

[Bibiano et al., 2019] 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: PROCEEDINGS OF THE 13TH INTERNATIONAL SYMPOSIUM ON EMPIRICAL SOFTWARE ENGINEERING AND MEASUREMENT (ESEM), p. 1–11, 2019.

[Bibiano et al., 2020] BIBIANO, A. C.; SOARES, V.; COUTINHO, D.; FERNANDES, E.; CORREIA, J. L.; TARCÍSIO, K.; OLIVEIRA, A.; GARCIA, A.;

GHEYI, R.; RIBEIRO, M.; FONSECA, B.; BARBOSA, C. ; OLIVEIRA, D.. **How does incomplete composite refactoring affect internal quality attributes?** In: PROCEEDINGS OF THE 28TH INTERNATIONAL CONFERENCE ON PROGRAM COMPREHENSION (ICPC), p. 1–11, 2020.

[Bieman and Kang, 1995] BIEMAN, J.; KANG, B.-K.. **Cohesion and reuse in an object-oriented system**. ACM SIGSOFT Software Engineering Notes, 20(SI):259–262, 1995.

[Briand et al., 2000] BRIAND, L.; WÜST, J.; DALY, J. ; PORTER, D. V.. **Exploring the relationships between design measures and software quality in object-oriented systems**. Journal of Systems and Software, 51(3):245–273, 2000.

[Burke, 2014] BURKE, J.. **Utilizing feature location techniques for feature addition and feature enhancement**. In: PROCEEDINGS OF THE 29TH INTERNATIONAL CONFERENCE ON AUTOMATED SOFTWARE ENGINEERING (ASE), p. 879–882, 2014.

[Chaparro et al., 2014] CHAPARRO, O.; BAVOTA, G.; MARCUS, A. ; DI PENTA, M.. **On the impact of refactoring operations on code quality metrics**. In: PROCEEDINGS OF THE 30TH INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE AND EVOLUTION (ICSME), p. 456–460, 2014.

[Chávez, 2017] LÓPEZ, A.. **How does refactoring affect internal quality attributes? A multi-project study**. Master's dissertation: Informatics Department (DI), Pontifical Catholic University of Rio de Janeiro (PUC-Rio), p. 1–80, 2017.

[Chávez et al., 2017] CHÁVEZ, A.; FERREIRA, I.; FERNANDES, E.; CEDRIM, D. ; GARCIA, A.. **How does refactoring affect internal quality attributes? A multi-project study**. In: PROCEEDINGS OF THE 31ST BRAZILIAN SYMPOSIUM ON SOFTWARE ENGINEERING (SBES), p. 74–83, 2017.

[Chidamber and Kemerer, 1994] CHIDAMBER, S.; KEMERER, C.. **A metrics suite for object oriented design**. IEEE Transactions on Software Engineering, 20(6):476–493, 1994.

[Chung and do Prado Leite, 2009] CHUNG, L.; DO PRADO LEITE, J. C.. **On non-functional requirements in software engineering**. In: CONCEPTUAL MODELING: FOUNDATIONS AND APPLICATIONS, p. 363–379. Springer, 2009.

[Cruzes and Dyba, 2011] CRUZES, D.; DYBA, T.. **Recommended steps for thematic synthesis in software engineering**. In: PROCEEDINGS OF THE 5TH INTERNATIONAL SYMPOSIUM ON EMPIRICAL SOFTWARE ENGINEERING AND MEASUREMENT (ESEM), p. 275–284, 2011.

[Destefanis et al., 2014] DESTEFANIS, G.; COUNSELL, S.; CONCAS, G. ; TONELLI, R.. **Software metrics in agile software: An empirical study**. In: PROCEEDINGS OF THE 15TH INTERNATIONAL CONFERENCE ON AGILE SOFTWARE DEVELOPMENT (XP), p. 157–170, 2014.

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

[Elfatatry, 2007] ELFATATRY, A.. **Dealing with change: Components versus services**. Communications of the ACM, 50(8):35–39, 2007.

[Fernandes, 2019a] FERNANDES, E.. **Stuck in the middle: Removing obstacles to new program features through batch refactoring**. In: PROCEEDINGS OF THE 41ST INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE), COMPANION PROCEEDINGS, p. 206–209, 2019.

[Fernandes et al., 2016a] FERNANDES, E.; FERREIRA, F.; NETTO, J. A. ; FIGUEIREDO, E.. **Information systems development with pair programming: An academic quasi-experiment**. In: PROCEEDINGS OF THE 12TH BRAZILIAN SYMPOSIUM ON INFORMATION SYSTEMS (SBSI), p. 486–493, 2016.

[Fernandes et al., 2016b] FERNANDES, E.; OLIVEIRA, J.; VALE, G.; PAIVA, T. ; FIGUEIREDO, E.. **A review-based comparative study of bad smell detection tools**. In: PROCEEDINGS OF THE 20TH INTERNATIONAL CONFERENCE ON EVALUATION AND ASSESSMENT IN SOFTWARE ENGINEERING (EASE), p. 18:1–18:12, 2016.

[Fernandes et al., 2017a] FERNANDES, E.; FERREIRA, L. P.; FIGUEIREDO, E. ; VALENTE, M. T.. **How clear is your code? An empirical study with programming challenges**. In: PROCEEDINGS OF THE 20TH IBERO-AMERICAN CONFERENCE ON SOFTWARE ENGINEERING (CIBSE), EX-

PERIMENTAL SOFTWARE ENGINEERING (ESELAW) TRACK, p. 1–14, 2017.

[Fernandes et al., 2017b] FERNANDES, E.; VALE, G.; SOUSA, L.; FIGUEIREDO, E.; GARCIA, A. ; LEE, J.. **No code anomaly is an island: Anomaly agglomeration as sign of product line instabilities.** In: PROCEEDINGS OF THE 16TH INTERNATIONAL CONFERENCE ON SOFTWARE REUSE (ICSR), p. 48–64, 2017.

[Fernandes et al., 2019b] FERNANDES, E.; UCHÔA, A.; BIBIANO, A. C. ; GARCIA, A.. **On the alternatives for composing batch refactoring.** In: PROCEEDINGS OF THE 3RD INTERNATIONAL WORKSHOP ON REFACTORING (IWOR), CO-LOCATED WITH THE 41ST INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE), p. 9–12, 2019.

[Fernandes et al., 2019c] FERNANDES, E.; UCHÔA, A.; SOUSA, L.; OLIVEIRA, A.; DE MELLO, R.; BARROCA, L. P.; CARVALHO, D.; GARCIA, A.; FONSECA, B. ; TEIXEIRA, L.. **VazaZika: A software platform for surveillance and control of mosquito-borne diseases.** In: PROCEEDINGS OF THE 16TH INTERNATIONAL CONFERENCE ON INFORMATION TECHNOLOGY: NEW GENERATIONS (ITNG), p. 617–620, 2019.

[Fernandes et al., 2020] 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.

[Ferreira, 2018] FERREIRA, I.. **Assessing the bug-proneness of refactored code: Longitudinal multi-project studies.** Master's dissertation: Informatics Department (DI), Pontifical Catholic University of Rio de Janeiro (PUC-Rio), p. 1–90, 2018.

[Ferreira et al., 2018] FERREIRA, I.; FERNANDES, E.; CEDRIM, D.; UCHÔA, A.; BIBIANO, A. C.; GARCIA, A.; CORREIA, J. L.; SANTOS, F.; NUNES, G.; BARBOSA, C.; FONSECA, B. ; DE MELLO, R.. **The buggy side of code refactoring: Understanding the relationship between refactorings and bugs.** In: PROCEEDINGS OF THE 40TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE), COMPANION PROCEEDINGS, p. 406–407, 2018.

[Ferreira et al., 2020] FERREIRA, F.; FERNANDES, E.; OLIVEIRA, J.; SOUZA, M. ; FIGUEIREDO, E.. **How difficult and effective is writing asser-**

tions for observing bugs at runtime? In: PROCEEDINGS OF THE 23RD IBERO-AMERICAN CONFERENCE ON SOFTWARE ENGINEERING (CIBSE), EXPERIMENTAL SOFTWARE ENGINEERING (ESELAW) TRACK, p. 1–14, 2020.

[Fontana and Spinelli, 2011] FONTANA, F. A.; SPINELLI, S.. **Impact of refactoring on quality code evaluation**. In: PROCEEDINGS OF THE 4TH WORKSHOP ON REFACTORING TOOLS (WRT), CO-LOCATED WITH THE 33RD INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE), p. 37–40, 2011.

[Fowler, 2018] FOWLER, M.. **Refactoring: Improving the Design of Existing Code**. Addison-Wesley Professional, 2nd edition, 2018.

[Gamma et al., 1993] GAMMA, E.; HELM, R.; JOHNSON, R. ; VLISSIDES, J.. **Design patterns: Abstraction and reuse of object-oriented design**. In: PROCEEDINGS OF THE 7TH EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING (ECOOP), p. 406–431, 1993.

[Gousios et al., 2015] GOUSIOS, G.; ZAIDMAN, A.; STOREY, M.-A. ; VAN DEURSEN, A.. **Work practices and challenges in pull-based development: The integrator's perspective**. In: PROCEEDINGS OF THE 37TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE), p. 358–368, 2015.

[Henry and Kafura, 1981] HENRY, S.; KAFURA, D.. **Software structure metrics based on information flow**. IEEE Transactions on Software Engineering, SE-7(5):510–518, 1981.

[Hinkle et al., 2002] HINKLE, D.; WIERSMA, W. ; JURS, S.. **Applied Statistics for the Behavioral Sciences**. Houghton Mifflin, 5th edition, 2002.

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

[Kalinowski et al., 2020] KALINOWSKI, M.; LOPES, H.; TEIXEIRA, A. F.; DA SILVA CARDOSO, G.; KURAMOTO, A.; ITAGYBA, B.; BATISTA, S. T.; PEREIRA, J. A.; SILVA, T.; WARRAK, J. A.; DA COSTA, M.; FISCHER, M.; SALGADO, C.; TEIXEIRA, B.; CHUEKE, J.; FERREIRA, B.; LIMA, R.; VILLAMIZAR, H.; BRANDÃO, A.; BARBOSA, S.; POGGI, M.; PELIZARO, C.; LEMES, D.; WALTEMBERG, M.; LOPES, O. ; GOULART, W.. **Lean R&D:**

An agile research and development approach for digital transformation. In: PROCEEDINGS OF THE 21ST INTERNATIONAL CONFERENCE ON PRODUCT-FOCUSED SOFTWARE PROCESS IMPROVEMENT (PROFES), p. 106–124, 2020.

[Källén et al., 2014] KÄLLÉN, M.; HOLMGREN, S. ; ÞÓRA HVANNBERG, E.. **Impact of code refactoring using object-oriented methodology on a scientific computing application.** In: PROCEEDINGS OF THE 14TH INTERNATIONAL WORKING CONFERENCE ON SOURCE CODE ANALYSIS & MANIPULATION (SCAM), p. 125–134, 2014.

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

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

[Kontio et al., 2004] KONTIO, J.; LEHTOLA, L. ; BRAGGE, J.. **Using the focus group method in software engineering: Obtaining practitioner and user experiences.** In: PROCEEDINGS OF THE 3RD INTERNATIONAL SYMPOSIUM ON EMPIRICAL SOFTWARE ENGINEERING (ISESE), p. 271–280, 2004.

[Lanubile and Visaggio, 1995] LANUBILE, F.; VISAGGIO, G.. **Iterative reengineering to compensate for quick-fix maintenance.** In: PROCEEDINGS OF THE 10TH INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE (ICSM), p. 1–7, 1995.

[Lanza and Marinescu, 2006] LANZA, M.; MARINESCU, R.. **Object-oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems.** Springer Science & Business Media, 1st edition, 2006.

[Le et al., 2016] LE, D.; CARRILLO, C.; CAPILLA, R. ; MEDVIDOVIC, N.. **Relating architectural decay and sustainability of software systems.** In: PROCEEDINGS OF THE 13TH WORKING CONFERENCE ON SOFTWARE ARCHITECTURE (WICSA), p. 178–181, 2016.

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

[Li and Henry, 1993] LI, W.; HENRY, S.. **Object-oriented metrics that predict maintainability**. Journal of Systems and Software, 23(2):111–122, 1993.

[Lin et al., 2016] LIN, Y.; PENG, X.; CAI, Y.; DIG, D.; ZHENG, D. ; ZHAO, W.. **Interactive and guided architectural refactoring with search-based recommendation**. In: PROCEEDINGS OF THE 24TH INTERNATIONAL SYMPOSIUM ON FOUNDATIONS OF SOFTWARE ENGINEERING (FSE), p. 535–546, 2016.

[Liu et al., 2011] LIU, H.; MA, Z.; SHAO, W. ; NIU, Z.. **Schedule of bad smell detection and resolution: A new way to save effort**. IEEE Transactions on Software Engineering, 38(1):220–235, 2011.

[Lorenz and Kidd, 1994] LORENZ, M.; KIDD, J.. **Object-oriented Software Metrics: A Practical Guide**. Prentice Hall, 1st edition, 1994.

[Mantyla et al., 2004] MANTYLA, M.; VANHANEN, J. ; LASSENIUS, C.. **Bad smells: Humans as code critics**. In: PROCEEDINGS OF THE 20TH INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE (ICSM), p. 399–408, 2004.

[Martin, 2002] MARTIN, R.. **Agile Software Development: Principles, Patterns, and Practices**. Prentice Hall, 1st edition, 2002.

[McCabe, 1976] MCCABE, T.. **A complexity measure**. IEEE Transactions on Software Engineering, SE-2(4):308–320, 1976.

[Meirelles et al., 2010] MEIRELLES, P.; SANTOS JR, C.; MIRANDA, J.; KON, F.; TERCEIRO, A. ; CHAVEZ, C.. **A study of the relationships between source code metrics and attractiveness in free software projects**. In: PROCEEDINGS OF THE 24TH BRAZILIAN SYMPOSIUM ON SOFTWARE ENGINEERING (SBES), p. 11–20, 2010.

[Mens et al., 2010] MENS, T.; GUEHÉNÉUC, Y.-G.; FERNÁNDEZ-RAMIL, J. ; D'HONDT, M.. **Guest editors' introduction: Software evolution**. IEEE Software, 4(27):22–25, 2010.

[Murgia et al., 2011] MURGIA, A.; TONELLI, R.; COUNSELL, S.; CONCAS, G. ; MARCHESI, M.. **An empirical study of refactoring in the context of FanIn and FanOut coupling**. In: PROCEEDINGS OF THE 18TH WORKING CONFERENCE ON REVERSE ENGINEERING (WCRE), p. 372–376, 2011.

[Murphy-Hill et al., 2012] MURPHY-HILL, E.; PARNIN, C. ; BLACK, A.. **How we refactor, and how we know it**. IEEE Transactions on Software Engineering, 38(1):5–18, 2012.

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

[Oliveira et al., 2020a] OLIVEIRA, E.; FERNANDES, E.; STEINMACHER, I.; CRISTO, M.; CONTE, T. ; GARCIA, A.. **Code and commit metrics of developer productivity: A study on team leaders perceptions**. Empirical Software Engineering, 25(4):2519–2549, 2020.

[Oliveira et al., 2020b] OLIVEIRA, R.; DE MELLO, R.; FERNANDES, E.; GARCIA, A. ; LUCENA, C.. **Collaborative or individual identification of code smells? On the effectiveness of novice and professional developers**. Information and Software Technology, 120:106242, 2020.

[Paixao et al., 2019] PAIXAO, M.; KRINKE, J.; HAN, D.; RAGKHITWETSAGUL, C. ; HARMAN, M.. **The impact of code review on architectural changes**. IEEE Transactions on Software Engineering, p. 1–19, 2019.

[Palomba et al., 2014] 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: PROCEEDINGS OF THE 30TH INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE AND EVOLUTION (ICSME), p. 101–110, 2014.

[Pantiuchina et al., 2018] PANTIUCHINA, J.; LANZA, M. ; BAVOTA, G.. **Improving code: The (mis) perception of quality metrics**. In: PROCEEDINGS OF THE 34TH INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE AND EVOLUTION (ICSME), p. 80–91, 2018.

[Potvin and Levenberg, 2016] POTVIN, R.; LEVENBERG, J.. **Why google stores billions of lines of code in a single repository**. Communications of the ACM, 59(7):78–87, 2016.

[Prete et al., 2010] PRETE, K.; RACHATASUMRIT, N.; SUDAN, N. ; KIM, M.. **Template-based reconstruction of complex refactorings**. In: PROCEEDINGS OF THE 26TH INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE (ICSM), p. 1–10, 2010.

[Ralph et al., 2020] RALPH, P.; BALTES, S.; ADISAPUTRI, G.; TORKAR, R.; KOVALENKO, V.; KALINOWSKI, M.; NOVIELLI, N.; YOO, S.; DEVROEY, X.; TAN, X.; ZHOU, M.; TURHAN, B.; HODA, R.; HATA, H.; ROBLES, G.; MILANI FARD, A. ; ALKADHI, R.. **Pandemic programming**. Empirical Software Engineering, 25:4927–4961, 2020.

[Revelle et al., 2011] REVELLE, M.; GETHERS, M. ; POSHYVANYK, D.. **Using structural and textual information to capture feature coupling in object-oriented software**. Empirical Software Engineering, 16(6):773–811, 2011.

[Runeson and Höst, 2009] RUNESON, P.; HÖST, M.. **Guidelines for conducting and reporting case study research in software engineering**. Empirical Software Engineering, 14(2):131, 2009.

[Sadowski et al., 2018] SADOWSKI, C.; SÖDERBERG, E.; CHURCH, L.; SIPKO, M. ; BACCHELLI, A.. **Modern code review: A case study at Google**. In: PROCEEDINGS OF THE 40TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE), SOFTWARE ENGINEERING IN PRACTICE (SEIP) TRACK, p. 181–190, 2018.

[Samarthyam et al., 2016] SAMARTHYAM, G.; SURYANARAYANA, G. ; SHARMA, T.. **Refactoring for software architecture smells**. In: PROCEEDINGS OF THE 1ST INTERNATIONAL WORKSHOP ON REFACTORING (IWOR), CO-LOCATED WITH THE 31ST INTERNATIONAL CONFERENCE ON AUTOMATED SOFTWARE ENGINEERING (ASE), p. 1–4, 2016.

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

[Silva et al., 2016b] SILVA, M.; VALENTE, M. T. ; TERRA, R.. **Does technical debt lead to the rejection of pull requests?** In: PROCEEDINGS OF THE 12TH BRAZILIAN SYMPOSIUM ON INFORMATION SYSTEMS (SBSI), p. 248–254, 2016.

[Simons et al., 2015] SIMONS, C.; SINGER, J. ; WHITE, D.. **Search-based refactoring: Metrics are not enough**. In: PROCEEDINGS OF THE 7TH INTERNATIONAL SYMPOSIUM ON SEARCH BASED SOFTWARE ENGINEERING (SSBSE), p. 47–61, 2015.

[Soetens and Demeyer, 2010] SOETENS, Q. D.; DEMEYER, S.. **Studying the effect of refactorings: A complexity metrics perspective**. In:

PROCEEDINGS OF THE 7TH INTERNATIONAL CONFERENCE ON THE QUALITY OF INFORMATION AND COMMUNICATIONS TECHNOLOGY (QUATIC), p. 313–318, 2010.

[Szőke et al., 2015a] SZŐKE, G.; NAGY, C.; FÜLÖP, L.; FERENC, R. ; GYIMÓTHY, T.. **FaultBuster: An automatic code smell refactoring toolset**. In: PROCEEDINGS OF THE 15TH INTERNATIONAL WORKING CONFERENCE ON SOURCE CODE ANALYSIS AND MANIPULATION (SCAM), p. 253–258, 2015.

[Szőke et al., 2015b] SZŐKE, G.; NAGY, C.; HEGEDŰS, P.; FERENC, R. ; GYIMÓTHY, T.. **Do automatic refactorings improve maintainability? An industrial case study**. In: PROCEEDINGS OF THE 31ST INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE AND EVOLUTION (ICSME), p. 429–438, 2015.

[Taibi et al., 2017] TAIBI, D.; JANES, A. ; LENARDUZZI, V.. **How developers perceive smells in source code: A replicated study**. Information and Software Technology, 92:223–235, 2017.

[Tao et al., 2012] TAO, Y.; DANG, Y.; XIE, T.; ZHANG, D. ; KIM, S.. **How do software engineers understand code changes? An exploratory study in industry**. In: PROCEEDINGS OF THE 20TH INTERNATIONAL SYMPOSIUM ON THE FOUNDATIONS OF SOFTWARE ENGINEERING (FSE), p. 1–11, 2012.

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

[Tsantalis et al., 2018] TSANTALIS, N.; CHAIKALIS, T. ; CHATZIGEORGIOU, A.. **Ten years of JDeodorant: Lessons learned from the hunt for smells**. In: PROCEEDINGS OF THE 25TH INTERNATIONAL CONFERENCE ON SOFTWARE ANALYSIS, EVOLUTION AND REENGINEERING (SANER), p. 4–14, 2018.

[Tufano et al., 2017] TUFANO, M.; PALOMBA, F.; BAVOTA, G.; OLIVETO, R.; DI PENTA, M.; DE LUCIA, A. ; POSHYVANYK, D.. **When and why your code starts to smell bad (and whether the smells go away)**. IEEE Transactions on Software Engineering, 43(11):1063–1088, 2017.

[Uchôa et al., 2019] UCHÔA, A.; FERNANDES, E.; FONSECA, B.; DE MELLO, R.; BARBOSA, C.; NUNES, G.; GARCIA, A. ; TEIXEIRA, L.. **On gamifying an existing healthcare system: Method, conceptual model and evaluation.** In: PROCEEDINGS OF THE 1ST INTERNATIONAL WORKSHOP ON SOFTWARE ENGINEERING FOR HEALTHCARE (SEH), CO-LOCATED WITH THE 41ST INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE), p. 9–16, 2019.

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

[Veerappa and Harrison, 2013] VEERAPPA, V.; HARRISON, R.. **An empirical validation of coupling metrics using automated refactoring.** In: PROCEEDINGS OF THE 7TH INTERNATIONAL SYMPOSIUM ON EMPIRICAL SOFTWARE ENGINEERING AND MEASUREMENT (ESEM), p. 271–274, 2013.

[Yamashita and Moonen, 2013] YAMASHITA, A.; MOONEN, L.. **Do developers care about code smells? An exploratory survey.** In: PROCEEDINGS OF THE 20TH WORKING CONFERENCE ON REVERSE ENGINEERING (WCRE), p. 242–251, 2013.

[Yoshida et al., 2016] YOSHIDA, N.; SAIKA, T.; CHOI, E.; OUNI, A. ; INOUE, K.. **Revisiting the relationship between code smells and refactoring.** In: PROCEEDINGS OF THE 24TH INTERNATIONAL CONFERENCE ON PROGRAM COMPREHENSION (ICPC), p. 1–4, 2016.

[Zimmermann et al., 2007] ZIMMERMANN, T.; PREMRAJ, R. ; ZELLER, A.. **Predicting defects for Eclipse.** In: PROCEEDINGS OF THE 3RD INTERNATIONAL WORKSHOP ON PREDICTOR MODELS IN SOFTWARE ENGINEERING (PROMISE), CO-LOCATED WITH THE 29TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE), p. 1–7, 2007.

A
Background Form for All Focus Group Sessions

A study on critical attributes - Background form

Dear participant,

This form is part of a study on critical attributes of software systems proposed by Eduardo Fernandes, a PhD candidate in Informatics at DI/PUC-Rio, in the context of his doctoral thesis.

By filling out and submiting this form, you confirm that you have deliberately and voluntarily accepted to participate in this study. In addition, you allow us to collect, store, analyze, and report your study data in the context of Eduardo's study. Finally, you allow us to keep audio and video records of the discussions that follow this form filling (via Zoom app), in the context of the same study.

We respect your privacy, and your study data will be anonymized for the purpose of data report in scientific publications and dissemination of this study.

Thanks for participating in this study!

Best wishes,

Eduardo Fernandes, PhD candidate at DI/PUC-Rio
Prof. Dr. Marcos Kalinowski, PhD advisor

* Required

1. Email address *

____

2. What is your full name? *
   Please enter your full name, including first name and last name

____

3. What is your highest instruction degree? *

Please enter only the highest degree for which you have a conclusion certificate or a diploma

Mark only one oval.

- ◯ High school
- ◯ Undergraduate
- ◯ Specialization
- ◯ Master's
- ◯ Doctorate
- ◯ Other: ____________________

4. How many years of experience in the industry do you have? *

Please count both 1) development in software companies and 2) development in academia for industry partners

____

5. How many software development projects have you participated in? *

Please count both 1) projects developed in software companies and 2) projects developed in academia for industry partners

____

6. How familiar are you with software metrics? *

Mark only one oval.

- ◯ I have never heard of them
- ◯ I have heard about them but I am not so sure what they are
- ◯ I have a general understanding, but do not use them in my software projects
- ◯ I have a good understanding, and use them in my software projects sometimes
- ◯ I have a strong understanding, and use them in my software projects frequently

7. "I am concerned with improving the quality of source code while participating in software development projects." How much do you agree with this sentence? *

Mark only one oval.

- ◯ Strongly disagree
- ◯ Disagree
- ◯ Indifferent
- ◯ Agree
- ◯ Strongly agree

8. How familiar are you with internal quality attributes? Examples of these attributes are cohesion, complexity, coupling, inheritance, and size. *

Mark only one oval.

- ◯ I have never heard of them
- ◯ I have heard about them but I am not so sure what they are
- ◯ I have a general understanding, but do not analyze them in my software projects
- ◯ I have a good understanding, and analyze them in my software projects sometimes
- ◯ I have a strong understanding, and analyze them in my software projects frequently

B
Feedback Form for All Focus Group Sessions

A study on critical attributes - Feedback form

Dear participant,

This form is part of a study on critical attributes of software systems proposed by Eduardo Fernandes, a PhD candidate in Informatics at DI/PUC-Rio, in the context of his doctoral thesis.

You are invited to fill out and submit this form because you have filled out the Background Form and successfully participated in the discussions promoted via Zoom app.

By filling out and submitting this form, you allow us to collect, store, analyze, and report your study data in the context of Eduardo's study. We respect your privacy, and your study data will be anonymized for the purpose of data report in scientific publications and dissemination of this study.

Thanks, one again, for participating in this study!

Best wishes,

Eduardo Fernandes, PhD candidate at DI/PUC-Rio
Prof. Dr. Marcos Kalinowski, PhD advisor

* Required

1. Email address *

____

2. What is your full name? *

   Please enter your full name, including first name and last name

____

3. "I was confident when discussing the relevance of each critical attribute." How much do you agree with this sentence? *

   Mark only one oval.

   - ◯ Strongly disagree
   - ◯ Disagree
   - ◯ Indifferent
   - ◯ Agree
   - ◯ Strongly agree

4. "I was confident when ranking the critical attributes by relevance." How much do you agree with this sentence? *

   Mark only one oval.

   - ◯ Strongly disagree
   - ◯ Disagree
   - ◯ Indifferent
   - ◯ Agree
   - ◯ Strongly agree

5. "I was comfortable with sharing my opinion during the discussions." How much do you agree with this sentence? *

   Mark only one oval.

   - ◯ Strongly disagree
   - ◯ Disagree
   - ◯ Indifferent
   - ◯ Agree
   - ◯ Strongly agree

6. Besides critical attributes, what else makes it hard for you to either add new features or enhance existing features?

C
Video Transcription of Focus Group Session 1 (Case A)

C.1
Low Class Cohesion: Relevant or Irrelevant – Why?

Participant B – Many aspects of low class cohesion are relative.

Participant A – Yes.

Participant B – There are performance aspects to consider. Thus, everything is relative. It is hard to decide whether low class cohesion is relevant or not.

Participant C – It depends on each case.

Participant B – Well, we may think of many cases we have faced.

Applicant – Do these cases suggest that low class cohesion is irrelevant in practice? If so, you could add a comment to "Irrelevant – Why?," right?

Participant C – Sure.

Participant B – Apparently yes, but I am not sure about that.

Participant A – With respect to performance, if parts of the source code are largely used in the program – as in the case of code duplication – one could gather all those parts in a single class. Thus, one could improve performance. Do you know what I mean?

Participant B – Sure, but doing that will not necessarily improve class cohesion. Well...

Participant A – I know that, but it is not about cohesion after all. If one part of the source code realizes a single feature and the whole program uses it, you could mark it as a critical point in the program. Thus, you would know your development time should use this part of the source code.

Participant B – Would not this decision harm class cohesion?

Participant A – No, it would not. One could just find the critical point in the program only because the code realizing that feature has good cohesion. Otherwise, one would not even find the feature itself, because the source code is scattered.

Participant B – Got it, you are right.

Participant A – In summary, I think class cohesion may improve program performance. Still, I think it depends on things beyond good class cohesion.

Participant B – Yes. In this case, good cohesion is a consequence of gathering features frequently used in the program. Am I right?

Participant A – I would not say that low class cohesion is irrelevant. I cannot stand non-cohesive source code because... I may take a year to understand it.

Participant B – Yes, you are right. When it comes to irrelevance, I do not know.

Participant C – Are these comments all?

Participant B – When one needs to deliver fast a part of the code, perhaps class cohesion is not that important.

Participant A – You are right. Low class cohesion is irrelevant only when time to delivery is short.

Participant C – This is not hard to happen, correct?

Participant A – Exactly! Short time to delivery is very common.

Participant B – We end up being a little paranoid. Low class cohesion annoys us, so we try to fix the source code now and then.

Participant C – Yes, you are right.

Participant A – Yes, but you stop fixing things as soon as possible.

Participant C – Participant A is the one in our team who cares about the beauty of source code the most. Am I wrong?

Participant B – What did you say, Participant C?

Participant C – I said you care a lot about the beauty of source code.

Participant A – Sure, I like to make the source code beautiful.

[...]

Participant A – There is another comment to add to "Irrelevant – Why?". This is not cool, but low class cohesion occurs in the case of legacy code.

Participant B – Yes!

C.2 High Class Complexity: Relevant or Irrelevant – Why?

Participant A – I more or less agree that high class complexity is relevant, so I have no more comments to add.

[...]

Participant A – What were you about to write, Participant B?

Participant B – I was about to write that "complex classes tends to be coupled," but this is not exactly what I mean. Maybe I should improve my comment. Highly complex class tends to be highly coupled, right? I wanted to write it in a better way, but...

[...]

Participant B – With respect to "Irrelevant – Why"...

Participant A – Does coupling mean that one class depends on another class?

Participant B – Yes, that is right.

Participant A – Well, this is not necessarily a problem. Do you know what I mean?

Participant B – Yes, I do. Not necessarily. I guess splitting a simple class into many others is fine, but complex class...

Participant A – I know.

Participant B – ... have too many elements. So, class complexity...

Participant A – What about program maintenance? Where are the comments about it?

[...]

Participant A – Hey, I thought of something!

Participant B – With respect to "Irrelevant – Why?"...

[...]

Applicant [Comment: "Podem gerar lentidão desnecessária"] – Is slowness associated with program performance during the execution of features in highly complex classes?

Participant A – Exactly.

Applicant [Comment: "A complexidade de classes tende ao acoplamento"] – Based on you discussion I suppose that, as one class becomes more complex, it may become more highly coupled – due to the many class inter-dependencies.

[...]

Applicant – One of you said that complex classes are not always highly coupled. Am I right?

Participant B – Yes.

Applicant – If a class is not highly coupled, is high complexity relevant anyway? May I tolerate high complexity if high complexity does not imply high coupling?

Participant B – Yes, you may. Still, high complexity may lead to other problems, right?

Participant A – Sure. Let us think of our class responsible for computing batch route.

Participant B – Yes.

Participant A – That class is considerable complex but it has to be. Anyway, our class is not highly coupled – it only depends on "Map".

Participant B – I guess too complex classes usually become libraries, to be consumed...

Participant A – I think that, when a class is complex...

Participant B – ... because sometimes a class has to be complex to...

Participant A – ... it must be cohesive. If a class is cohesive...

Participant B – ... perform well.

Participant A – ... and complex, this is fine because the class is isolated from the rest of the program. In this case, the class is complex but only used by the rest of the program.

Participant B – Certain classes must perform better, so high complexity...

Participant A – Exactly! High complex and coupled classes...

Participant B – ... is necessary.

Participant A – ... are problematic from my point of view. One could isolate the source code of this class a little more.

C.3 High Class Coupling: Relevant or Irrelevant – Why?

Participant B – Shortly, high class coupling is terrible but sometimes necessary.

[...]

Applicant [Comment: "Propagação de erro"] – Does this comment refer to what nature of error propagation? What is the type of error propagation?

Participant A – It refers to when a class is largely used in the program. The error propagates to all classes trying to use the error-infected class.

[...]

Applicant [Comment: "Object pool"] – Could the author of this comment explain it?

Participant A – Object Pool is a pattern similar to the Singleton design pattern, but targeting multiple objects. A cool pattern, by the way.

Applicant – Do you think this pattern leads to high class coupling?

Supervisor – Yes, Object Pool is a highly coupled class, right? This is the point.

Participant A – Yes. This pattern allows managing many classes at once. Instead of defining a Singleton per class, you can create an Object Pool. This pattern is good because...

Supervisor – For instance, a ten-sized Object Pool allows you to share ten objects at one with the rest of the program.

Participant A – Exactly.

Supervisor – Object Pool naturally leads to high class coupling. On the other hand, this pattern is helpful when computational resources are scarce – for instance, when it is necessary to manage the number of connections between objects.

Participant A – Exactly.

Applicant – Participant A, you were saying something like "this pattern is good because"...

Participant A – Object Pool is good when part of the source code often deals with a specific number of static objects. You may pass Object Pool as parameters to constructors rather than passing each object individually. Thus, this pattern improves the method signatures.

Applicant [Comment: "Definição de entidades"] – Could you please explain this comment?

Participant A – Sure! Consider a class that interrelates with many others, an entity with several relations you need to map.

Applicant – Is this entity a data entity?

Participant A – Yes! Let me further explain it...

Supervisor – Participant A, do you refer to an entity of a conceptual model? An entity with several relations in the conceptual model?

Participant A – Exactly. If you implement the entity before modeling it, the class may become highly coupled – because the class has to be this way.

Applicant [Comment: "Em alguns momentos o acoplamento temporário pode acelerar o desenvolvimento"] – What do you mean by "temporary coupling"?

Participant B – It may be easier to implement source code in a few classes or a single class, even in a procedural manner, when the software project begins. "Temporary coupling" refers to this scenario. Splitting code into many classes is harder, though better. In addition, as the source code becomes more complex and lengthy, it is necessary to split the code to prevent high coupling. Nevertheless, while developing a program, there is no way out. To implement code in a single block may sometimes favor code comprehension.

C.4
Large Class Hierarchy Depth: Relevant or Irrelevant – Why?

Participant A – May I add a comment to both "Relevant – Why?" and "Irrelevant – Why?"? On the one hand, very specialized classes are good because the responsibility of each subclass is well defined. On the other hand, an undisciplined use of specialized classes may create several little classes that could be a single one.

Applicant – In this case, you can duplicate your comment.

Participant A [Comment: "Subclasses muito especializadas..."] – Both comments differ a little, so where do I add this one?

Supervisor – You could modify the comment in "Irrelevant – Why?"...

Participant A – Fine.

Supervisor – ... or any of both comments, so the researchers can keep record of your discussion.

[...]

Applicant [Comment: "Implementar métodos abstratos demais"] – Could you explain this comment a little more?

Participant A – Maybe I expressed myself wrongly. I should have written "a too abstract method." If a class has five or more abstract properties, the inheriting classes should implement at least one property. Thus, if any intermediate inheriting classes do not...

Applicant – Got it! Large hierarchy depth would become a problem because it is necessary to implement methods in the lowest hierarchy levels. Am I right?

Participant A – Yes! Child classes would have to implement all methods because their parents did not do it.

Supervisor – Possibly, large hierarchy depth would make child classes at the lowest hierarchy levels implement several methods, which lacked implementation at the intermediary levels.

Participant A – Exactly! So here is a question: If the child classes must implement too many methods, does the established hierarchy make any sense? Why to define too abstract parent classes if the child classes must implement all features?

C.5
Large Class Hierarchy Breadth: Relevant or Irrelevant – Why?

Participant A – Participant B, I disagree when you say it: "large hierarchy breadth may increase class complexity". What do you mean?

Participant B – What do I mean? I see no problem with large hierarchy depth, but with large hierarchy breadth. Large breadth may increase class complexity, because several features come from a single source. Right?

Participant A – Is not that good?

Participant B – This is polymorphism. I am not saying this is bad, but... Look, I have added a comment to "Irrelevant – Why?" saying that large breadth may help decouple and split program features. Still, I see no negative effect of large hierarchy depth.

Supervisor – Does complexity refer to the whole set of classes?

Participant B – For me, hierarchy breadth is more complex than hierarchy depth. Maybe you see it differently.

Supervisor – All right! Please comment whatever you think about it.

Participant A – Exactly. I just wanted to know you opinion better. That is all.

[...]

Applicant [Comment: "Reaproveitamento de propriedades utilizadas em todas as entidades (Modelo ER)"] – Could you explain this comment a little more?

Participant A – This comment refers to the conceptual model, similar to my previous comment on large hierarchy depth. Consider a data identifier shared by the tables of a database. One could create a class to deal with that identifier, to be inherited by other classes. Thus, it is not necessary to define static identifiers: one classes handles with this issue.

[...]

Applicant [Comment: "Muitas subclasses podem usar método que foi desenvolvido na classe mãe"] – Maybe, large hierarchy breadth is not a problem if there is a reuse benefit...

Participant B – I see large hierarchy breadth as polymorphism. I see large hierarchy depth similarly, but I thought of this issue with respect to breadth – this is why I commented on class complexity.

C.6 Large Class Size: Relevant or Irrelevant – Why?

Participant A – Very large classes are possibly non-cohesive.

[...]

Participant A – Have anyone commented that it is hard to test large classes?

Participant C – We left this comment for you to add, Participant A! We expected you to remember this argument.

Participant A – The "hard to test" argument usually appears very fast.

Participant C – We knew that, right? But we said: "Participant A will remember".

Participant A – "After all, he remembered of testing before. He will remember it again".

[...]

Participant A – Urgency in delivering a program is always an issue. Am I right?

Participant C – Yes, urgency in delivery is always there to decrease program quality.

Participant A – Exactly!

[...]

Participant A – Very large classes may be problematic as Proofs of Concept (PoC) and, still, they are approved, right?

Participant C – For sure.

Participant A – A PoC class simply proves a concept.

Participant C – Yes.

Participant A – Thus, this class is not in a production environment. Still...

Participant C – The problem to forward a PoC class to production environment. Am I right?

Participant A – Yes. This is very common.

Participant C – Exactly!

Participant A [Comment: "PoCs que realmente são PoCs"] – Let me clarify my point of view: If you expect to forward a PoC class to production environment, you will not make it too large. If you know that class is just a PoC class, fine.

Participant C – Yes, this is true.

[...]

Participant A – I just read a comment that says "hard to understand". I will delete the comment "hard to read," because "understand" is prettier and both terms are the same.

[...]

Participant A – The whole "Relevant – Why?" area could be summarized by "large class is hard to maintain".

Participant C – Exactly.

Participant B – Large classes are hard, complicated, and complex to maintain.

[...]

Applicant [Comment: "Difícil organização"] – Could you further explain this comment?

Participant C – I wrote it. A class with five or more elements is too large. "Organization" refers to the internal organization of source code in each method rather than the organization of all elements in a class. If a class and its methods are all very large, organizing the content within each method is harder.

Participant A – Would organize methods within a class be similarly hard? It would be hard to know what methods call one another and group these methods. This is the least one should do in large classes. Unfortunately, these tasks are hard to perform when methods call one another in a zigzag.

Applicant – Great!

Participant A – In this case, the integrated development environment (IDE) does not help us that much.

[...]

Applicant [Comment: "Métodos que realmente são muito complexos"] – Does this comment refer to the program domain? Programs belonging to certain domains, which provide specific services or features, tend to be more complex than other programs.

Participant A – Yes. One could isolate a very complex part of the source code in a single class, but this create a large class. You are at least isolating that complex part of the code, so having a large class is not a problem. I think the same reasoning applies to program domains. If you wish to prevent from scattering a very complex part of the code in a program, you could gather the code and define a specific error point like "this class may have many errors," thereby making simpler the rest of the program.

Applicant – Cool!

[...]

Applicant [Comment: "Quando as funcionalidades são muitas e difícil de desacoplarem"] – Does this comment mean that, sometimes, it is fine to keep large a class implementing several features which are hard to decouple?

Participant C – Yes! This scenario has something to do with program domain. It is harder to maintain two classes that are equivalent to a single class.

Applicant – Understood. Excellent!

[...]

Applicant [Comment: "Urgência na entrega"] – Some of you said this comment is always applicable, as if urgency in delivery affects the program quality as a whole. You also said that short time to delivery is always a problem. I found this comment interesting. Would you like to add a comment like "urgency in delivery" to the other critical quality attributes? Or is urgency in delivery particularly relevant in the case of large classes?

Participant A – I guess we added this comment here because urgency in delivery really stands out in this particular case. For instance, we would not spend time defining class hierarchies i four time to delivery is short, because hierarchies are complex to define.

Applicant – Got it.

Participant A – So...

Applicant – Does anyone want to add up something on this matter?

Participant C – No. Participant A is right. Maybe we commented on urgency in delivery when this argument really stands out.

Participant A – Well, we commented on urgency in delivery while discussing cohesion, coupling...

Participant C – When better organizing the source code is necessary, we did not comment on urgency in delivery. In cases like hierarchy depth and breadth, we discuss that there are to many hierarchical levels. We would never define too many levels if time to delivery is short. Am I right?

Participant A – Yes.

[...]

Applicant [Comment: "PoCs que realmente não PoCs"] – Could you further explain this comment?

Participant A – Working on PoC classes requires fast delivery because you simply prove concepts. Successfully proving a concept with a large class is fine, because you will theoretically not send it to production environment. You will improve the PoC class before sending it to production. Unfortunately, the theory does not apply to practice: PoC classes often become production classes. Still, if you create a PoC class only for proof purposes, and you must do it quickly in two days or a week, fine.

Applicant – Got it. Excellent!

Participant C – One more comment about urgency in delivery. I think that urgency in delivery possibly affects these critical attributes: low class cohesion, high class complexity, high class coupling, and large class size.

Applicant – Perfect, thanks!

D
Video Transcription of Focus Group Session 2 (Case B)

D.1
Low Class Cohesion: Relevant or Irrelevant – Why?

Participant B – I usually pay attention to the theme each class implement – for instance, data base access. Then I ask myself if everything implemented within the class is associated with database access Are the method parameters and the class attributes all associated with database?

Participant A – It is easier to maintain a program where each class has a clear team, because those who will maintain the program...

Participant B – Yes! In the case of...

Participant A – ... will know exactly where to find a theme and make any necessary chances.

Participant B – ... problems with the database access, the developers will search for classes implementing this particular theme.

Participant A – That is it!

Participant B – On the other hand, bugs affecting a non-cohesive class may be harder to find.

[...]

Participant B – When low class cohesion occurs, it may be necessary to debug and trace program behaviors to find where the bug is.

Supervisor – Perfect! There seem to be comments associated with this discussion.

Participant C – Yes.

Supervisor – For instance, Participant A said that low class cohesion is relevant because it makes it hard to maintain a program. Am I right?

Participant A – Yes!

Supervisor – Participant B said that low class cohesion makes it hard to find defects. Correct? You may add comments on these arguments to "Relevant – Why?".

Participant C – Participant B, you said that low class cohesion makes it hard to find defects. Do you assume that a class is non-cohesive because the source code realizing that particular theme is scattered in other classes?

Participant B – Exactly! Finding the bug will require carefully debugging and tracing source code. Would not this process be easier if the classes of a program are cohesive? Sometimes, one can find a bug based on the name of the classes or methods.

Participant C – Right.

Participant B – Highly cohesive classes may speed up the search for bugs, because the methods implementing each theme are close to one another within the program. Besides that, suppose you want to know the impact of changes applied to method on the rest of the program. You could use the integrated development environment (IDE) to know where the method has been used the most. Depending on the class cohesion, close classes use this method the most, so it is easier to assess the impact of changes.

[...]

Participant B [Comment: "Se a classe for utilitária"] – I may say that low class cohesion in utility classes is irrelevant. Certain classes host features that fit no other part of the program – they are the so-called utility classes. Examples are classes responsible for handling objects, converting or editing data, counting... all useful in many part of the program. In these cases, low class cohesion is irrelevant.

[...]

Participant B – Any other comments to add?

Supervisor – Some of you said something about program defects, right?

Participant C – Yes. I was about to add a comment like "low class cohesion makes it hard to track defects".

Participant B – You are right, but your comment looks like part of another comment on the difficulty of maintaining a program. The same reasoning applies to the difficulty of evolving a program. Both defect tracking and evolution are part of program maintenance.

Supervisor – In this case, you could add a new comment like "low class cohesion makes it hard to perform evolutive maintenance". Am I right?

Participant B – Is it possible to draw relations between comments in the MURAL tool?

Supervisor – Drawing relations is not the purpose of this experiment.

Participant B – Okay.

Supervisor – Do not you worry about it.

Applicant – Participant B, if you think that two comments are interrelated, say it out loud, so we can take notes of it. Is that okay?

[...]

Participant B [Comment: "Dificulta a rastreabilidade de defeitos" e "Dificulta evolução"] – Both defect tracking and evolution are part of program maintenance, right?

Supervisor – Okay, cool!

Participant B – I thought of two issues regarding the irrelevance of low class cohesion. One of them I wrote in my comment on utility classes; I forgot the other one. When is low class cohesion irrelevant?

Participant A – I am also struggling to think of cases where low class cohesion is irrelevant.

Applicant – Could you speak a little louder, Participant A?

Participant B – Right!

Participant A – Sorry, is my voice that low?

Participant B – Low class cohesion may be irrelevant if the metrics used to capture cohesion do not consider the theme of a class – I guess this is something automated tools could never do. If a class that realizes a particular calculus but its methods share nothing but a common theme, automatically stating the class is non-cohesive may be irrelevant.

[...]

Participant B – How could I document this argument?

[...]

Participant B [Comment: "Se os métodos compartilharem conceitos..."] – Suppose I gathered several types of calculus within a single class. If each method has different parameters, an automated tool would probably ignore it. Instead, the tool would say the methods neither are interrelated nor share parameters or attributes, even though the developers sees this scenario differently.

Participant A – Do you think that low class cohesion is irrelevant if the program is very small?

Participant B – Yes, especially in the case of serverless environments, where programs are often fragmented in several features and little code blocks. Thus, low class cohesion is natural.

Participant A – Right.

Applicant [Comment: "Dificulta evolução"] – What do you mean by evolution?

Participant B – Evolution includes, for instance, deciding where to implement new features. I would easily find the best place to add features in a cohesive class. Web programs usually have a controller layer, whose code is the first to run after triggering a browser request. Adding the new feature to the controller layer is reasonable if the feature validates data forms, for instance. If classes within the controller layer are cohesive, I will know where to add the feature. Otherwise, any less experienced or familiar developer could choose an appropriate place – like the service layer – and worsen class cohesion.

Applicant – Could this mistake even harm the program architecture?

Participant B – Exactly! Feature addition would depend on the guidance of either program architects or experienced developers.

Applicant – Understood.

[...]

Applicant [Comment: "Se os membros compartilharem conceitos..."] – I suppose concepts refer to program features, right?

Participant B – Well, concept at the program or business level.

Applicant – Got it.

[...]

Applicant – Participant A asked if low class cohesion might be irrelevant for very small programs. Participant B agreed. Could you add a comment on this matter to "Irrelevant – Why?"?

Participant B – Come on, Participant A! You had the idea.

Participant A – Oh... [UNINTELLIGIBLE].

Participant B – I suggest adding a comment like "too fragmented program architectures may cause low class cohesion". Thus, low class cohesion is natural.

Applicant – Is this a problem to evolving program features?

Participant B – Yes.

Applicant – Right.

Participant B – I mean, low class cohesion will be irrelevant because the program architecture led me to that. Low class cohesion remains relevant anyway, but this is an architectural issue. If the program architecture led me to create non-cohesive classes, low class cohesion reports will not bother me.

Applicant – Still, low class cohesion will make it somehow difficult to evolve features. Regardless of a tool report, do you see cases where low class cohesion is a problem?

Participant B – Sure. As I said, low class cohesion remains relevant, but I am constrained by the program architecture.

Applicant – Got it.

Participant B – In a sense, this problem is irrelevant only because there is nothing to do about it. Still, low class cohesion is relevant by itself.

Applicant – Right. Are you still going to write about very small programs, Participant A? Did you change your mind on low class cohesion being irrelevant if the program is small?

Participant A – No, I still think that low class cohesion is irrelevant if the program is very small. I have been thinking of small programs that evolve little. Maintaining this program will require little changes once the program has only a few features. The program will grow little and, then, low class cohesion is irrelevant.

D.2
High Class Complexity: Relevant or Irrelevant – Why?

Participant A – I think high class complexity is relevant for the same reason that low class cohesion is. I mean relevant for program maintenance. High complexity is intricate when you must maintain program features, but...

Participant C – For sure, especially you must maintain a program originally implemented by others.

Participant A – Yes, but...

Participant B – I think high complexity is irrelevant if I am the one who created the class, ...

Participant C – Exactly!

Participant B – ... preferably not long ago. It is fine if I have created such a monster and have to maintain it next week. High complexity is very irrelevant because I still have it clear in my mind what the class does. However, high complexity will become very relevant three or six months after, because not even I will remember what the context of the class.

Participant A – Exactly!

Applicant – Interesting! Participant A e Participant B, you can add a comment on this matter if you will.

Participant C – Participant B, you could write that high class complexity may be irrelevant in the case of short-term maintenance.

Participant A – Yes, similar to the case of low class cohesion.

[...]

Participant A – Certain classes have to be complex, right? It may be hard to split certain business rule into different methods of a class.

Applicant – Cool! Do you remember any situation in the context of your project that illustrates this scenario?

Participant B – Oh, yes! We have our own "pet monsters".

Participant A – Some classes end up being complex because they implement too many business rules. I have implemented one of these classes and I always find it hard to change it. Still, this class is complex only because of it implements several business rules and has changed a lot along with the program development.

Participant B – Highly complex classes usually have a "father" and a "mother" who will take care of them.

Participant A – Yes, and take care of them forever.

Participant B – A complex class is like a child. Its parent should take care of it until maturing. Our programs have parts implementing themes with clear parents: who implemented each part.

Applicant – Okay.

Participant B – A class implements several features often because of its context. Certain features require several variables, which probably become too many parameters because of their scattering in program. While handling with hierarchies, the developer may forget that these variables could be more accessible if implemented in a single object. Consider a calculus that creates variables for reuse in later steps of calculus. If the source code is too fragmented, it may be hard to identify those variables.

Applicant – Do you mean that the high complexity of the calculus class favors reuse?

Participant B – Yes.

Applicant – Okay.

Participant B – Fragmenting a feature too much may help reusing certain parts of the source code in other program features. However, if the calculus itself needs to reuse those parts, such fragmentation may be a problem.

[...]

Applicant [Comment: "Dificulta o entendimento da funcionalidade"] – Are you talking about the feature implemented by the complex class?

Participant A – Yes.

[...]

Applicant [Comment: "Dificulta criar novas regras de negócio"] – What do you mean by creating new business rules?

Participant A – I am talking about the program evolution, that is, when you either create a new feature or change an existing feature. I refer to when you need to change a particularly complex feature – for instance, a feature with several conditional branches.

Applicant [Comment: "Autor único e última manutenção foi a pouco tempo"] – Does "single author" refer to the author of the whole program or the complex class only?

Participant B – The complex class only.

Applicant – Okay. Does "last maintenance" refers to the last maintenance performed on any part of the program?

Participant B – No, it does refer to the last maintenance performed on the complex class.

Applicant – Was this recent maintenance performed on the program as a whole or on the highly complex class in specific?

Participant B – I will not care very much if an automated tool says that one class with a single author and recently changed is complex.

D.3 High Class Coupling: Relevant or Irrelevant – Why?

Participant B – I am a little skeptical about the coupling measurement itself, ...

Participant C – Somehow...

Participant B – ... especially Coupling between Objects (CBO). Bootstrap classes in our programs may have terrible CBO values...

Participant A – Yes.

Participant B – ... because our bootstrap classes are responsible for data persistence. Thus, they communicate with each class that interacts with the database. If a program has ten data entities, our bootstrap class will communicate with the classes that interact with each of the ten data entities. As a result, CBO for this class is terrible! However, high coupling in this case will affect the same level, that is, data persistence classes only. No coupling will occur with external services, for instance. Therefore, I think that coupling computed by simply counting the number of coupled classes is unreliable. It is important to know the nature of each coupled class. Our bootstrap class communicates with them or twenty persistence classes but okay: all classes belong to the same level.

Participant A – In other words, the bootstrap class is cohesive.

Participant B – On the other hand, if the bootstrap class implements unrelated features...

[...]

Applicant – Participant B, you said earlier: "if coupled classes are all in the same hierarchical level, high class coupling is fine". Am I right?

Participant B – Right.

Applicant – Then you can add a comment about this topic.

[...]

Applicant – Participant B also said, and Participant A agreed, that the relevance of high class coupling depends on context. You could add a comment on this topic as well. You discussed an example about relevance depending on context. What is the name of that class from your program that is highly coupled?

Participant B – Bootstrap, which is responsible for managing test data.

Applicant – Is this class highly coupled by default?

Participant B – Yes, but the class only deals with other data persistence data.

Participant A – Right.

Applicant – Perfect! So, please, add a comment about this topic.

Participant A – The bootstrap class is cohesive, right?

Applicant – Would you say that highly cohesive classes might be highly coupled without major problems?

Participant A – Yes!

Applicant – Right.

Participant B – It is natural for a cohesive class to be highly coupled as well. Thus, high coupling will not cause further problems.

Applicant – Understood.

Participant A – Do you want me to write this comment, Participant B?

Participant B – I thought of writing something like "if the class is cohesive, its high coupling is not that relevant".

Participant A – Fine!

[...]

Applicant – Participant C, I see you surrounding the "Relevant – Why?" area of our mural. What do you think about high class coupling?

Participant C – I think of the opposite scenario. I agree with this: "if a class is cohesive, its high coupling is not that relevant." However, ...

Participant B – Well, we could add to "Relevant – Why?" one comment that opposes to each comment from "Irrelevant – Why?".

Participant C – Exactly! What problems high coupling could cause on a non-cohesive class?

Applicant – Interesting!

Supervisor – Please comment on your arguments if you will.

[...]

Participant A – Maybe high coupling becomes more relevant when it co-occurs with other issues. I think that high coupling is not a problem itself. However, for instance, if high coupling affects a non-cohesive class, high coupling becomes a problem.

Applicant – Interesting!

Participant B – I am adding a comment like "fault tolerance." Once a coupled class uses several components, each component is a potential point of failure.

Applicant – This is also interesting!

Participant B – If a single class manages email, archives e database, you should treat different universes of failures. As a result, the fault handling code will have different natures. In other words, high coupling is relevant when implementing fault tolerance.

Participant C – I think that, the more you organize source code in reusable classes, the higher is class coupling. Am I right?

Applicant – Could you repeat what you said, please?

Participant C – The more you organize the source code structure in reusable classes, the more um tend to create highly coupled classes.

Applicant – This is interesting. In summary, high coupling is not very problematic when it derives from a reuse-oriented code structure, right?

Participant C – Exactly!

Participant B – There is a concept we did not discuss yet. There is a difference between using variables provided by interfaces and concrete classes.

Applicant – Could you further explain this comment?

Participant B – One class of our program is responsible for managing documents. Persistence occurs in a file system. In our PCs, persistence occurs via file management system and the documents sent via browser are saved in a folder. Differently, in our servers, we use a REST service for managing documents. We carefully coupled those objects that manage documents: you can choose in runtime the persistence mechanism to use.

Applicant – Cool.

[...]

Applicant [Comment: "Se o CBO for alto, mas entre classes em níveis diferentes"] – Why is high class coupling problematic when affecting classes in different hierarchical levels?

Participant B – Because the highly coupled class probably implements several features. The class may be responsible for managing emails, files, database... Maybe the class is overloaded with responsibilities.

Applicant – Does it occur because the highly coupled class is far below in the class hierarchy and has several parents, grandparents, and so forth? Are there many superclasses above this highly coupled class?

Participant B – Not actually. I am not thinking about class hierarchies after all. Suppose a highly coupled class has many attributes that the class features use.

Applicant – Okay.

Participant B – For instance...

Applicant – Does this scenario occur because the highly coupled class has inherited many resources from other classes?

Participant B – Not really. The highly coupled class uses many other classes to realize several features altogether.

Applicant – What does this have to do with the hierarchical level of the highly coupled class – if I understood your use of the term "level" correctly?

Participant B – Well...

Applicant – I made this question because you mentioned "level" in your comment.

Participant B – When I wrote "in the same level" in my comment, I was referring to either a package or a theme of a given program.

Participant A – I thought you were referring to a given concept of the program. Am I right?

Participant B – Yes, you are. If a class uses other ten classes, but all these classes deal with data persistence, high class coupling is fine.

Applicant – Okay.

Participant B – But if two out of the ten classes manage email, two other classes manage REST services and two other manage databases, high class coupling becomes a problem.

Applicant – Understood. Participant A, do you want to comment on this matter?

Participant A – No. I just wanted to talk about the levels that you have mentioned. I thought "level" referred to the "concept" that a class realizes...

Applicant – Oh, yes.

Participant A – ... rather than to a class hierarchy level.

Applicant – So this "level" has nothing to do with class hierarchy, right? Got it.

Participant B – You are right. "Level" refers to either the package where the class is...

Applicant – Understood.

Participant B – ... or the module that contains the class.

Applicant – Okay.

[...]

Applicant [Comment: "Se classe não foi altamente coesa"] – Some of you said that, if a class is not highly cohesive – that is, the class has a few unrelated elements – high class coupling becomes a problem. Thus, more than one problem affects the class: low class cohesion and high class coupling. Am I right?

Participant B – If high coupling and low cohesion co-occur, there is clear need for refactoring the class in order to split responsibilities.

[...]

Applicant [Comment: "Tolerância a falhas/tratamento de erros"] – Could the author of this comment further explain it?

Participant B – High class coupling is relevant while implementing fault tolerance. I mean, highly coupled classes depend on many others to realize their features and of those classes many have problems of different natures. It implies handling with different faults.

Applicant – Perfect!

Participant A – Participant B, are you saying that highly coupled classes have many scattered points of faults?

Participant B – Yes. In this case, a single class may be vulnerable to problems of different natures.

Applicant – This scenario occurs because the highly coupled class uses several functions, each possibly affected by faults. It will be barely possible handling all these faults, even though this is necessary.

Participant B – Correct. There is this policy in our project for notifying, via email, each time you change a certain entity. A document is attached to the notification email. After performing a change on an entity, you must create a document, persist it, and send it by email.

Applicant – Who receives the notification email?

Participant B – The person responsible for the changed entity.

Applicant – Okay.

Participant B – Shortly, our business rule says this: every time the entity changes, we notify the person who is responsible for the entity.

Applicant – Is this person the class author?

Participant B – No, this person is a business or application expert.

Applicant – I still did not get it. Whom should you notify? A developer?

Participant B – No. A system user.

Applicant – Right. Interesting.

Participant B – A program entity triggers events in the database. Each event has a responsible, who received an email notification for each change associated with this event. In case of changing an entity, we create, persist, and send bye mail a notification document. The problem here is the need for handling failures related to different issues: sending emails, persisting documents and so forth. This creates business conflicts all the time. The source code that copes with these issues is complex.

Applicant – Interesting.

Participant B – The code complexity increased because the class is highly coupled, and the class is highly coupling due to a business rule associated with changing entities. Creating fault tolerance mechanisms required handing errors of different natures and notifying the use about possible faults.

Applicant – Got it.

[...]

Applicant [Comment: "Variáveis de tipos concretos em vez de interface"] – How does this issue related with high class coupling?

Participant B – In the previous example, the browser sends an archive that a class must persist using the currently configured persistence mechanism: either the PC file system or another document storage system. Using variables from interfaces allows the class to choose a persistence mechanism in runtime, thereby making it clear the source code. The program injects in runtime the implementation corresponding to the chosen mechanism.

Applicant – Right, but what does it have to do with high class coupling becoming a problem to program evolution?

Participant B – On the other hand, using variables from concrete types requires source code duplication. For instance, I would duplicate code in the class the previously mentioned class for addressing browser requests to allow configuring the persistence mechanism.

Applicant – Would the source code duplication increase the class coupling?

Participant B – Not really. Such duplication is due to the high coupling of variables from concrete types.

Applicant – Got it.

[...]

Applicant [Comment: "Pode auxiliar o reuso"] – Could you further explain it?

Participant C – I think that defining well divided and reusable classes increases coupling because you must call the same source code from many classes, thereby creating dependencies.

Applicant – Does it occur, for instance, in the case of utility classes?

Participant C – Sure.

Applicant – Do you mean that this scenario applies to class that are called everywhere in a program?

Participant C – Exactly. But I think that...

Applicant – And certain classes have several utility functions, right?

Participant C – Yes. Still, in this case, I do not see high class coupling as a problem.

Applicant – Interesting.

D.4
Large Class Hierarchy Depth: Relevant or Irrelevant – Why?

Participant B – I rarely found too deep class hierarchies in the majority of projects I participated in. We rarely use more than one or two hierarchical levels.

Applicant – Interesting.

Participant B – You may find deeper hierarchies, with four or five levels, when our classes make use of libraries. This is because libraries tend to define hierarchies aimed at a better design.

Applicant – Are these hierarchies in libraries deep?

Participant B – Yes, or at least deeper than usual. The library author may have defined three or four hierarchical levels. We create at most one or two hierarchical levels when inheriting resources from library classes.

Applicant – That is, large hierarchy depth is not necessarily a problem if caused by using libraries.

Participant B – You are right, because other developers – the library authors – are responsible for maintaining the majority of hierarchy levels. Deep hierarchies are relevant when we authored them. However, if inherited from a library, it is not that relevant because other developers will maintain it. In addition, the library authors are usually concerned about cohesion, complexity, etc. Thus, class hierarchy depth is not that problematic.

Applicant – Got it. But, in this case, considering...

Participant B – But...

Applicant – Sorry. Please, continue.

Participant B – Deep hierarchies that we created are relevant and make us ask this: How to extend our program? At what hierarchical level to implement a new feature? Duplicating code may be necessary. [UNINTELLIGIBLE].

Applicant – Do you see cases in which deciding where in the class hierarchy to add a new feature is challenging? It may be necessary to duplicate code.

Participant B – Yes, I may have to duplicate source code because of that.

Applicant – Interesting. You could add a comment on this topic. Participant C and Participant A, have you ever had any experiences with deep class hierarchies? Where these deep hierarchies a problem for you?

Participant A – I agree with Participant B. I think I never had to maintain classes too deep in the class hierarchy levels for reporting any challenges.

Applicant – Very interesting! What about you, Participant C?

Participant C – I never faced such a situation. However, I think one could find it hard (or complex) to understand the source code depending on the class hierarchy depth. It may be challenging to track from what hierarchical level certain behaviors of a class come from.

Applicant – Understood.

Participant C – As Participant B said, [UNINTELLIGIBLE].

Applicant – Right. Would you like to document this issue? I will give you two more minutes.

[...]

Applicant – Well, I will give you two more minutes. Please, Participant C, add a comment on the complexity of understanding the source code, where the features are... [...] Participant B and Participant A, feel free to add up or discuss something else.

Participant A – The most relevant comment so far is the one Participant B added about how hard is to know where to add a new feature.

[...]

Applicant – You have discussed that deep class hierarchies may be problematic while adding or evolving features – for instance, by making it hard to understand the code structure.

Participant B – Excuse me. What if we start our discussion from the comment "DIT is rarely too high" of "Irrelevant – Why?"?

Applicant – Sure!

Participant B – I requested that because class hierarchy depth is rarely large, especially because Depth of Inheritance Tree (DIT) values are rarely high. However, when DIT values are high, the class hierarchy depth becomes relevant.

Applicant – Got it.

Participant B – Well, this phenomenon rarely occurs, but when it does...

Applicant – Understood.

[...]

Applicant [Comment: "Pode provocar duplicação de código"] – Is it correct to assume that duplicating source code is a problem?

Participant B – Yes, it is.

D.5 Large Class Hierarchy Breadth: Relevant or Irrelevant – Why?

Participant B – Is Number of Children (NOC) the only metric for capturing large class hierarchy breadth?

Applicant – There are other metrics, actually.

Participant B – I am trying to think of another one.

Applicant – Large class hierarchy breadth may affect the descendants of a given class. Thus, we may extend the counting of child classes from the next hierarchical level to all the following levels – that is, the sub-tree whose root is the current class.

Participant B – Okay.

Applicant – This could be another way of computing hierarchy breadth, okay?

Participant B – Okay. In fact, NOC only computes the direct children of a given class. Got it.

Applicant – NOC is the simplest example, because it only computes the number of direct children rather than grandchildren, great-grandchildren, and so forth.

[...]

Participant B – Large hierarchy breadth is irrelevant when the child classes do not overwrite the attributes and methods of a class. However, if the attributes and methods defined by the parent class and only used by the child classes...

Applicant – That is, if these attributes and methods are used rather than overwritten.

Participant B – ... and none of the child classes overwrites them...

Applicant – Interesting.

Participant B – In this case, the parent class encapsulates its features. Thus, there is no problem if a class has twenty or thirty child classes. However, if the child classes overwrite the inherited attributes and methods, you may have a great complexity – it may be necessary to verify what child classes perform overwriting.

Participant A – [UNINTELLIGIBLE].

Participant B – Large class hierarchy breadth is irrelevant if the child classes cannot change features inherited by the parent class, because the implementation focuses on the parent class. Otherwise, large breadth becomes relevant. There may be a metric based on the number of child classes that overwrite features.

Applicant – Oh, yes. This metric exists.

Participant B – That would be an interesting metric for me. If many child classes redefine attributes and methods of the parent class, a high metric value may be a problem. It may be necessary to conduct refactoring for reorganizing features or, maybe, create a new hierarchical level for separating child classes performing overwriting differently.

Applicant – Got it.

Participant B – Large class hierarchy breadth is irrelevant if the child classes do not redefine attributes and methods inherited from the parent class. Am I right?

Participant A – Do you mean redefining behaviors realized by the parent class?

Participant B – Yes, because I am assuming that the child classes make use of the concrete features of the parent class. Obviously, child classes must implement features defined by the parent class as interfaces. Still, in this case... The object-orientation vocabulary is escaping me.

Applicant – Express yourself freely. Participant A and Participant C, have you seen cases of problems regarding number of child classes and hierarchy breadth? For instance, while implementing polymorphism or fixing bugs, are there cases where large class breadth became a problem?

Participant B – One more thing: large class hierarchy breadth becomes more problematic when co-occurs with large class hierarchy depth.

Applicant – Interesting. Participant A and Participant C, do you want to add u something?

Participant A – I would add a comment to "Relevant – Why?" about the issue of tracking behaviors in the program.

Applicant – Oh...

Participant A – I mean, large hierarchy breadth is relevant when you have to know what child classes implement a given behavior.

Applicant – Fine.

Participant B – Does this case occur if a child class redefines a behavior of its parent class?

Participant A – Yes.

Participant C – If the child class redefines a behavior inherited from its class parent, maintaining the parent class should carefully consider such redefinition. Am I right?

Applicant – Got it. Feel free to add comments on this topic, because I gave you two and a half more minutes. And do not you worry about having similar comments. Each participant adds his own comment and we will analyze all comments later.

[...]

Participant B – I would like to add something about large class hierarchy depth. I see this problem recurrently affecting programs in functional programming.

Applicant – How does it occur?

Participant B – There are several property passing among components. Certain components belong to the lowest hierarchical levels and require properties located at highest levels. You must either pass or copy properties among functions because defining global variables is non-recommended in functional programming. Passing parameters may prevent side effects because the universe of available properties is constrained by each function. I think large hierarchy depth also affects functional programming. You may have to adapt several components to bring properties from the highest to the lowest hierarchical levels. Intermediate components must propagate properties they do not use, thereby generating false positives in the analysis of hierarchy depth – that is, several components simply passing properties.

Applicant – Understood.

Participant B – In other words, several components supposedly use a property, but most of them only pass the parameter because the distance between components that hold and actually use a property is large.

Applicant – This situation is very interesting.

Participant B – This scenario often occurs in Web frameworks for front-end development, which uses functional programming a lot. Developers used to implement object-orientation in JavaScript, which is weird, but people already replaced object-orientation with functional programming. Although we abandoned some basic concepts of object-orientation, we use composition to distribute features as an alternative to inheritance.

Applicant – Got it.

Participant B – We establish inheritance through composition. [UNINTELLIGIBLE].

Applicant – Great!

Participant B – In functional programming, composition causes similar problems to large hierarchy depth.

Applicant – Cool!

Participant B – Object-oriented programs rarely have too deep and broad hierarchies in practice.

Applicant – Curiously, there seems to be a consensus among you all on this matter, right?

Participant B – Yes. Using composition in functional programming leads to similar problems to large hierarchy depth.

D.6 Large Class Size: Relevant or Irrelevant – Why?

Participant B – I believe that classes with hundreds of characters per line are also large classes. I mean, a method with ten conditional branches in a single line is lengthy.

Applicant – Right. You all feel free to start the discussions.

Participant A – Fine. I do not think that size is relevant in itself. If a class has several methods, each with well-defined responsibilities, it will be easier to understand what each method does, perform changes, and evolve the program.

Participant C – Large size does not imply high class complexity.

Participant A – Yes, right?

[...]

Participant A [Comment: "Monitor 'pequeno"'] – Wow, someone added a comment about small monitor size!

Participant B – Yes, but this is not a joke. Depending on the monitor size adopted by the developer, the number of scrolls can make the developer lose his working context while programming. The developer may be like "what does the part of source code I just read but disappeared?" The larger a class, the worse may be this problem. On the other hand, when does large class size is not that relevant?

Applicant – Interesting...

Participant B – Large classes rarely lack complexity, coupling, and cohesion problems.

Participant A – Yes, but now we are only discussing class size, right?

Applicant – You are discussing whether class size is relevant or not.

Participant A – Okay.

Participant B – If you add the content of a figure to the source code of a class, the class will be lengthy. Once the added content is just one more code block to scroll, this is fine. The programmer would read the variable declaration and assignment with about a hundred or two hundred lines, and scroll it until it finds some code parts of interest. Thus, the scrolled lines will not be a problem for the developer.

Applicant – Got it.

Participant B – Another scenario is associated with comment line within the source code. Our program has a class with about 90% of commented lines.

Applicant – Would you say everything is fine in this case?

Participant B – This class has two hundreds of commented lines and other fifteen non-commented lines. The class has only fifteen lines after all. The source file of this class may be large, but this is irrelevant. To become a problem, problems that are more serious should affect the class, like high complexity, high coupling, and low cohesion.

Applicant – Participant C, do you have...

Participant B – Besides that, in the case of utility classes, large class size is irrelevant, is not it? The same reasoning applies here.

Applicant – Are you saying that large classes are forgiven if they are utility classes?

Participant B – I am saying this factor increases the tolerance of large class size.

Applicant – Right!

Participant B – If the programming language under use does not allow implementing more cohesive source code, and I need to duplicate code, a large class size is forgivable. Not forgivable, but irreparable.

Applicant – Do you have any examples to discuss with us?

Participant B – Sure. Clear Architecture recommends not exposing classes that represent data entities to external parties. While querying the database and returning a class instance, avoid converting this class into a JSON format file and sending it via browser. Instead, define a class for realizing this use case and then manipulate the entities of interest. Depending on the language, you may have to exchange data field by field. If a data entity has several fields, you may have twenty or thirty lines only for exchanging data.

Applicant – Understood.

Participant B – In summary, a limitation of language or library led to a verbose source code.

Applicant – Right.

Participant B – Verbose languages will make the class to become large. Another example is associated with exception handling in Java – which is complicated and requires several code lines even in simple cases. I will pay less attention to large classes due to the limitations of this language.

Applicant – Got it.

Participant B – Although large class size remains problematic, this is something imposed by the language.

Applicant – Got it.

Participant B – Although large class size is relevant, there is nothing to do about it due to other factors. The problem is relevant, but there is no possible correction.

Applicant – Right. It is good to know that, although large classes may be problematic, their correction is not always worthwhile.

Participant B – That is it.

Applicant – Interesting.

[...]

Applicant [Comment: "Se não for complexa"] – Did Participant A write this comment?

Participant A – Yes.

Applicant – At the beginning of the discussion, we said something like "I think that a large class is not a problem if the class is well documented, structured..."

Participant A – You are right.

Applicant – Does the complexity you mentioned in your comment refer to that argument?

Participant A – Yes.

Applicant – Great!

Participant A – This case is similar to the one regarding the comment about "having cohesive methods." If the methods of a class are well divided, each with well-defined features, you or another developer in charge of maintaining the class will not struggle to understand it.

Applicant – Excellent!

Participant A – That is it.

Applicant – Perfect. Class size is associated with the comment on "having cohesive methods" because both indicators do not indicate a relevant problem.

[...]

Applicant [Comment: “Dificilmente ocorre sozinho”] – I think Participant B was the one who wrote large class size is a problem that rarely occurs in isolation. I found interesting this comment. Do you all agree with this comment? Participant A, Participant B, and Participant C, do you agree that large class size rarely occurs in isolation?

Participant B – Well, I agree with myself.

Participant A – I agree.

Applicant – Sure, Participant B!

Participant C – No doubt.

Participant A – Agreed.

Applicant – Perfect.

[...]

Applicant [Comment: “Se o editor conseguir colapsar grandes blocos”] – In this case, class size is a development environment problem rather than a program problem.

Participant B – If comments compose a significant part of the class, you may configure the integrated development environment (IDE) to collapse comments and method documentations by default. You can expand the collapsed content if you will. When you have large line blocks at the top of the classes, the IDE may collapse these blocks automatically.

Applicant – This is the case of Apache projects, for instance. Many classes have hundreds of commented lines regarding copyright and license.

Participant B – Another example is associated with static object initialization. The IDE may automatically collapse initialization as well. By doing that, large classes become manageable. Do you know what I mean?

Applicant – Got it.

Participant B – IDEs may help managing side effects of having lengthy source files.

Applicant – Perfect. Based on what we have...

Participant A – I would like to add up something about the comment “Rarely occurs in isolation.” Should we place this comment in the “Relevant – Why?” area? I am asking this exactly because large class size is a problem that does not occur in isolation.

Applicant – Understood.

Participant B – No. Solving the other problems affecting a large class makes its length irrelevant.

Participant A – Oh, no...

Participant B – After all, I have already improved class complexity, coupling, and cohesion, right?

Applicant – Participant A, do you think it is worth replicating this comment in “Relevant – Why?”?

Participant A – Yes, because my understanding opposes to the current one. That is, large classes end up being relevant exactly because this problem does not occur in isolation.

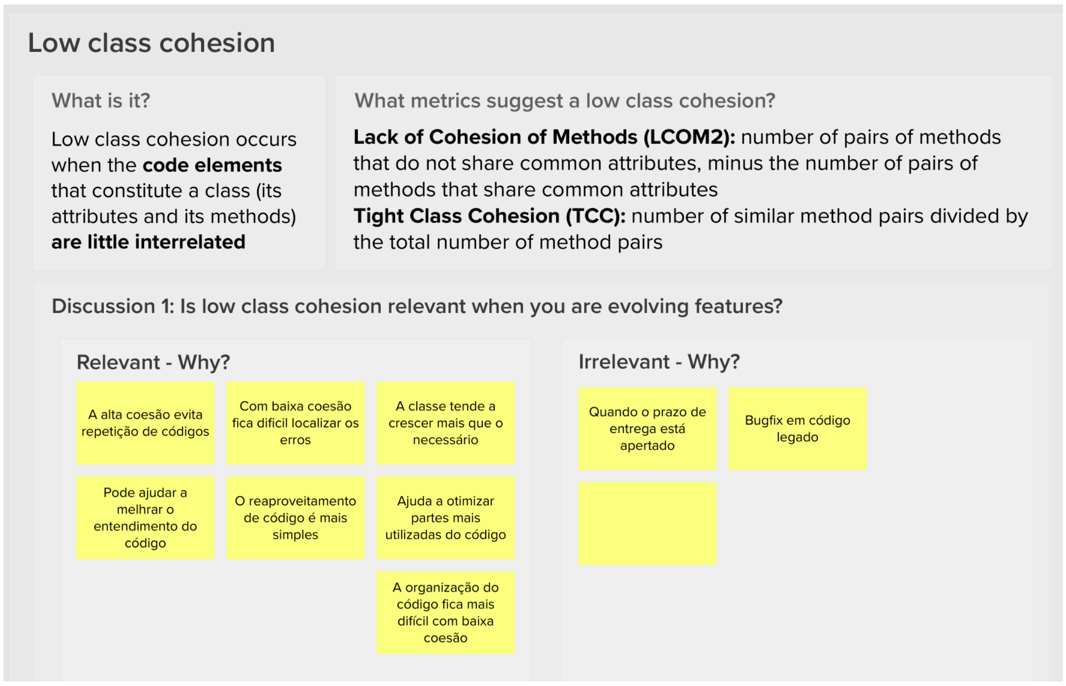
Applicant – Got it.

Participant A – Yes, if...

Applicant – In this case, feel free to replicate the comment.

E
MURAL Prints of Focus Group Session 1 (Case A)

Figure E.1 depicts the raw discussions (in Brazilian Portuguese) among participants of Focus Group Session 1 on **low class cohesion**.



Low class cohesion

What is it?

Low class cohesion occurs when the **code elements** that constitute a class (its attributes and its methods) **are little interrelated**

What metrics suggest a low class cohesion?

Lack of Cohesion of Methods (LCOM2): number of pairs of methods that do not share common attributes, minus the number of pairs of methods that share common attributes
Tight Class Cohesion (TCC): number of similar method pairs divided by the total number of method pairs

Discussion 1: Is low class cohesion relevant when you are evolving features?

Relevant - Why?

A alta coesão evita repetição de códigos

Com baixa coesão fica difícil localizar os erros

A classe tende a crescer mais que o necessário

Pode ajudar a melhrar o entendimento do código

O reaproveitamento de código é mais simples

Ajuda a otimizar partes mais utilizadas do código

A organização do código fica mais difícil com baixa coesão

Irrelevant - Why?

Quando o prazo de entrega está apertado

Bugfix em código legado

Figure E.1: Raw Discussions on Low Class Cohesion (Case A)

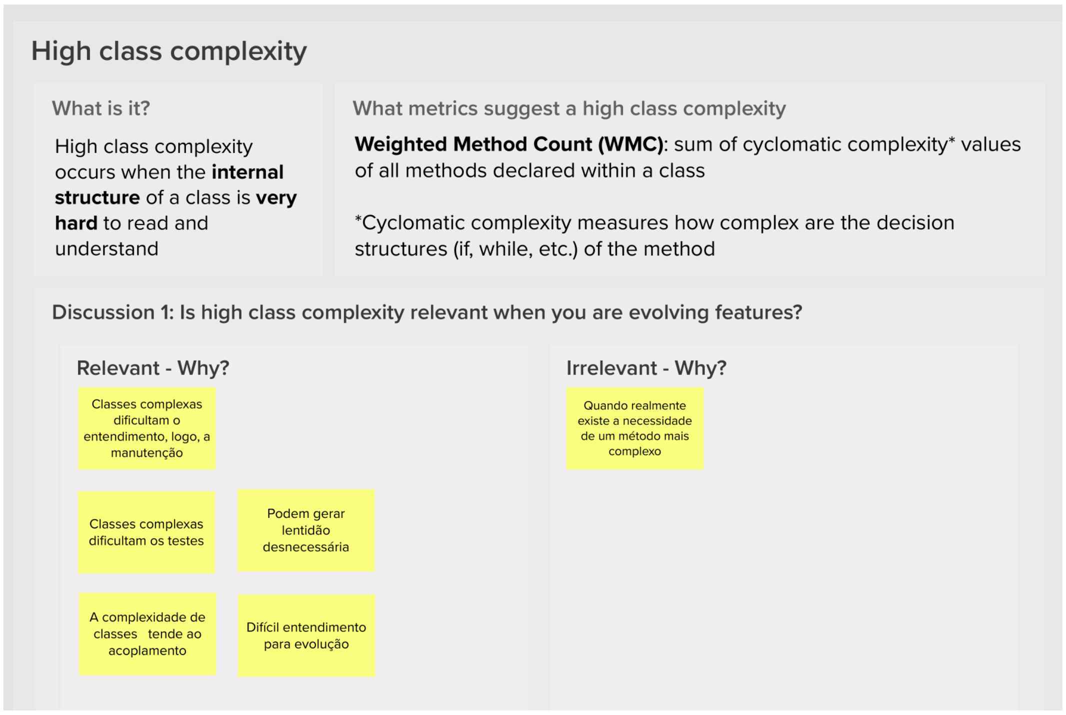
Figure E.2 depicts the raw discussions (in Brazilian Portuguese) among participants of Focus Group Session 1 on **high class complexity**.

Figure E.3 depicts the raw discussions (in Brazilian Portuguese) among participants of Focus Group Session 1 on **high class coupling**.

Figure E.4 depicts the raw discussions (in Brazilian Portuguese) among participants of Focus Group Session 1 on **large class hierarchy depth**.

Figure E.5 depicts the raw discussions (in Brazilian Portuguese) among participants of Focus Group Session 1 on **large class hierarchy breadth**.

Finally, Figure E.6 depicts the raw discussions (in Brazilian Portuguese) among participants of Focus Group Session 1 on **large class size**.



High class complexity

What is it?

High class complexity occurs when the **internal structure** of a class is **very hard** to read and understand

What metrics suggest a high class complexity

Weighted Method Count (WMC): sum of cyclomatic complexity* values of all methods declared within a class

*Cyclomatic complexity measures how complex are the decision structures (if, while, etc.) of the method

Discussion 1: Is high class complexity relevant when you are evolving features?

Relevant - Why?

Classes complexas dificultam o entendimento, logo, a manutenção

Classes complexas dificultam os testes

Podem gerar lentidão desnecessária

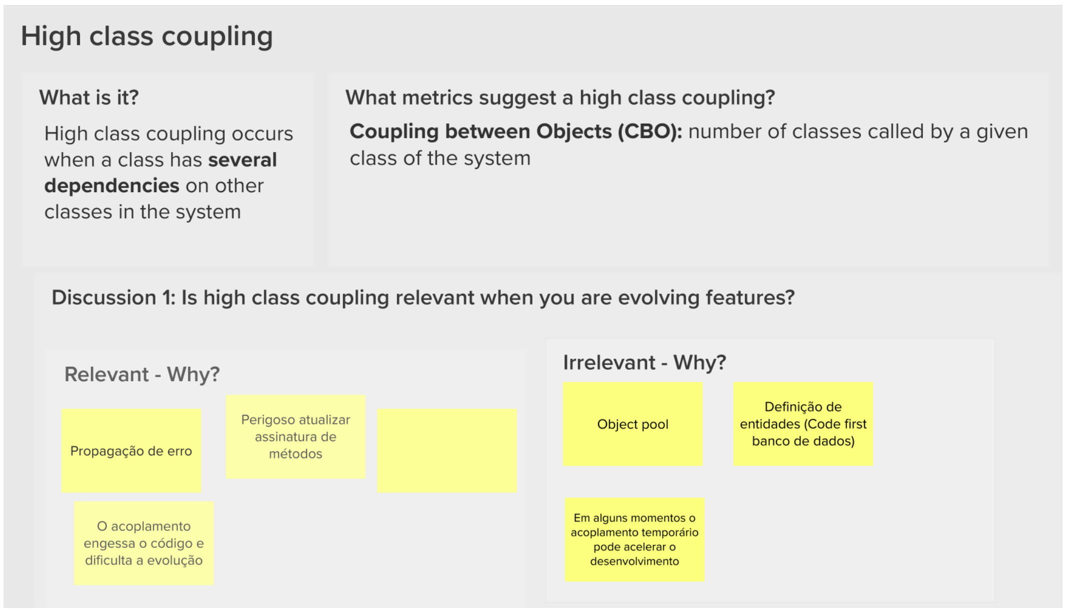
A complexidade de classes tende ao acoplamento

Difícil entendimento para evolução

Irrelevant - Why?

Quando realmente existe a necessidade de um método mais complexo

Figure E.2: Raw Discussions on High Class Complexity (Case A)



High class coupling

What is it?

High class coupling occurs when a class has **several dependencies** on other classes in the system

What metrics suggest a high class coupling?

Coupling between Objects (CBO): number of classes called by a given class of the system

Discussion 1: Is high class coupling relevant when you are evolving features?

Relevant - Why?

Propagação de erro

Perigoso atualizar assinatura de métodos

O acoplamento engessa o código e dificulta a evolução

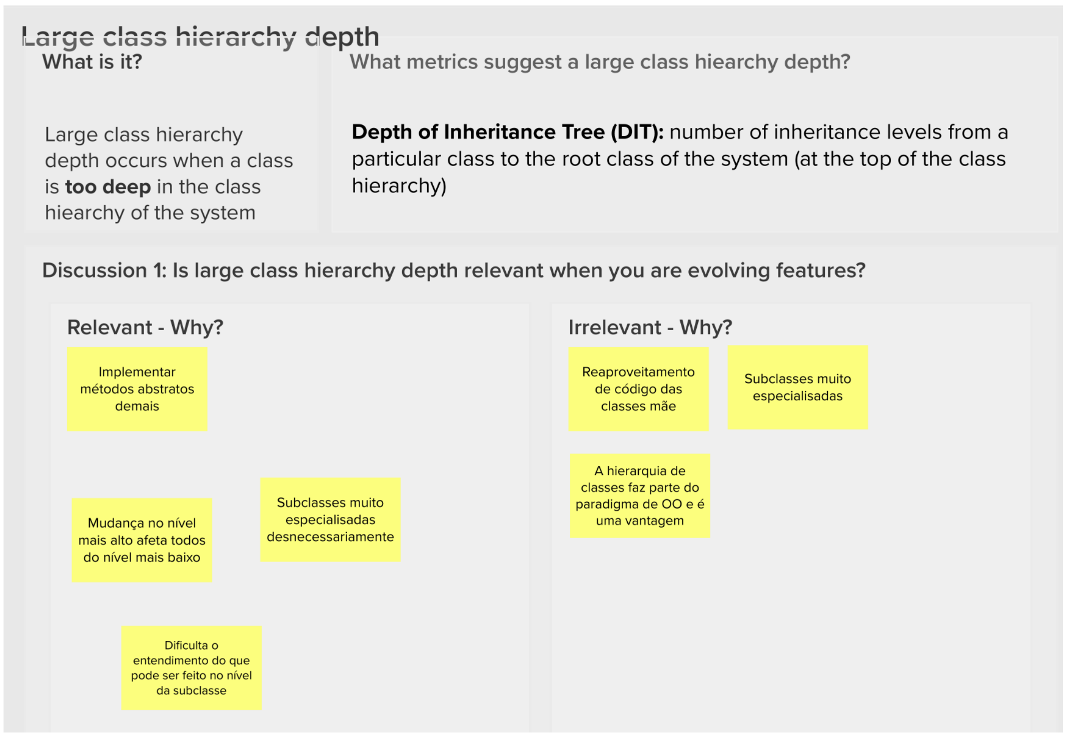
Irrelevant - Why?

Object pool

Definição de entidades (Code first banco de dados)

Em alguns momentos o acoplamento temporário pode acelerar o desenvolvimento

Figure E.3: Raw Discussions on High Class Coupling (Case A)



Large class hierarchy depth

What is it?

Large class hierarchy depth occurs when a class is **too deep** in the class hiearchy of the system

What metrics suggest a large class hiearchy depth?

Depth of Inheritance Tree (DIT): number of inheritance levels from a particular class to the root class of the system (at the top of the class hierarchy)

Discussion 1: Is large class hierarchy depth relevant when you are evolving features?

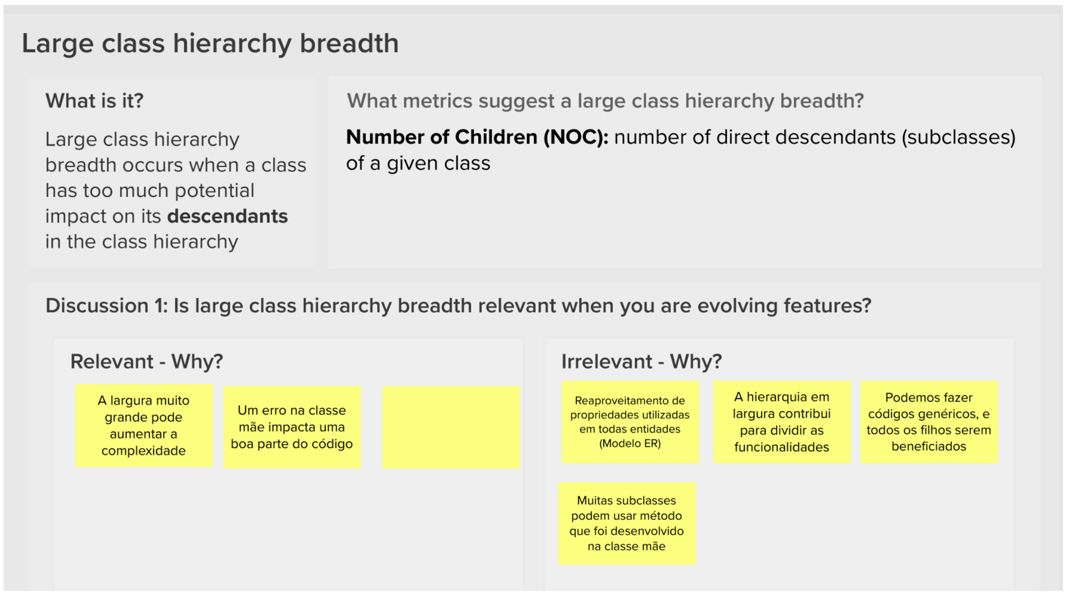
Relevant - Why?

- Implementar métodos abstratos demais
- Subclasses muito especialisadas desnecessariamente
- Mudança no nível mais alto afeta todos do nível mais baixo
- Dificulta o entendimento do que pode ser feito no nível da subclasse

Irrelevant - Why?

- Reaproveitamento de código das classes mãe
- Subclasses muito especialisadas
- A hierarquia de classes faz parte do paradigma de OO e é uma vantagem

Figure E.4: Raw Discussions on Large Class Hierarchy Depth (Case A)



Large class hierarchy breadth

What is it?

Large class hierarchy breadth occurs when a class has too much potential impact on its **descendants** in the class hierarchy

What metrics suggest a large class hierarchy breadth?

Number of Children (NOC): number of direct descendants (subclasses) of a given class

Discussion 1: Is large class hierarchy breadth relevant when you are evolving features?

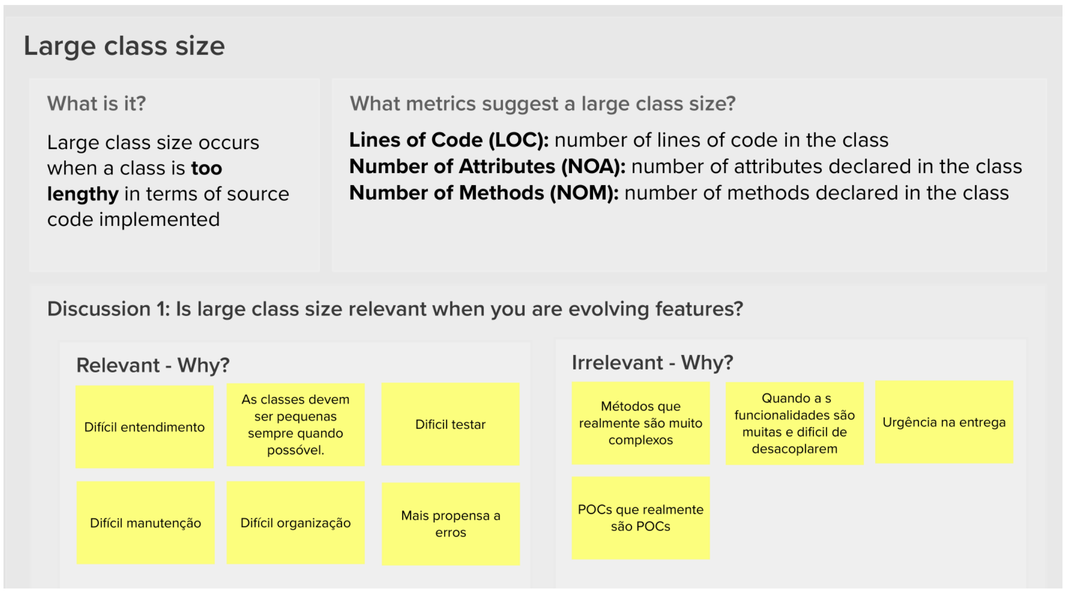
Relevant - Why?

- A largura muito grande pode aumentar a complexidade
- Um erro na classe mãe impacta uma boa parte do código

Irrelevant - Why?

- Reaproveitamento de propriedades utilizadas em todas entidades (Modelo ER)
- A hierarquia em largura contribui para dividir as funcionalidades
- Podemos fazer códigos genéricos, e todos os filhos serem beneficiados
- Muitas subclasses podem usar método que foi desenvolvido na classe mãe

Figure E.5: Raw Discussions on Large Class Hierarchy Breadth (Case A)



Large class size

What is it?

Large class size occurs when a class is **too lengthy** in terms of source code implemented

What metrics suggest a large class size?

Lines of Code (LOC): number of lines of code in the class
Number of Attributes (NOA): number of attributes declared in the class
Number of Methods (NOM): number of methods declared in the class

Discussion 1: Is large class size relevant when you are evolving features?

Relevant - Why?

- Difícil entendimento
- As classes devem ser pequenas sempre quando possóvel.
- Dificil testar
- Difícil manutenção
- Difícil organização
- Mais propensa a erros

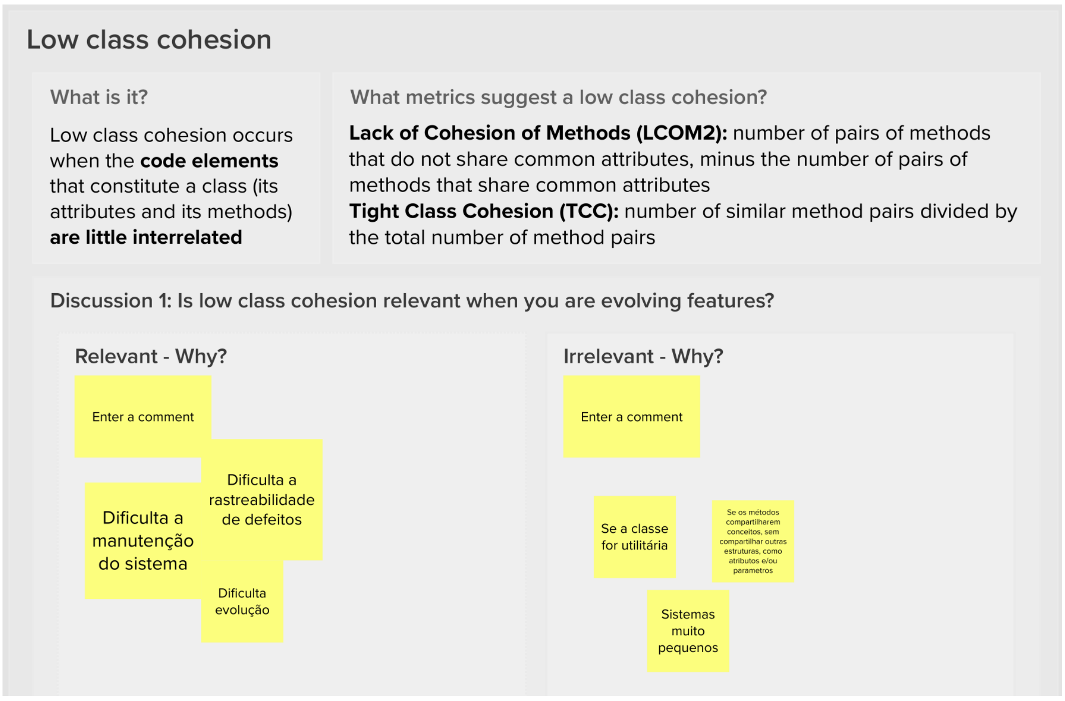
Irrelevant - Why?

- Métodos que realmente são muito complexos
- Quando a s funcionalidades são muitas e dificil de desacoplarem
- Urgência na entrega
- POCs que realmente são POCs

Figure E.6: Raw Discussions on Large Class Size (Case A)

F
MURAL Prints of Focus Group Session 2 (Case B)

Figure F.1 depicts the raw discussions (in Brazilian Portuguese) among participants of Focus Group Session 2 on **low class cohesion**.



Low class cohesion

What is it?

Low class cohesion occurs when the **code elements** that constitute a class (its attributes and its methods) **are little interrelated**

What metrics suggest a low class cohesion?

Lack of Cohesion of Methods (LCOM2): number of pairs of methods that do not share common attributes, minus the number of pairs of methods that share common attributes
Tight Class Cohesion (TCC): number of similar method pairs divided by the total number of method pairs

Discussion 1: Is low class cohesion relevant when you are evolving features?

Relevant - Why?

Enter a comment

Dificulta a rastreabilidade de defeitos

Dificulta a manutenção do sistema

Dificulta evolução

Irrelevant - Why?

Enter a comment

Se a classe for utilitária

Se os métodos compartilharem conceitos, sem compartilhar outras estruturas, como atributos e/ou parametros

Sistemas muito pequenos

Figure F.1: Raw Discussions on Low Class Cohesion (Case B)

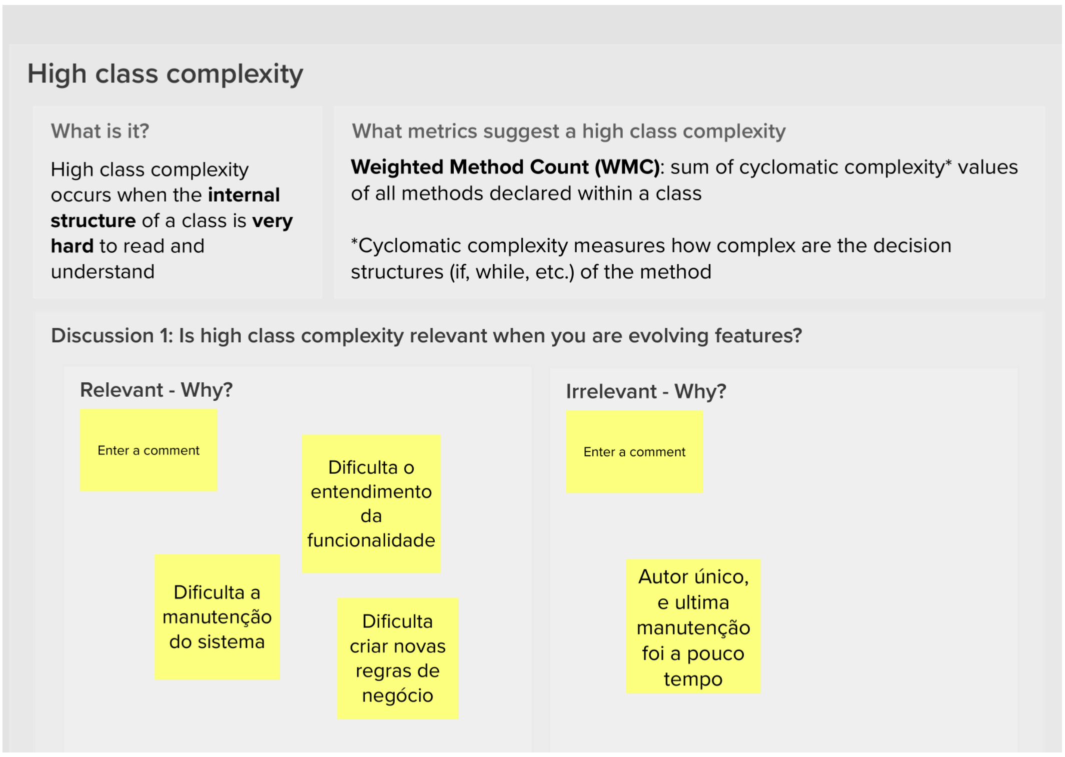
Figure F.2 depicts the raw discussions (in Brazilian Portuguese) among participants of Focus Group Session 2 on **high class complexity**.

Figure F.3 depicts the raw discussions (in Brazilian Portuguese) among participants of Focus Group Session 2 on **high class coupling**.

Figure F.4 depicts the raw discussions (in Brazilian Portuguese) among participants of Focus Group Session 2 on **large class hierarchy depth**.

Figure F.5 depicts the raw discussions (in Brazilian Portuguese) among participants of Focus Group Session 2 on **large class hierarchy breadth**.

Finally, Figure F.6 depicts the raw discussions (in Brazilian Portuguese) among participants of Focus Group Session 2 on **large class size**.



High class complexity

What is it?

High class complexity occurs when the **internal structure** of a class is **very hard** to read and understand

What metrics suggest a high class complexity

Weighted Method Count (WMC): sum of cyclomatic complexity* values of all methods declared within a class

*Cyclomatic complexity measures how complex are the decision structures (if, while, etc.) of the method

Discussion 1: Is high class complexity relevant when you are evolving features?

Relevant - Why?

Enter a comment

Dificulta o entendimento da funcionalidade

Dificulta a manutenção do sistema

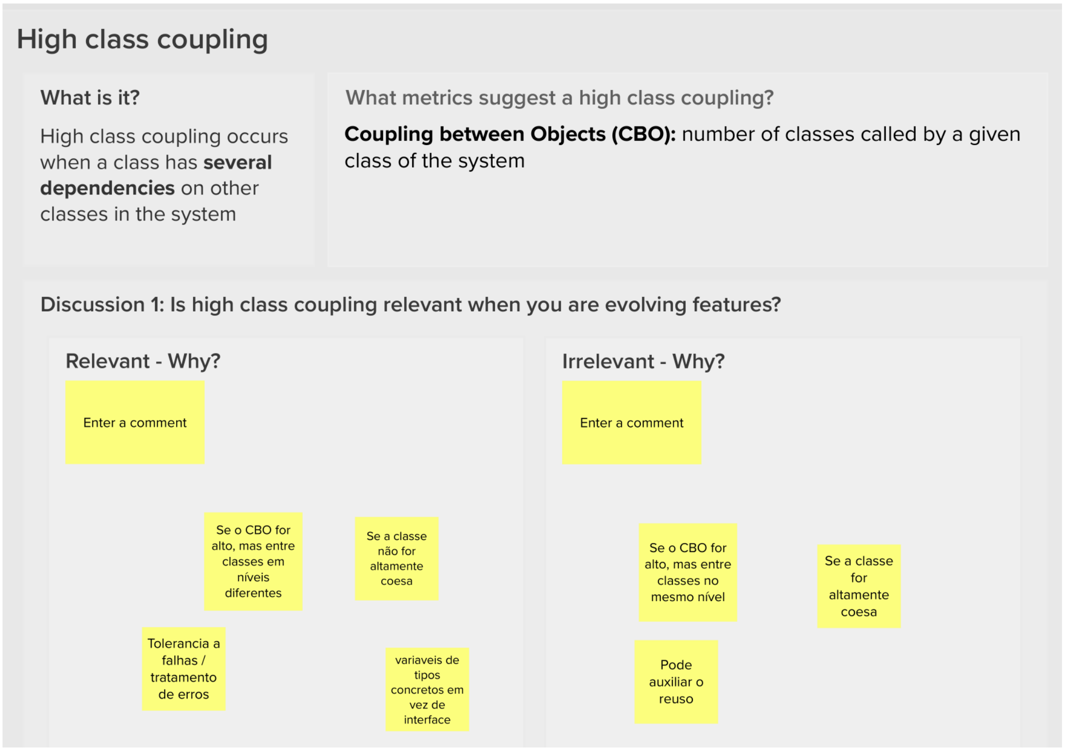
Dificulta criar novas regras de negócio

Irrelevant - Why?

Enter a comment

Autor único, e ultima manutenção foi a pouco tempo

Figure F.2: Raw Discussions on High Class Complexity (Case B)



High class coupling

What is it?

High class coupling occurs when a class has **several dependencies** on other classes in the system

What metrics suggest a high class coupling?

Coupling between Objects (CBO): number of classes called by a given class of the system

Discussion 1: Is high class coupling relevant when you are evolving features?

Relevant - Why?

Enter a comment

Se o CBO for alto, mas entre classes em níveis diferentes

Se a classe não for altamente coesa

Tolerancia a falhas / tratamento de erros

variaveis de tipos concretos em vez de interface

Irrelevant - Why?

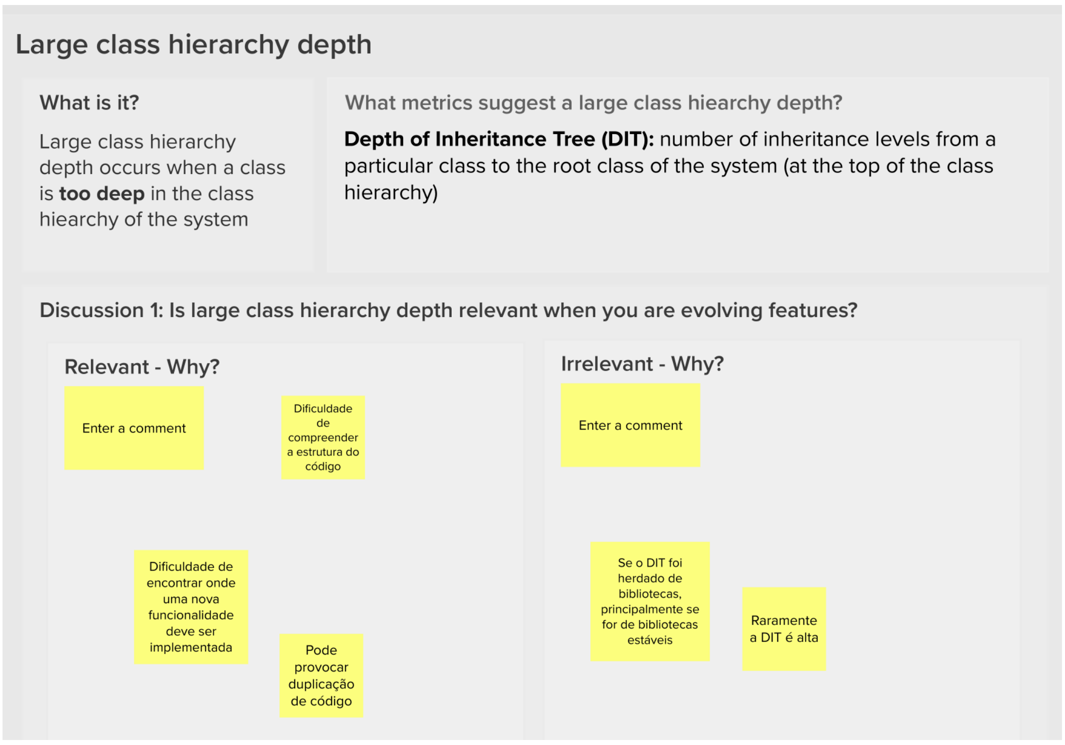
Enter a comment

Se o CBO for alto, mas entre classes no mesmo nível

Se a classe for altamente coesa

Pode auxiliar o reuso

Figure F.3: Raw Discussions on High Class Coupling (Case B)


Large class hierarchy depth

What is it?

Large class hierarchy depth occurs when a class is **too deep** in the class hiearchy of the system

What metrics suggest a large class hiearchy depth?

Depth of Inheritance Tree (DIT): number of inheritance levels from a particular class to the root class of the system (at the top of the class hierarchy)

Discussion 1: Is large class hierarchy depth relevant when you are evolving features?

Relevant - Why?

Enter a comment

Dificuldade de compreender a estrutura do código

Dificuldade de encontrar onde uma nova funcionalidade deve ser implementada

Pode provocar duplicação de código

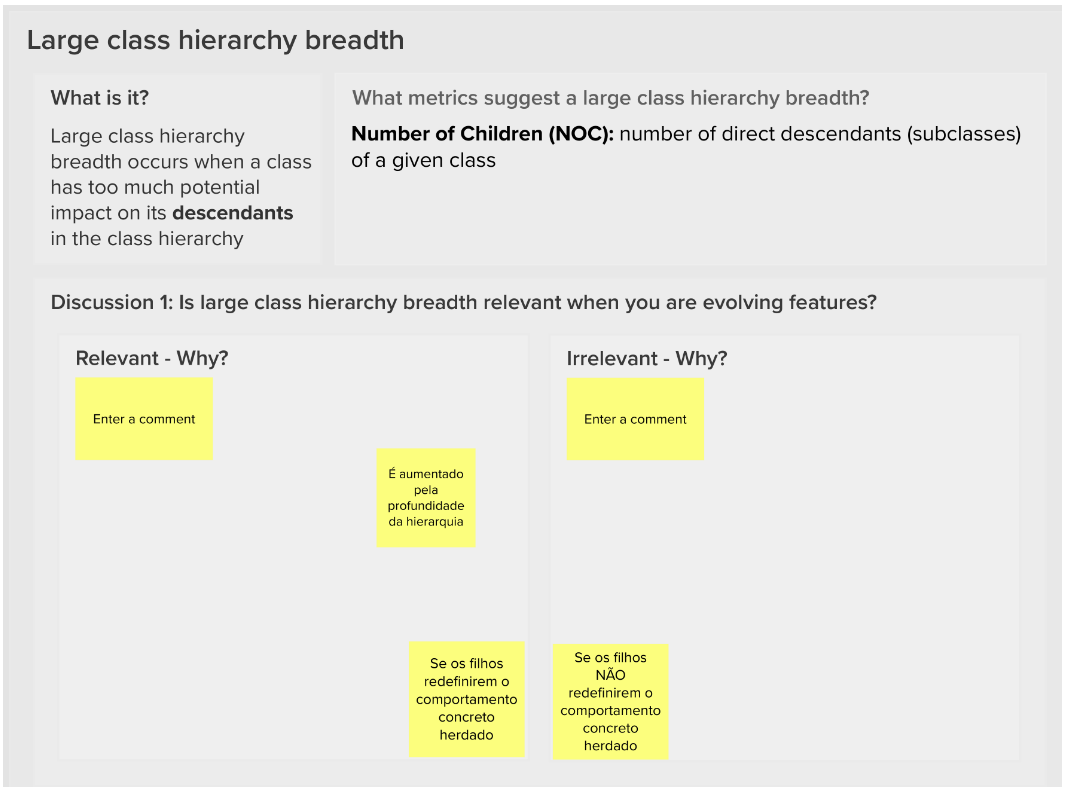
Irrelevant - Why?

Enter a comment

Se o DIT foi herdado de bibliotecas, principalmente se for de bibliotecas estáveis

Raramente a DIT é alta

Figure F.4: Raw Discussions on Large Class Hierarchy Depth (Case B)


Large class hierarchy breadth

What is it?

Large class hierarchy breadth occurs when a class has too much potential impact on its **descendants** in the class hierarchy

What metrics suggest a large class hierarchy breadth?

Number of Children (NOC): number of direct descendants (subclasses) of a given class

Discussion 1: Is large class hierarchy breadth relevant when you are evolving features?

Relevant - Why?

Enter a comment

É aumentado pela profundidade da hierarquia

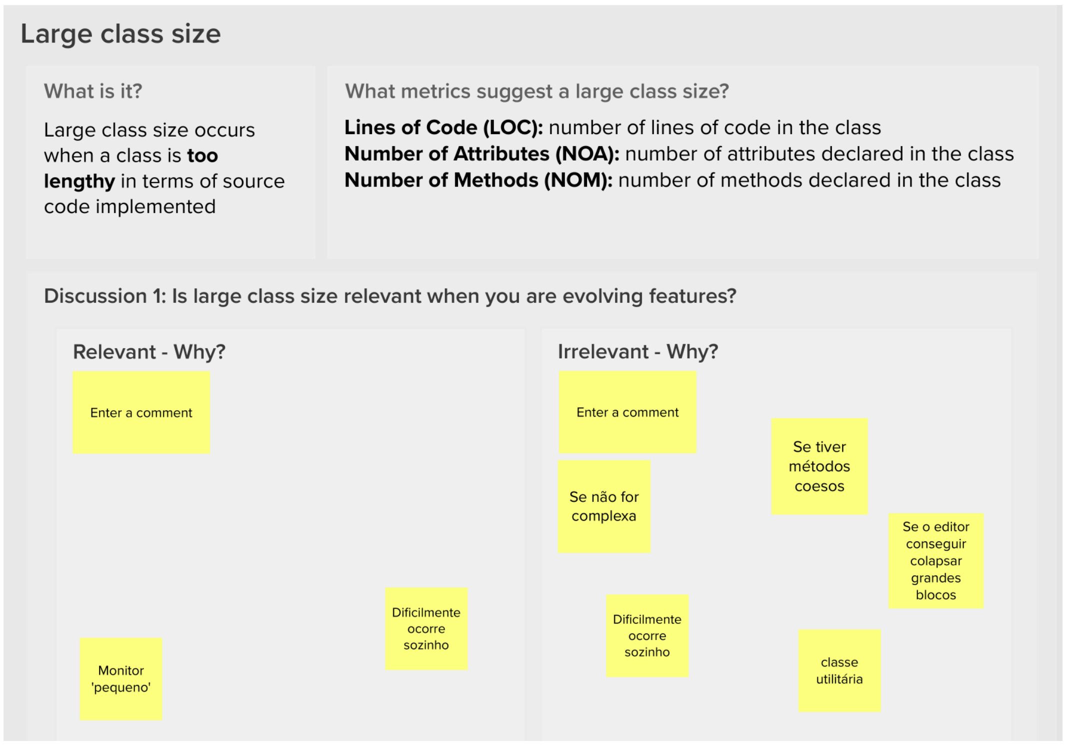
Se os filhos redefinirem o comportamento concreto herdado

Irrelevant - Why?

Enter a comment

Se os filhos NÃO redefinirem o comportamento concreto herdado

Figure F.5: Raw Discussions on Large Class Hierarchy Breadth (Case B)


Large class size

What is it?

Large class size occurs when a class is **too lengthy** in terms of source code implemented

What metrics suggest a large class size?

Lines of Code (LOC): number of lines of code in the class
Number of Attributes (NOA): number of attributes declared in the class
Number of Methods (NOM): number of methods declared in the class

Discussion 1: Is large class size relevant when you are evolving features?

Relevant - Why?

Enter a comment

Dificilmente ocorre sozinho

Monitor 'pequeno'

Irrelevant - Why?

Enter a comment

Se tiver métodos coesos

Se não for complexa

Se o editor conseguir colapsar grandes blocos

Dificilmente ocorre sozinho

classe utilitária

Figure F.6: Raw Discussions on Large Class Size (Case B)