Pontifícia Universidade Católica
do Rio de Janeiro

Anderson José Silva de Oliveira

Unveiling Design Problems Identification: Combining Multiple Symptoms

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. Alessandro Fabricio Garcia
Co-advisor: Profa. Juliana Alves Pereira

Rio de Janeiro
October 2023

Pontifícia Universidade Católica
do Rio de Janeiro

Anderson José Silva de Oliveira

Unveiling Design Problems Identification: Combining Multiple Symptoms

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 undersigned Examination Committee.

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

Profa. Juliana Alves Pereira
Co-advisor
Departamento de Informática – PUC-Rio

Prof.ª Greis Francy Mireya Silva Calpa
Departamento de Informática – PUC-Rio

Prof. Jose Alberto Rodrigues Pereira Sardinha
Departamento de Informática – PUC-Rio

Prof Baldoino Fonseca dos Santos Neto
Universidade Federal de Alagoas – UFAL

Prof. Rafael Maiani de Mello
Universidade Federal do Rio de Janeiro – UFRJ

Rio de Janeiro, October 3rd, 2023

Anderson José Silva de Oliveira

The author received his Bachelor's degree in Computer Science from the Federal University of Alagoas (UFAL), Brazil, in 2016. During his undergraduate degree, he did an exchange program (2014) at the Eötvös Loránd University (ELTE), Hungary. He received his Master's degree in Computer Science from the Pontifical Catholic University of Rio de Janeiro (PUC-Rio) in 2019. His main research interests are Design Problems, Software Architecture, Code Smells, Software Maintenance, Software Evolution, and Non-Functional Requirements.

Bibliographic data

Oliveira, Anderson

Unveiling Design Problems Identification: Combining Multiple Symptoms / Anderson José Silva de Oliveira; advisor: Alessandro Fabricio Garcia; co-advisor: Juliana Alves Pereira. – Rio de janeiro: PUC-Rio, Departamento de Informática, 2023.

v., 192 f: il. color. ; 30 cm

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

Inclui bibliografia

1. Design Problems; – Teses. 2. Code smells; – Teses. 3. Non-Functional Requirements; – Teses. 4. Refactoring. – Teses. 5. Problemas de Projeto;. 6. Anomalias de Código;. 7. Requisitos Não-Funcionais;. 8. Refatoração.. I. Garcia, Alessandro. II. Alves Pereira, Juliana. III. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. IV. Título.

CDD: 004

Acknowledgments

I would like to begin by expressing my deep gratitude and appreciation to my parents, who made everything I have achieved possible. I am profoundly thankful to my mother, Ioneide Oliveira, and my father, Antonio Oliveira, who have always supported my decisions. Both are my greatest inspirations and provide me with the motivation I need. I also want to thank my sister, Leilane Oliveira, who has always offered advice and support when needed.

I am deeply grateful to my advisor, Alessandro Garcia, who not only guided me but also became a friend. I appreciate all the advice and suggestions he provided. I am certain that during this period of work and friendship, I have grown not only as a researcher but also as a person. I thank my co-advisor, Juliana Alves Pereira, who supported and guided me throughout my thesis. Her expertise helped enrich the quality of this thesis.

I would like to thank my girlfriend, Mariana Borges, for all her support during this period. She has always been extremely important for me to stay motivated through all the advice and affection that she always gave to me. I also thank her family, Ideraldo José and Maria Aparecida, who have become my own family.

I want to express my gratitude to Leonardo Sousa, Diego Cedrim, and Willian Oizumi, who has been with me since my MSc and have become great friends. They had always provided valuable advice and suggestions, some of which I admit I didn't want to hear but needed to. They all helped me both technically and personally. These are people I will carry with me throughout my life. I would like to thank João Lucas, Caio Barbosa, and Daniel Coutinho, who were always willing to help with research, often pointing out improvements that I couldn't see at times. I am thankful to all the current and former members of the Opus research group who were essential for the development of this thesis. Alexander Chávez, Ana Carla Bibiano, Anderson Uchôa, Daniel Tenório, Eduardo Fernandes, Isabella Ferreira, Johny Arriel, Paulo Libório, Roberto Oliveira, and Vinicius Soares.

I extend my thanks to the thesis committee members: Alberto Sardinha, Baldoino Fonseca, Greis Calpa, Rafael Maiani, Alberto Raposo, and Wesley K.G. Assunção. I appreciate their efforts and dedication in providing valuable suggestions and constructive criticisms that were essential for the success of this thesis.

I am grateful to all the professors at PUC-Rio, who played a crucial role in the development of this research. I thank all the staff at the Department of Informatics, who were always helpful and are also part of this thesis. I also

want to thank my professors at UFAL, who paved the way for me to reach this point.

Finally, I would like to express my gratitude for the funding provided by PUC-Rio, and CNPq (140185/2020-8). Through their support, this entire research endeavor was made possible. This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001.

Abstract

Oliveira, Anderson; Garcia, Alessandro (Advisor); Alves Pereira, Juliana (Co-Advisor). **Unveiling Design Problems Identification: Combining Multiple Symptoms**. Rio de Janeiro, 2023. 192p. Tese de doutorado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Software design results from stakeholder decisions made through software development. Some of these decisions may lead to design problems, negatively impacting non-functional requirements (NFRs). Even though identifying design problems is crucial, this is a complex task, especially when the source code is the only artifact available. Along this task, developers may have to reason about multiple symptoms (*e.g.*, code smells and non-conformities with NFRs) to identify even a single design problem. In fact, previous studies suggest that relying on a single symptom may be inadequate for the design problem identification. Thus, in this thesis, we investigate the role that the use of multiple symptoms may have on the identification of design problems. In our first study, we focused on investigating the use of well-known code smells (called here maintainability smells) to support this task. Our results indicated that developers could benefit from this type of symptom when smell occurrences affect the same program location and form a pattern; *i.e.*, a set of co-occurring maintainability smells may better indicate the presence of a design problem. Nevertheless, we also reveal the limitations of relying solely on this type of symptom, highlighting the need for additional context. This leads us to the second study, where we investigate an additional type of symptom, robustness smells, and its combined use with maintainability smells. Our results indicated that the use of both types of smells can help developers in the identification of design problems mainly related to bad modularization of the system (*e.g.* excess of responsibilities assigned to the same component). Through these two studies, we observed the need to understand the perspectives and strategies of developers toward the NFRs of the system. In doing so, we can potentially understand who are the developers better able to prevent, discuss and identify design problems. That led us to our third study, where we investigated how developers discuss and address NFRs in their systems, uncovering common strategies toward these requirements. These results led us to a more comprehensive understanding of how developers can combine different symptoms and how they perceive their significance within their systems.

Keywords

Design Problems; Code Smells; Non-Functional Requirements; Refactoring.

Resumo

Oliveira, Anderson; Garcia, Alessandro; Alves Pereira, Juliana. **Compreendendo a Identificação de Problemas de Projeto: Combinando Multiplos Sintomas**. Rio de Janeiro, 2023. 192p. Tese de Doutorado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

O projeto de software resulta das decisões ao longo do seu desenvolvimento. Algumas dessas decisões podem levar a problemas de projeto, afetando negativamente os requisitos não funcionais (RNFs). Embora seja crucial identificar esses problemas, essa é uma tarefa complexa, especialmente quando o código-fonte é o único artefato disponível. Nessa tarefa, os desenvolvedores podem ter que considerar vários sintomas (por exemplo, anomalias de código) para identificar até mesmo um único problema de projeto. Estudos anteriores sugerem que usar um único sintoma pode ser inadequado para identificar tais problemas. Portanto, nesta tese, investigamos como múltiplos sintomas podem ser usados nessa identificação. Em nosso primeiro estudo, nos concentramos em investigar o uso de anomalias de código bem conhecidos (anomalias de manutenabilidade). Nós identificamos que os desenvolvedores podem se beneficiar desse tipo de sintoma quando as ocorrências das anomalias afetam a mesma localização do programa e formam um padrão, podendo indicar melhor a presença de um problema de projeto. No entanto, também revelamos as limitações ao depender exclusivamente desse tipo de sintoma, destacando a necessidade de contexto adicional. Isso nos levou ao segundo estudo, onde investigamos um tipo adicional de sintoma, anomalias de robustez, e seu uso combinado com anumalias de manutenabilidade. Nós identificamos que ambos os tipos de anomalia de código podem ajudar os desenvolvedores na identificação de problemas de projeto principalmente relacionados à má modularização do sistema. Através desses dois estudos, observamos a necessidade de compreender as perspectivas e estratégias dos desenvolvedores em relação aos RNFs do sistema. Ao fazê-lo, podemos potencialmente entender quem são os desenvolvedores mais capazes de prevenir, discutir e identificar problemas de projeto. Isso nos levou ao terceiro estudo, onde investigamos como os desenvolvedores discutem e abordam RNFs em seus sistemas, revelando estratégias comuns em relação a esses requisitos. Esses resultados nos proporcionaram uma compreensão mais abrangente de como os desenvolvedores podem combinar diferentes sintomas e como percebem sua importância dentro de seus sistemas.

Palavras-chave

Problemas de Projeto; Anomalias de Código; Requisitos Não-Funcionais; Refatoração.

Table of contents

List of figures

List of tables

1
Introduction

Software design results from design decisions stakeholders take during the software development process (Tang *et al.* 2010). These decisions can emerge from multiple sources, such as official meetings, daily discussions, mailing lists between the developers and stakeholders, and developers' collaborative repositories (Baker *et al.* 2011). Brunet *et al.* show that a considerable part of the discussions related to the system design are addressed by developers on commits, issues, and pull requests on open-source systems (OSSs) (Brunet *et al.* 2014). In case design decisions are not taken into account accordingly, they can negatively affect non-functional requirements (NFRs), such as maintainability and robustness (Perry and Wolf 1992, Xiao, Cai and Kazman 2014). In that case, these decisions result in design problems.

A critical set of design problems are those affecting how the system is organized in terms of components and how these components communicate with each other (Lippert 2006). The presence of these design problems usually lead to significant maintenance effort (Garcia *et al.* 2009b, Schach *et al.* 2002, Yamashita and Moonen 2012). An example of a design problem is the so-called *Scattered Concern* (Garcia *et al.* 2013), which consists of multiple components responsible for realizing the same high-level concern. This responsibility should be modularly implemented into a single component, following the *Single Responsibility Principle* (Martin and Martin 2006).

When these design problems are neglected, the complexity and difficulty of maintaining the system tend to increase over time (Curtis, Sappid and Szynkarski 2012). This leads to longer maintenance cycles, increased effort required for bug fixes or feature additions, and a higher probability of introducing new design problems during maintenance activities (Oliveira, Valente and Terra 2016). These factors contribute to an overall increase in the maintenance burden, progressively creating higher maintenance costs, resulting in software discontinuation (MacCormack, Rusnak and Baldwin 2006, Godfrey and Lee 2000, Schach *et al.* 2002). To mitigate these negative effects, developers must identify and remove these design problems as early as possible for subsequent removal.

However, the identification of design problems is not an easy task.

Several reasons make this task challenging for developers. One significant challenge is the lack of reliable and up-to-date design documentation. Design documentation is a valuable resource for understanding the intended design and identifying potential design problems more accurately. However, proper documentation in software projects is often nonexistent or outdated (Kaminski 2007). A complementary challenge for the identification of design problems is the lack of modularity in the affected software system. Various types of design problems usually affect multiple elements of a component and/or multiple inner modules of a component (*e.g.*, classes or packages), thus requiring major effort from developers to find and inspect all these elements together. This task becomes even more time-consuming for the developers without proper knowledge of the system design and without documentation support. Hence, they are forced to rely almost only on the source code.

Relying on the source code is a practical approach because design problems often manifest in the software through observable *symptoms* in the source code (Sousa *et al.* 2018). Examples of design problem symptoms include code smells (Fowler 1999) and non-conformity of NFRs. Developers combine these symptoms to identify the design problems more confidently (Sousa *et al.* 2018, Oizumi *et al.* 2016, Sharma *et al.* 2020). One example of code smell is the Feature Envy, which indicates that a method is more interested in data from another class than the data from its own class (Lanza and Marinescu 2006). This code smell is related to non-conformity with the maintainability NFR and can indicate a *Scattered Concern* (Sousa *et al.* 2020).

There are certain scenarios where multiple maintainability smells appear together in a certain code location, better reinforcing the presence of a design problem in that location. A specific combination of smell types may occur often across software systems. Given their recurring nature, they are called *maintainability smell patterns* (Sousa *et al.* 2020). These patterns may potentially better indicate the occurrence of a design problem and which refactoring operations should be performed to remove this problem (Sousa *et al.* 2020). Refactoring is a process utilized to fix or improve the degraded code without changing the program intended behavior (Fowler 1999).

Multiple studies focused on using maintainability smells as the driving symptom for identifying design problems (Oliveira *et al.* 2019, Oizumi *et al.* 2016, Sousa *et al.* 2018). However, no evidence exists on whether developers would benefit from patterns of maintainability smells in practice. In addition, since design problems can be present in multiple system components, the developer may need more information regarding the context of the class, component, or system analyzed. By better understanding the context, developers can bet-

ter anticipate the potential impact of design problems. Therefore, analyzing multiple symptoms is often necessary to ensure more accurate identification and complete removal of design problems.

Another possible symptom of design problems is represented by robustness smells (Kechagia and Spinellis 2014, Oliveira *et al.* 2016), which negatively impact the error handling part. Robustness is the ability of a program to continue its execution properly in the presence of an error (Lee *et al.* 1990). Thus, robustness code smells occur in the part of the program explicitly using error handling constructs (*e.g.*, catch clauses in Java)

One example of robustness smell is the *empty catch block*, which happens when a method's *catch block* is neglected and, thus, its inner code is left empty. Current studies of design problem identification do not explicitly consider the exceptional part of the code. While the normal code refers to a program's regular or expected behavior, the exceptional code refers to the code that handles erroneous conditions that can occur during the program execution. The exceptional code is placed within the catch block. Therefore, while maintainability smells often manifest in normal code, robustness smells tend to emerge in exceptional code. Therefore, developers can identify various design problems by analyzing both types of code smells together instead of solely focusing on maintainability smells. Addressing maintainability and robustness smells is essential to ensure the system meets its defined NFRs effectively. Neglecting these smells can result in non-conformities with their respective NFRs. This includes the risk of high maintenance costs and inadequate responses to unexpected behaviors.

Identifying non-conformities with NFRs can be challenging for developers, primarily when they lack the necessary knowledge about these requirements (Lausen 2002). NFRs can often be implicit in the system, which makes it necessary to infer them from multiple sources, such as discussions on issues and pull requests. Moreover, NFRs can be interdependent, meaning that changes to one NFR can impact others, making it difficult for inexperienced developers to fully comprehend these dependencies (Cacho et al. 2014). To address these challenges effectively, it is crucial to identify how developers deal with NFRs and understand their best strategies to solve NFR-related problems. Developers concerned with NFRs are better positioned to identify potential design problems within the system, as these problems often manifest as non-conformities with NFRs (Martin and Martin 2006). In this context, analyzing how experienced developers combine different symptoms to identify design problems can provide valuable insights to less experienced developers. Less experienced developers could benefit from tools that, besides detecting

multiple symptoms, also combine them for revealing design problems (Sousa *et al.* 2018).

In the following section, we present a hypothetical motivating example about *(i)* how developers can benefit from the identification of multiple symptoms to identify design problems correctly; and *(ii)* the importance of naming experienced developers to identify and remove such problems.

1.1 Motivating Example

Let us consider a software application called *ManagementSystem*, which manages book loans and sales. After a few years of development, developers observed that significant effort was required to maintain and evolve the system. Although the developers were aware of the presence of maintainability smells affecting multiple classes, they found it difficult to identify them, and more importantly, they were unsure which specific elements (*e.g.*, classes or methods) and smells to prioritize and address with their efforts.

Figure 1.1 illustrates a UML-based representation of a certain part of the system. The class `Book` in this system has multiple issues. Some of these issues are maintainability code smells, which are presented by smell rectangles in red in the figure. Firstly, the `sell()` method updates data from the `Client` and `Contract` classes at the same time, violating the *Single Responsibility Principle* (SRP) (Martin and Martin 2006). This method also accesses data from these two classes, configuring the occurrence of a *Feature Envy (FE)* smell (Lanza and Marinescu 2006).

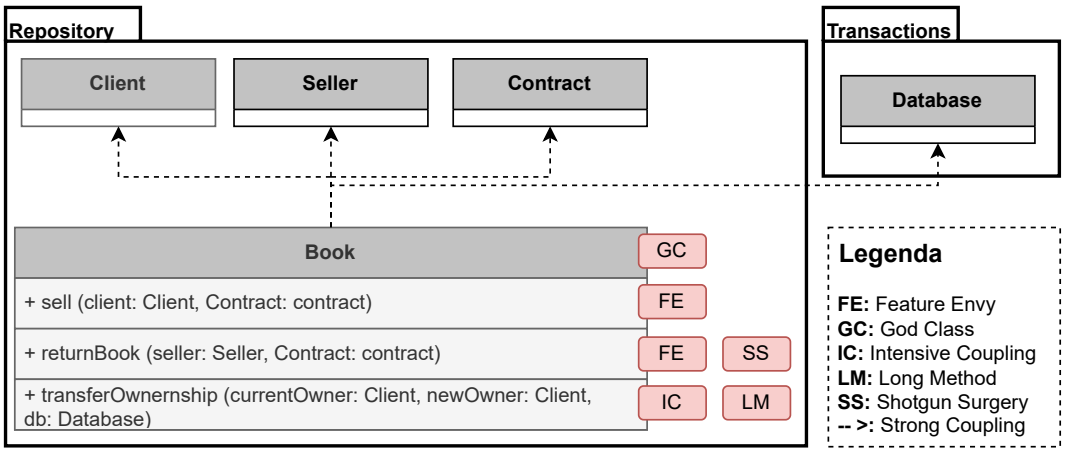

Figure 1.1: Maintainability Smells Affecting the ManagementSystem Project

The `returnBook()` method, besides violating the SRP, is also affected by *Shotgun Surgery (SS)* (Fowler 1999) and a *Feature Envy (FE)* maintainability smells. The *SS* smell indicates that if any of the logic changes within the

method, multiple changes have to be carried out in other classes: `Seller` and `Contract` in this particular case. Meanwhile, the *FE* smell indicates that the method is more interested in data from other classes than inner data from the class where the method is declared. The method `transferOwnership()` contains an *Intensive Coupling (IC)* (Buschmann *et al.* 1996) as it directly manipulates the `Client` object. In addition, this method uses the `Database` class, which is part of another component of the system, the `Transactions` component.

Finally, the `Book` class is a *God Class (GC)* (Fowler 1999) due to its high complexity and multiple unrelated responsibilities that are performed. Combining all these maintainability smells, they are part of a pattern that indicates the presence of the *Concern Overload* design problem (Sousa *et al.* 2020). In this particular situation, a less experienced developer or someone less familiar with system design might not have full confidence in identifying this design problem. They may lack sufficient context to determine if this problem genuinely exists within the system. Additional information, such as the purpose of the class and methods, is necessary to make a proper identification.

Furthermore, the developer could have the necessary context by performing a more in-depth analysis of the methods within the system. In this scenario, robustness smells also affected key methods in addition to the aforementioned maintainability smells. Figure 1.2 shows the exception handling part of the method `sell()`. Any exceptions that might occur when setting the *seller* attribute are caught; however, only a message is logged. In that case, the method is affected by two robustness smells, the *Generic Catch* and *Empty Catch Block*. This happens because the class has multiple concerns, making it difficult for the developer to handle all the exceptions that these concerns require. Therefore, they use a generic catch and leave the catch block empty. These smells can hide errors and make identifying problems in the code difficult.

```
public void sell(Seller seller) {
  try{
    this.seller = seller;
    ...
  }catch (Exception e) {
    logger.log("Error setting seller");
  }
}
```

Figure 1.2: Robustness Smells *Generic Catch* and *Empty Catch Block* Affecting the `sell()` method

By observing all the smells in the `sell()` method (*i.e. Feature Envy*,

Generic Catch, and *Empty Catch Block*), and that the class is a *God Class*, the developer can have more confidence in pinpointing the presence of the *Concern Overload* design problem. This is a scenario where, by pointing out to developers even the simplest code symptom, such as a *Catch Generic Exception*, combined with (patterns of) maintainability smells, inexperienced developers could have more confidence in identifying a design problem.

Developers will likely struggle to identify and effectively remove design problems without proper knowledge of the system's structure. They may not completely understand the context in which the class is being used and its dependencies with other classes and components. Hence, they can inadvertently introduce new design problems, affecting other related components. Thus, assisting developers with recommendations regarding key code elements and their corresponding maintainability and robustness smells will enable them to make informed decisions. In addition, it is still important that they gather more knowledge of the system design, which can be obtained with the help of experienced developers.

1.2 Problem Statement

The identification of design problems can be a difficult task. Developers need to understand well the system architecture, and the lack of proper documentation makes the task even harder (see Section 1.1). Therefore, developers use code smells as a driving source for identifying design problems (Yamashita and Moonen 2013, Sousa *et al.* 2018). Thus, previous empirical studies investigated whether *maintainability code smells* are related to design problems (Fontana *et al.* 2019, Martins 2020, Oizumi *et al.* 2016, Oizumi *et al.* 2015, Moha, Gueheneuc and Leduc 2006, Moha *et al.* 2010, Macia *et al.* 2012b, Macia 2013, Vidal *et al.* 2016).

When specific code smells appear together, they form a *pattern*. These patterns may be used in the identification of certain design problems (Sousa *et al.* 2020). For instance, when the maintainability smells *God Class*, *Intensive Coupling*, *Long Method*, *Feature Envy* appear together, they are likely to indicate a *Concern Overload* design problem (see Section 1.1). This happens due to the nature of these smells, which combined indicate that the class and/or method is fulfilling too many responsibilities that should be better modularized. Then, a developer becoming aware of this smell pattern could now promptly identify the occurrence of the *Concern Overload* design problem. In addition, the information about a smell pattern occurrence can be used to select proper code refactoring actions (Fowler 1999). These refactorings can be

used to reduce or remove these design problems. For instance, *Feature Envy* smell is commonly associated with the refactorings *Move Method*, *Move Field*, and *Extract Field* (Fowler 1999) (see Section 2.4) However, we still do not have evidence of whether developers could benefit by using smell patterns in practice. Given this context, we define our first research problem.

> **Research Problem 1.** The extent to which developers can benefit from the use of smell patterns in the identification and removal of design problems is unknown.

Maintainability smells are more popularly known and empirically studied, but they are only one of the multiple symptoms of design problems (Sousa *et al.* 2017). Thus, the sole use of maintainability smell patterns may not be enough to identify certain design problems (see Section 1.1). In this research, we focus on investigating robustness smells and how these two types of smells can complement each other to identify design problems.

Previous studies do not explore the combination of maintainability and robustness smells for the purpose of identifying design problems effectively (Oizumi *et al.* 2016, Sousa *et al.* 2018, Sousa *et al.* 2020). Thus, our study aims to understand how changes in the exceptional code combined with maintainability smells can lead to or point out design problems. Hence, we defined our second research problem.

> **Research Problem 2.** The extent to which robustness smells can be combined with maintainability smells for the identification of design problems is unknown.

In addition to robustness and maintainability smells, the non-conformity of NFRs in a system is another symptom of design problems (Sousa *et al.* 2018). A non-conformity emerges when a NFR, defined in the early stages of the software system, does not meet the requirements expected (*e.g.*, specific thresholds to define acceptable performance). Developers could use the requirements documentation to ensure the system meets the NFRs defined. However, NFRs' documentation is often neglected (Bhowmik *et al.* 2019, Chung *et al.* 2012, Scacchi 2009). In this scenario, it becomes important to understand developers' strategies and practices to manage NFRs in their systems.

Even though other studies explored how developers perceive NFRs in their systems (Ameller *et al.* 2012, Glinz 2007), they do not consider their best

practices for handling NFRs and how they discuss such requirements. Therefore, this thesis aims to investigate developers' best practices and strategies when dealing with NFRs. To achieve that, we need also to understand how the developers discuss the NFRs in their systems. In addition, developers' discussions can help us understand whether developers are truly concerned with NFRs in a software system. Hence, we defined our third research problem.

> **Research Problem 3.** How developers discuss and address Non-Functional Requirements (NFRs) throughout the project's development stages is unknown.

By exploring these three research problems, we aim to better support the developers in identifying design problems through reasoning about multiple and diverse symptoms. Thus, one can prevent new design problems in the system and help further investigate such problems. Therefore, our main goal is to understand how developers can use and combine multiple symptoms to identify and address design problems.

> **General Problem.** How developers use multiple and diverse symptoms for identifying design problems is still unknown.

A widely studied type of symptom of design problems is the maintainability-related code smells (Sousa *et al.* 2020, Oizumi *et al.* 2020, Oliveira *et al.* 2019). While there are state-of-the-art tools to detect maintainability smells (Oizumi *et al.* 2015) and to partially assist the task of identifying design problems, there is a need to investigate other types of design problem symptoms. In this context, robustness smells can give more confidence to developers and reduce their efforts along this task. In addition, understanding how developers deal with NFRs, including robustness and maintainability, may help one to prevent system design problems. However, we still do not know how the developers can use multiple symptoms and how they discuss and address NFRs in their systems. Hence, we defined our general problem above.

1.3 Goal and Research Questions

As aforementioned, identifying design problems is not trivial (Section 1.1). To support the developer in the identification of design problems effectively, it is crucial to understand how they may use multiple and diverse

symptoms along this task. In particular, we need to characterize who are the developers more likely to deal with such symptoms. Understanding the expertise of developers in solving design problems enables effective task management within a development team. By assigning the most experienced developers to address specific design problems, resource allocation can be optimized, leading to increased productivity and efficient problem resolution. In addition, by analyzing these experts individually, we can provide insights into potential process improvement, where additional tooling and automation can be proposed to assist less experienced developers. Given this scenario, the goal of this thesis is stated as follows.

> **Goal.** Support developers on the identification of design problems by combining multiple and diverse symptoms

In Figure 1.3, we present an overview of the goal, contributions, and research questions of this thesis. The arrows represent the relationships between the research questions, their respective contributions, and how they relate to our main goal. For each research question, we also present the chapter where the study is detailed.

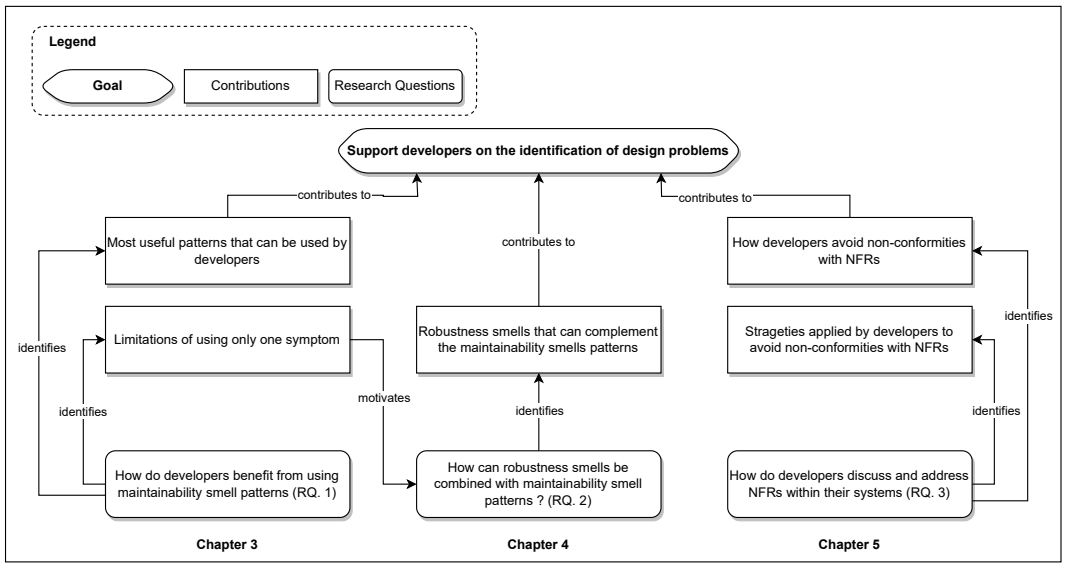

Figure 1.3: Overview of the Contributions of this Thesis

To accomplish our goal, we defined three research questions aligned with our identified research problems (see Section 1.2). For each research question, we defined a specific goal to be met. Our first goal is to *understand how developers can use the maintainability smell patterns to identify design problems in their systems*. For that purpose, we conducted a quasi-experiment (Shadish, Cook and Campbell 2001) to investigate developers' ability to identify design

problems in their software systems by using different sets of scenarios with smell patterns. We defined the first research question (*RQ. 1*) to address this goal.

> **RQ. 1** How do developers benefit from using maintainability smell patterns to identify design problems in practice?

To answer *RQ. 1*, we defined two steps. As the first step, we aimed to understand how experienced developers used maintainability smell patterns. Thus, we asked developers to identify design problems in their software systems. For that purpose, we developed a tool that presented the developers with the patterns of maintainability smells and their respective refactorings. Specifically, the developers were asked to identify design problems in three different scenarios.

In the first scenario, they used a "single-smell pattern". For instance, the design problem *Incomplete Abstraction* can be indicated by a *Lazy Class* maintainability smell. In the second scenario, they used "multiple-smells patterns". In this case, for instance, the design problem *Cyclic Dependency* can be indicated by two maintainability smells: *Intensive Coupling* and *Shotgun Surgery*. In the last scenario, they used a combination of other maintainability smells (*Others*).

After collecting the data from this task, both quantitative and qualitative analyses were conducted. We applied the statistical test One-Way ANOVA (Shaw and Mitchell-Olds 1993) and Chi-Square Test (McHugh 2013) for our quantitative analysis. The One-Way ANOVA was used for comparing the use of different scenarios (multiple smell patterns, single-smell pattern, and others) by the same group of participants. We applied the Chi-Square to compare the different groups of responses regarding the proportion of smell patterns that indicated design problems and smell patterns that indicated other types of problems. In the qualitative analysis, we systematically analyzed the descriptive responses given by the developers when analyzing each scenario with a set of smell patterns.

As the second step, we observed how developers employed the suggested refactorings provided by the tool support. We decided to analyze this since the next step after identifying design problems is to remove them, which can be accomplished through the use of refactorings (Fowler 1999). Beyond showing developer information about metrics and maintainability smells (see Chapter 2), we show refactoring recommendations for each maintainability smell (see Section 2.5.2). Finally, we asked the developers their perceptions on how

the recommended refactorings could effectively remove the identified design problems. The purpose was to understand the scenarios where developers *accepted*, *partially accepted*, or *rejected* the suggested refactorings.

Furthermore, we investigate other factors the developers could use to complement the maintainability smell patterns. We hypothesize that the robustness smells can be combined with these patterns to identify design problems (Section 1.1). This way, we defined our second goal: *understand how robustness smells can be combined with maintainability smells as complementary symptoms of design problems*. With this goal in mind, we defined our second research question (*RQ. 2*).

> **RQ. 2** How can robustness smells be combined with maintainability smell patterns to identify design problems?

To answer *RQ. 2*, we analyzed over 160k methods from 10 OSSs. We start the analysis by collecting maintainability smells, robustness smells, and robustness changes (*e.g.*, changes made within the catch block) from commits between selected pairs of releases within those systems. To answer this research question, we defined three steps. First, we analyzed the correlation between robustness changes and maintainability smells, using Fisher's exact test (Fisher 1922). To conduct this test, we grouped the methods based on two factors: *(i)* the robustness changes they underwent between two releases and *(ii)* the presence of maintainability smells in these methods (see Section 4.4).

In the second step, we investigated whether the robustness changes performed on methods resulted in the degradation of the classes with methods that underwent such changes. This degradation is measured as the number of maintainability smells present in the method (Sousa *et al.* 2018). In the third step, we investigated how poor robustness changes can be used as symptoms of design problems. These poor robustness changes are signaled by the presence of robustness smells, such as the *empty catch block*.

Once we understand how developers can use multiple symptoms to identify design problems, we now want to understand how they deal with NFRs in their systems. Thus, we defined our third goal: *document best practices and strategies of developers who discuss and address NFRs within their systems*. With this goal in mind, we defined our third research question (*RQ. 3*).

> **RQ. 3** How do developers discuss and address NFRs within their systems?

To answer *RQ. 3*, we first explored Pull Requests (PRs) discussions since it is a common mechanism used by developers to discuss new features and improvements in their systems (Soares *et al.* 2015, Gousios *et al.* 2014, Gousios *et al.* 2015, Jiang *et al.* 2021). To perform this analysis, we manually classified 1,533 PRs from 3 OSSs within the Spring Ecosystem[1]. We selected this ecosystem due to its diversity of features and services (Cosmina *et al.* 2017) and its active discussions through PRs. To select the NFRs that we would focus on, we performed an analysis to observe the most prominent ones in the systems analyzed. Therefore, we focused on four NFRs *maintainability*, *robustness*, *performance*, and *security*.

To understand the discussions related to these four NFRs, we followed a well-structured procedure to perform the manual classification of PRs. We identified in each PR discussion: *(i)* the presence of the NFR type addressed, *(ii)* the location in the PR where the discussions about NFR are triggered, *(iii)* keywords mentioned in the discussion, and *(iv)* the main message addressing the NFR. That procedure allowed us to characterize the PRs and identify developers involved in the discussions. To define the developers we would analyze, we identified 63 developers who were more engaged in PR discussions. We investigated the characteristics of each one of these developers, searching for their common activities. To complement our analysis, we surveyed 44 developers regarding their perceptions and actions towards NFRs in their systems.

To summarize, by addressing the first two research questions, we aim to gain insights into how developers use maintainability smell patterns combined with robustness smells to identify design problems in practice. Furthermore, in the third research question, we explored how developers discuss and address NFRs in their systems, including robustness and maintainability. By documenting their experience and skills regarding the NFRs, we can prevent future design problems and help newcomers on identifying and dealing with them. In addition, by understanding how developers use these multiple symptoms, we can offer insights into the development of tools for the identification of design problems. Thus, we enable advancements in state-of-the-art automation support in identifying design problems by considering code smells and NFRs.

1.4 Main Findings

With this thesis, we aim to support developers in identifying design problems by combining multiple symptoms (*e.g.*, maintainability and robustness

[1]Spring | Home, `https://spring.io/`, Accessed on 10/21/2022

smells, and non-conformity with NFRs). Next, we highlight the key findings we obtained for each research question defined in Section 1.3.

RQ. 1: How do developers benefit from using maintainability smell patterns to identify design problems in practice? We verified whether developers could benefit from using maintainability smell patterns in practice. We observed that developers generally agree that maintainability smell patterns can indicate design problems. This agreement is influenced by various factors, such as the development platform (*e.g.*, Android) and the level of effort required to refactor and remove the problem. The developers also indicated that the information provided regarding the maintainability smell patterns may be insufficient for more complex problems, such as design problems that affect multiple modules. This also led them to partially accept the refactorings recommended/suggested for the design problem removal. Therefore, these results indicated that more contextual information is needed for more efficient identification and removal of design problems (see Section 3.4).

RQ. 2: How can robustness smells be combined with maintainability smell patterns to identify design problems? We verified whether robustness smells could be combined with the maintainability smell patterns to support the identification of design problems. We first observed that maintainability smells commonly co-occur with robustness changes, even when these changes are small. Moreover, we also observed cases when the maintainability smells were introduced and how this smell introduction was related to the robustness change performed in the method.

When analyzing the impact of robustness changes on the maintainability smells, we analyzed the density of maintainability smells (*i.e.*, the number of smells in the method(s) affected by the change). This can information indicate the structural degradation level of a method. We found that robustness changes can have an unsatisfactory effect on classes with methods affected by those changes, which leads to a higher density of maintainability smells. Developers often rely on this increased density of smells to identify classes with potential design problems. The introduction or further deterioration of the smelly code was primarily related to maintainability smells that indicate design problems with the system's modularity. These findings suggest a high correlation between robustness changes and the presence multiple maintainability smells, emphasizing the importance of considering the interplay between them in effectively identifying and addressing design problems.

Finally, we analyzed the co-occurrence of robustness smells and maintain-

ability smell patterns. We aimed to identify which specific robustness smells could complement these patterns in identifying design problems. In particular, we observed that the robustness smells *catch generic exception* and *empty catch block* co-occurred with maintainability smells more often. By manually analyzing these cases, we observed that generic and empty catches could indicate the presence of maintainability smell patterns (see Section 4.5).

RQ. 3: How do developers discuss and address NFRs within their systems? To address this question, we explored how developers discuss and address design problems in their systems. First, we investigated the dataset of 1,533 PR discussions manually labeled by seven specialists. This investigation showed that discussions about NFRs are typically triggered on the PR title or description (77% of the cases). That brings us to the fact that developers are aware and concerned about these NFRs before opening the PR. This can be important, especially to identify who are these developers opening the PRs, since they might be the ones with more knowledge on NFRs.

To better understand the developers engaged in discussions about NFRs, we analyzed 63 members of the repository. For this purpose, we investigated their characteristics, network, and participation in the discussions. We observed that these developers were central contributors to the systems, highly participating in tasks such as commits, reviews, and refactorings. Furthermore, we observed that these developers played central roles related to NFRs in their companies. For instance, a developer who discussed more security-related problems was the *Senior Security Engineer* from his company. This suggests that discussions regarding NFRs have the potential to bring the attention of experienced developers in the OSS community.

To complement our analysis of NFR discussions, we ran a survey with 44 developers. Through this survey, we identified the reasons behind developers' participation in NFR discussions. The majority of developers emphasized the impact that NFRs have on software quality (*e.g.* performance improvements). That was also related to the mitigation of risks that could emerge when an NFR was neglected, such as increasing maintenance costs over time. According to the developers, the discussions usually occur in multiple phases of the development process, with a special focus on the design and test phases.

Regarding how the developers addressed the NFRs, they mentioned that collaboration with other team members was one of the more important aspects of this task. They also highlighted the use of automated tools combined with practices, such as continuous integration, to identify and address the NFRs in the system. Among the challenges when dealing with NFRs, the main ones

were the lack of documentation regarding NFRs and how to deal with trade-offs between the NFRs (*e.g.* decrease the performance to increase security).

When we look at the specialists that handle the NFRs in the systems, the developers mentioned that, usually, there was no such role in the company. Instead, the NFR was usually handled by engineers and architects with expertise in specific NFR domains vs. types of NFR, such as security and performance. However, we also observed a case with a team specialized in the systems' NFRs. However, this team was assigned to deal with NFRs from different projects within the company.

Through these research questions, we observed some strategies developers could adopt to address NFRs. The NFR should be treated with as much priority as other requirements, with a focus on the discussion with other team members. The adequacy of NFR with the system objectives can be accomplished using technologies that have already received market approval, which can be improved with practices such as continuous integration. In addition, developers highlighted the importance of updating NFR documentation with their main concerns and best practices and making it available for any team member.

Our findings gave us insights into how developers use maintainability smell patterns in real-world scenarios and the limitations of using only this smell to identify design problems. Additionally, we observed how robustness smells could be combined with maintainability smell patterns to assist developers better. Finally, we investigated how developers discuss and address NFRs, looking for their best strategies to avoid design problems.

1.5 Collaborations and Publications

During my Ph.D. research, I collaborated with multiple researchers on studies that are directly or indirectly related to the thesis. In addition to the collaborations with colleagues from *Opus Research Group*, part of the Software Engineering Laboratory (LES), I also worked with researchers from *Carnegie Mellon University*, *Johannes Kepler University Linz* (JKU), and *Federal University of Alagoas* (UFAL). Table 1.1 presents all the publications I (co-)authored during my PhD. The first row refers to a paper under submission and is presented as the content of Chapter 5. The second row refers to the study presented in Chapter 4, published in the 2023 edition of the *International Conference on Mining Software Repositories (MSR)*. The third row refers to the study reported in Chapter 3 and published in the 2022 edition of the *Brazilian Symposium on Software Engineering (SBES)*. The remaining rows

Table 1.1: Direct and Indirect Contributions of this Thesis

Paper	Qualis
Oliveira *et al.*: Understanding Discussions on Non-Functional Requirements: The Case of the Spring Ecosystem [**Submitted to a leading conference in Software Engineering**]	A1
Oliveira *et al.*: Don't Forget the Exception! Considering Robustness Changes to Identify Design Problems - 20th International Conference on Mining Software Repositories (MSR '23)	A1
Oliveira *et al.*: Smell Patterns as Indicators of Design Degradation: Do Developers Agree? - 36th Brazilian Symposium on Software Engineering (SBES '22)	A3
Uchôa *et al.*: Predicting Design Impactful Changes in Modern Code Review: A Large-Scale Empirical Study - 18th International Conference on Mining Software Repositories (MSR '21)	A1
Soares *et al.*: On the Relation between Complexity, Explicitness, Effectiveness of Refactorings and Non-Functional Concerns - 34th Brazilian Symposium on Software Engineering (SBES '20)	A3
Sousa *et al.*: Characterizing and Identifying Composite Refactorings: Concepts, Heuristics, and Patterns - 17th International Conference on Mining Softwre Repositories (MSR '20)	A1
Oizumi *et al.*: Recommending Composite Refactorings for Smell Removal: Heuristics and Evaluation - 34th Brazilian Symposium on Software Engineering (SBES '20) [**Distinguished Paper Award**]	A3
Oliveira *et al.*: Atoms of Confusion: The Eyes Do Not Lie - 34th Brazilian Symposium on Software Engineering (SBES '20)	A3
Bibiano *et al.*: Look Ahead! Revealing Complete Composite Refactorings and their Smelliness Effects - 37th International Conference on Software Evolution (ICSME '21)	A2
Sousa *et al.*: When Are Smells Indicators of Architectural Refactoring Opportunities: A Study of 50 Software Projects - 28th International Conference on Program Comprehension (ICPC '20)	A3
Bibiano *et al.*: How Does Incomplete Composite Refactoring Affect Internal Quality Attributes? - 28th International Conference on Program Comprehension (ICPC '20)	A3
Oizumi *et al.*: On the Density and Diversity of Degradation Symptoms in Refactored Classes: A Multi-case Study - 30th International Symposium on Software Reliability Engineering (ISSRE '19)	A3
Eposhi *et al.*: Removal of design problems through refactorings: Are we looking at the right symptoms? - 27th International Conference on Program Comprehension (ICPC '19) - Negative Results Track	A3

are studies either directly or indirectly related to this thesis.

1.6 Thesis Outline

This chapter provides an overview of this thesis. This thesis is structured as a compilation of three papers that have been published during the course of this Ph.D. research. Each paper contributes to addressing a specific research question presented here in this introductory chapter. Chapters 3, 4, and 5 will detail the study reported in each paper, providing in-depth discussions, analyses, and results for the questions addressed in this Doctoral Research.

The chapters are organized as follows. **Chapter 2** presents the background and related work of this thesis. **Chapter 3** presents the study to answer our *RQ. 1*, where we explored how developers can use maintainability smell patterns to identify and remove design problems. **Chapter 4** presents the study to answer our *RQ. 2*, where we explored how robustness smells can be combined with maintainability smells for a more comprehensive design problem identification. **Chapter 5** presents the study to answer our *RQ. 3*, where we investigated how developers deal with NFRs in their systems. **Chapter 6** presents the conclusions we reached and the future work based on our results.

2
Background and Related Work

This section presents the background and related work of this thesis. In the next sections, we provide the terminologies used throughout the chapters. Then, we present an overview of the state of the art, where the limitations and gaps in the current literature are thoroughly described. This chapter is structured around the following sections. Section 2.1 discusses the relation between software design and NFRs. Section 2.2 discusses and lists the design problems that will be explored in this thesis. Section 2.3 presents the symptoms of design problems. Section 2.4 presents how refactorings can be used to remove design problems. Section 2.5 presents the related work.

2.1
Software Design and Non-Functional Requirements Discussions

Software design results from multiple design decisions taken during software development (Tang *et al.* 2010). These decisions include the process of defining the characteristics that satisfy non-functional requirements (NFRs) defined for the system (Bass et al. 2003, Booch *et al.* 2005, Taylor et al. 2009, Sousa *et al.* 2018). Examples of NFRs include maintainability, robustness, and security (ISO-IEC 25010 2011).

The software design process can be divided into two main stages. The first stage is the *early software design*. In this stage, the stakeholders discuss and understand the problem domain, creating a high-level system structure (Booch *et al.* 2005, Taylor et al. 2009). At this point, the system is divided into components, interfaces, and how they communicate with each other.

The second stage is called *detailed design*. This stage concerns the specific decisions regarding implfhoementing each component and its interactions (Booch *et al.* 2005, Taylor et al. 2009). The decisions taken in this stage will address the NFRs that the system should satisfy. For instance, a design that considers the exceptional part of the code will be prepared to handle certain types of inputs or unexpected events that the system may face (Robillard 2000). Therefore, a carefully planned design that considers this aspect will guarantee a more robust software system. In addition, given the NFR importance for software evolution, stakeholders should specify them in requirement

documents, ideally during the early stages of the software development.

However, the requirements documentation often focuses on the system's functional requirements, missing details on the system' NFRs (Cleland-Huang *et al.* 2007). This lack of NFR documentation can negatively impact the system maintenance and evolution (Robiolo *et al.* 2019). Hence, studies have explored how to automatically identify the NFRs in the systems based on the available artifacts. Bikhonain and Zao have performed a review of the available techniques for the identification of NFRs using natural language processing (NLP) combined with machine learning techniques (Binkhonain and Zhao 2019).

The authors identified a series of limitations on these techniques. For instance, there is a lack of training data and few shared datasets. Furthermore, there is no consensus on the definition, classification, and representation of the NFRs, which causes a diverse range of terms to define the NFRs. Such problems make creating machine learning models challenging since they can be trained and used only for specific purposes. Hence, in this thesis, we could not rely on these techniques to identify the NFRs and use them in our analysis. Therefore, we created our own dataset of NFRs (Section 5.3.2). This dataset was composed of discussions from a common resource employed by developers in OSSs, the pull requests (Soares *et al.* 2015).

Pull Requests (PRs) discussions are an important mechanism used by developers to discuss the needs of the system, such as new features and maintenance tasks (Gousios *et al.* 2014). This thesis focuses on PR discussions since they can be used to discuss the system's requirements and design (Gousios *et al.* 2014, Barbosa *et al.* 2020). A PR discussion is composed of a title, a description, and the comments of other team members (Liu *et al.* 2019). In this approach, the PR author starts a discussion by proposing a change or reporting some problem within the system. Once started, other team members can discuss the proposed change or compromise themselves to solve the problem. These discussions can also emerge code reviews made by more experienced developers in the system (Silva *et al.* 2016, Zanaty *et al.* 2018).

PR discussions can bridge the NFRs defined for the project and its current implementation. When more experienced team members engage in discussions regarding NFRs, they end up sharing their knowledge with the rest of the team participating in the discussion. This can help newcomers learn with experienced developers how to address NFR tasks in the systems. It can also help newcomers deal with problems that can emerge due to bad design decisions early in the system.

2.2 Design Problems

Design Problems result from design decisions that negatively impact the NFRs of the systems (Garcia *et al.* 2009b). The most critical design problems are the ones that affect how a system is divided into modules and how these modules communicate (MacCormack, Rusnak and Baldwin 2006, Godfrey and Lee 2000, Schach *et al.* 2002, Gurp and Bosch 2002).

Design problems related to the software system modularity are often associated with high maintenance effort (Schach *et al.* 2002, Garcia *et al.* 2009b, Yamashita and Moonen 2012, Sousa *et al.* 2018). In fact, these problems can even lead to the software system's complete discontinuation or complete redesign (Godfrey and Lee 2000, Gurp and Bosch 2002, MacCormack, Rusnak and Baldwin 2006). In addition, this kind of design problem often affects multiple software system components, making identifying these problems a time-consuming task.

Developers could use the system design documentation to identify possible problems. However, most software systems do not offer detailed documentation of the design decisions in the system. Therefore, developers need to rely on analyzing code elements that can be affected by design problems.

This thesis explores the catalog of design problems presented in Table 2.1. This catalog covers different types of design problems related to *(i)* abstractions, *(ii)* dependencies, and *(iii)* separation of concerns (Sousa *et al.* 2018). Following, we describe each of these design problems.

- **Ambiguous Interface**: This design problem happens when the interface of a component is ambiguous and has non-cohesive services (Sousa 2018). It commonly occurs in systems where component interfaces include public methods with generic types as parameters. Such a problem may hamper the error handling of a system. This happens when the exceptional interface is not properly defined without clear messages, forcing developers to use generic exception-handling mechanisms.
- **Cyclic Dependency**: This design problem happens when a stakeholder creates dependencies that create cycles. In this scenario, the components depend on each other in a circular manner (Parnas 1978). As the software evolves, this kind of design problem can make it difficult to track and manage these dependencies. Furthermore, this design problem can negatively impact the understandability, testability, reusability, and maintainability of the software system (Parnas 1972). These cyclic dependencies can also impact the exception handling performed in the classes within these components.

- **Unwanted Dependency**: This design problem happens when the stakeholders create dependencies in the software system that violates the design decisions previously defined (Perry and Wolf 1992). This design problem can hamper the changeability of the system, reducing its maintainability and testability. In addition, it can lead to incorrect exception handling in the code since a component can be dependent on another component with a different type of error handling. It can make exception handling more complex.
- **Concern Overload**: This design problem occurs when a component is responsible for realizing multiple concerns (Macia 2013). This kind of design problem can make it difficult for the developer to understand which concerns they should focus on. Hence, the proper exception-handling logic may not be adequate in this scenario. In addition, this design problem can also lead the system to irrecoverability from faults, increasing the maintenance costs and speeding up the software erosion (Gurp and Bosch 2002, MacCormack, Rusnak and Baldwin 2006, Curtis, Sappid and Szynkarski 2012).
- **Fat Interface**: This design problem is a specialization of the *Concern Overload*. It also violates the *separation of concerns principle*. It happens when the stakeholders design the interface for more than one unrelated concern (Martin and Martin 2006). This kind of problem can hamper the system's extensibility, understandability, and testability (Sousa 2018). In addition, similar to the *Concern Overload*, this kind of design problem can hamper the quality of the exception-handling logic. This is the case since the developer must evaluate multiple components before applying any mechanism.
- **Scattered Concern**: This design problem happens when the stakeholders decompose the system into dependent components rather than independent ones (Parnas 1978, Dijkstra 1997). It can cause issues in the code, such as duplicated code, hampering the readability and maintainability of the software system. In addition, if the concerns are not properly separated, it may make it difficult for the stakeholders to guarantee the security and performance of the system.
- **Misplaced Concern**: This design problem happens when a stakeholder creates an abstraction that addresses a concern that is not the predominant one in its own component. As a consequence, the module with this design problem will face other design problems such as *Concern Overload* or *Scattered Concern* (Sousa 2018). The presence of this design problem

Table 2.1: Design Problems Description

Type	Design Problem	Description
Abstraction	Ambiguous Interface	It refers to interfaces representing the abstraction that does not reveal which services it offers.
Dependency	Cyclic Dependency	Two or more components that directly or indirectly depend on each other
	Unwanted Dependency	Dependency that violates an intended design rule
Separation of Concerns	Concern Overload	Abstraction that fulfills too many concerns
	Fat Interface	Interface with multiple non-cohesive services
	Misplaced Concern	An element that implements a concern, which is not the predominant one of their enclosing component
	Scattered Concern	Multiple components responsible for realizing a crosscutting concern

may be a signal that one NFR is prioritized excessively over others. For instance, focusing only on performance may make the system poorly scalable.

We chose these design problems for the following reasons. First, identifying these design problems can be difficult, as they may not be immediately apparent during the design or development process. Second, as described, all of them clearly impact multiple NFRs of the system. Furthermore, neglecting these design problems can lead the software to high maintenance costs or even its discontinuation (Gurp and Bosch 2002, MacCormack, Rusnak and Baldwin 2006, Curtis, Sappid and Szynkarski 2012). Therefore, it is important to identify them as early as possible. In that scenario, developers can benefit from using *design problem symptoms*.

2.3 Design Problems Symptoms

Identifying design problems is not a trivial task (Ciupke 1999, Trifu and Marinescu 2005). In the case of the design problems considered in this thesis, these problems can be scattered over multiple components on the system (Garcia *et al.* 2009b). Thus, developers need to evaluate these multiple components before identifying a design problem (Trifu and Marinescu 2005). Sousa *et al.* (Sousa *et al.* 2018) identified five categories of symptoms that developers use to identify design problems. These symptoms are surface indicators of the design problems. In fact, developers tend to combine multiple symptoms, considering factors such as the quantity and diversity of these symptoms (Oliveira *et al.* 2019, Oizumi *et al.* 2018, Macia *et al.* 2012).

Given the complexity of the identification task and the lack of well-structured design documentation, developers often have to rely on source code as the symptom of such problems. These source code level symptoms are the code smells, which can affect different NFRs of the systems, such as maintainability and robustness. Since design problems can be scattered

Table 2.2: Code Smells Descriptions

Level	Code Smell	Description
Method-level	Brain Method	Long and complex method that centralizes the intelligence of a class
	Dispersed Coupling	A method that accesses many elements dispersed among many classes
	Feature Envy	A method that is more interested in another class than its own class
	Intensive Coupling	A method that is tightly coupled with other methods, and these coupled methods are defined in the context of few classes
	Long Method	A method that is long in terms of lines of code
	Long Parameter List	A method that has a long list of parameters
	Message Chain	A long chain of methods is called to implement a class functionality
Class-level	Brain Class	Long and complex class that centralizes the intelligence of the system
	Class Data Should Be Private	A class exposing its fields
	Complex Class	A class with at least one method with high cyclomatic complexity
	Data Class	A class that contains only fields and accessor methods
	God Class	A class that centralizes the system functionalities
	Lazy Class	A class with small dimension, few methods and low complexity
	Refused Bequest	A class redefining most of the inherited methods
	Spaghetti Code	A class that implements complex methods that interact between them, with no parameters and using global variables
	Speculative Generality	A class declared as abstract that has very few child classes using its methods

through multiple components of the systems, developers may be required to examine multiple methods, classes, and even entire components to identify the problems effectively. That effort could be reduced when the developers combine multiple symptoms to understand the context of the problem. In that case, the use of the non-conformities with the NFRs (*e.g.* a system facing low performance) could also be used, combined with the code smells, reinforcing the presence of a design problem and making the identification more accurate. Therefore, this thesis explores three symptom types: *maintainability smells*, *robustness smells*, and *non-conformity with NFRs*.

2.3.1 Maintainability Smells

Code smells are micro-structures in the source code that can act as surface indicators of design problems (Fowler 1999). In the case of maintainability smells, they are the code smells that affect the maintainability NFR in the system. We can divide these smells into two groups: method-level smells (*e.g.* Long Method) and class-level smells (*e.g.* God Class). As the name suggests, these smells affect the scope of only the method and the whole class, respectively. Table 2.2 presents the 16 code smells explored in this thesis. We grouped them according to their level.

Multiple studies explored using these smells as symptoms of design problems (Trifu and Marinescu 2005, MacCormack, Rusnak and Baldwin 2006, Moha *et al.* 2010, Macia *et al.* 2012, Palomba *et al.* 2014, Oizumi *et al.* 2016, Sousa *et al.* 2018, Oliveira *et al.* 2019). For instance, Sousa *et al.* (Sousa *et al.* 2020) identified that the presence of the smells *God Class*, *Feature Envy*, *Intensive Coupling*, and *Long Method* can be strong indicators of the design problem *Concern Overload*.

A *God class* is characterized by centralizing multiple functionalities and

Table 2.3: Robustness Smells Descriptions

Robustness Smell	Description
Catch Generic Exception	Catches generic exceptions such as NullPointerException, RuntimeException, and Exception in try-catch block
Method Throws Exception	Explicitly throws java.lang.Exception
Empty Catch Block	An exception is caught, but nothing is done
Catch Null Pointer Exception	Code throwing NullPointerExceptions
Rethrows Exception	Catch blocks that rethrow a caught exception
Throw New Instance of Same Exception	Catch blocks that rethrow a caught exception wrapped inside a new instance of the same type
Throw Exception in Finally	Method throwing exceptions within a 'finally'
Exception as Flow Control	Uses exception statements as a flow control device
Throw Null Pointer Exception	Method that throws a Null Pointer Exception

handling many responsibilities. On the other hand, *Feature Envy* occurs when a method is more interested in another class than its own, which suggests that the responsibilities in that method should be implemented in another class or component. This smell can lead to *Intensive Coupling*, and make the method longer, causing the *Long Method* smell. When these smells occur together, it can strongly indicate *Concern Overload*, which happens when a component fulfills too many responsibilities that should be better modularized (Garcia *et al.* 2009b).

2.3.2 Robustness Smells

Robustness smells are code smells that negatively affect the system's robustness. In the context of this thesis, we consider the robustness smells that manifest in the source code, indicating a lack of proper error handling. Table 2.3 presents the nine smells explored in this thesis.

These smells emerge since developers may not be concerned about robustness when making decisions about the system architecture. In another scenario, when they are concerned, they may not design the decisions to comply with robustness properly (Robillard 2000). For instance, when the developers or architects decide on the number of layers or how the components will be organized according to an architectural style, the mechanism to deal with robustness might not be a deciding factor.

A type of robustness smell is the *Empty Catch Block* (Cabral and Marques 2007). This type of smell occurs when a developer includes a catch statement in their code but does not provide instructions for handling the exceptions that may occur. This can be problematic as the catch block is intended to handle exceptions thrown by the system. However, developers often ignore these smells and only address them reactively when errors occur (Shah *et al.* 2010). Despite being ignored, robustness smells can indicate design

problems (Ebert *et al.* 2015).

2.3.3 Non-Conformity with Non-Functional Requirements

Ideally, in the early stages of software development, the NFRs of a system should be specified. For instance, how well a system should be updated and maintained over time or how fast a system should answer a user request. These two characteristics are related to the maintainability and performance of a system, respectively. The non-conformity with NFRs occurs when these characteristics are not met in the system. For instance, the non-conformity with the maintainability NFR may cause the system to have a high maintenance cost and even discontinuation (Curtis, Sappid and Szynkarski 2012). This non-conformity can happen due to design decisions in the early stages of software development. Therefore, when the system does not meet the NFRs defined, it can indicate to the developers the presence of design problems in the system.

For instance, a design problem such as *Cyclic Dependency* can negatively impact the performance and maintainability NFRs. This happens when a stakeholder creates a dependency that generates cycles in the components. These cycles can impact the performance of the system, causing deadlocks (Chen *et al.* 2005). When a developer notices that the system's performance falls below the predefined thresholds, it could indicate deeper problems within the system. In addition, if the developer observes that the source code is hard to maintain because the changes are propagated in others, it can reinforce the presence of the design problem *Cyclic Dependency*. Moreover, when the developer encounters difficulties in maintaining the source code due to changes affecting multiple parts of the system, it can indicate the existence of other design problems, such as *Concern Overload*.

A *Concern Overload* design problem is related to the violation of the *Separation of Concerns* principle, which aims to improve the maintainability of the system (Macia *et al.* 2012). However, the presence of this design problem negatively impacts the maintainability of the code since it makes the concerns intertwined, making it difficult for the developer to understand, modify, and maintain the code. Therefore, identifying and addressing such design problems is crucial for ensuring the software meets its NFR objectives, ultimately contributing to a more robust and reliable system.

Therefore, when the stakeholders observe a non-conformity with NFRs in the system, it can be seen as a symptom of a design problem. Combined with other symptoms (*e.g.*, code smells), it can reinforce the presence of such problems. After identifying the design problem, the subsequent action

involves mitigating or minimizing the effects of them. To achieve this objective, developers can employ code refactorings.

2.4 Removing Design Problems Through Refactoring

Refactoring is defined as program transformations that aim to keep the observable behavior of the software system while improving its internal structure (Fowler 1999). Developers can use the refactoring operations to reduce or remove the design problems (Fowler 1999, Sousa *et al.* 2020). In fact, developers can use code smells as hints of which refactoring operation should be applied to remove the design problem in the code (Fowler 1999, Bourquin and Keller 2007).

The refactorings can be divided into types described in the literature (Fowler 1999, Tsantalis *et al.* 2018). Each of these refactoring types aims to improve a specific code structure. For instance, refactoring can be applied to attributes, methods, classes, or interfaces. This thesis considers 13 types of refactoring operations, defined in Fowler's catalog (Fowler 1999). Table 2.4 presents the descriptions of each refactoring used in this thesis. These are the most common types of refactoring operations (Murphy-Hill, Parnin and Black 2009) and are also applied to indicate the presence of design problems in the system (Sousa *et al.* 2020). For each refactoring type, we present the problem and which action should be taken to solve the problem.

Refactorings that affect the system architecture are the so-called architectural refactorings (Lin *et al.* 2016, Rachow 2019, Rizzi *et al.* 2018, Stal 2014, Zimmermann 2015). This kind of refactoring comprises one or more code-level refactoring with the goal of removing design problems (Zimmermann 2015). For instance, let us consider the *Concern Overload* design problem. This problem can be removed by applying multiple *move methods*, *move fields*, and *move classes*, placing them in the right classes. This will avoid code smells such as *Feature Envy*, *God Class*, and *Long Method*, providing a better modularization for the system and avoiding the design problem.

2.5 Related Work

In this section, we explore the themes of *(i)* identification of design problems symptoms (Section 2.5.1), *(ii)* refactorings for the removal of code smells (Section 2.5.2), *(iii)* exception handling in the context of software design (Section 2.5.3), and *(iv)* developers perceptions on NFRs (Section 2.5.4).

Table 2.4: Refactoring Types

Type	Problem	Action
Extract Interface	Several clients use the same subset of a class's interface, or two classes have part of their interfaces in common.	Extract the subset into an interface.
Extract Method	Code fragments that can be grouped together.	Separate the existing code into a new method and update the original code to call the newly created method.
Extract Superclass	Two classes with similar features.	Move the common features to a shared superclass.
Inline Method	A method body is more obvious than the method itself.	Create a new method with a similar body in the class that uses it most, and remove the old method.
Move Class	A class is in a package with other classes unrelated to its concerns.	Move the class to a more relevant package, or create a new package, if necessary.
Move Field	A field is or will be, used more by another class.	Create a new field in the target class and change all its users.
Move Method	A method is or will be, used more by another class.	Create a new method with a similar body in the class that uses it most, and either turn the old method into a simple delegation or remove it altogether.
Pull Up Field	Two subclasses have the same field.	Move a field to the superclass.
Pull Up Method	Subclasses have methods that perform similar work.	Move methods with identical results on subclasses to the superclass.
Push Down Field	A field used only in a few subclasses.	Move the field to these subclasses.
Push Down Method	A behavior is implemented in a superclass used by only one (or a few) subclasses.	Move this behavior to the subclasses.
Rename Class	The class name does not reveal its purpose.	Change the name of the class.
Rename Method	The method name does not reveal its purpose.	Change the name of the method.

2.5.1 Identification of Design Problems Symptoms

Studies have explored how to identify design problems (Sousa *et al.* 2020, Oliveira *et al.* 2019, Sousa *et al.* 2018, Oizumi *et al.* 2016). Sousa *et al.* proposed a theory on how developers identify design problems in practice (Sousa *et al.* 2018). Through a multi-trial industrial experiment with five companies, they identified the different types of symptoms that developers tend to use. In addition, they identified that developers tend to use a set of heterogeneous symptoms (*e.g.*, code smells, and violation of object-oriented principles). However, they did not explore how developers could use the robustness smells as a symptom of design problems.

Li *et al.* (Li *et al.* 2014) explored the identification of design problems by using modularity metrics, identifying that modularity metrics could be used as indicators of design problems. The first metric is called the Index of Package Changing Impact (IPCI), which quantifies the independence of packages. The second metric is called the Index of Package Goal Focus (IPGF) and indicates to what extent services of a package have the same objective. Even though they present these metrics, the authors do not evaluate how the developers could use them in practice. In our study, we apply a set of metrics and evaluate them in practice with the developers.

Oliveira *et al.* (Oliveira *et al.* 2019) explored the criteria developers use to prioritize critical classes with a possible design problem. In fact, this prioritization is a step before the proper identification of design problems. The two criteria were the high quantity and diversity of maintainability smells. However, this study only focused on maintainability smells as the symptom of design problems. Thus, this thesis explores the combination of NFR robustness changes and maintainability smells as symptoms of design problems.

Multiple studies explored using code smells to identify design problems (Sousa *et al.* 2018, Sousa *et al.* 2020, Oliveira *et al.* 2019, Oizumi *et al.* 2016, Oizumi *et al.* 2019, Macia *et al.* 2012, Coutinho *et al.* 2022). Sousa *et al.* (Sousa *et al.* 2020) investigated whether maintainability smells could indicate design problems and which refactorings should be applied to remove them. By investigating 52,667 refactored elements from 50 open-source systems, they identified the smell patterns that are often associated with design problems. These patterns consist of a set of maintainability smells that appear together and, due to their nature, can reinforce the presence of design problems. For instance, when the maintainability smells *Feature Envy* and *Intensive Coupling* occur together, it can indicate the presence of a *Scattered Concern* design problem (Garcia *et al.* 2013). However, this study does not explore how

developers can use these patterns in practice.

Tufano *et al.* (Tufano *et al.* 2015) found that code smells are introduced during the evolution tasks of a system based on mining over 500k commits and manually analyzing 9,164 of them. Some studies explored developers' perception regarding maintainability smells (Palomba *et al.* 2014, Yamashita and Moonen 2013, Tufano *et al.* 2015). Palomba *et al.* (Palomba *et al.* 2014) identified that most maintainability smells are not seen as problems by developers. However, some smell like *Complex Class* and *God Class* are still perceived as design problems. Surveys have been used in previous studies to investigate code smells. Still, they have limitations, such as questions not yielding useful and valid data and difficulty in phrasing questions so that all participants understand them. Additionally, the results of these studies have not been grounded in practice.

Sharma *et al.* (Sharma *et al.* 2020) explored the relation of design problems and code smells on 3,073 open-source C# software systems. They identified that their systems' design problem *Cyclic Dependency* was the most frequent. In addition, they found that the design problems had a strong positive relation with the code smells in general, where these smells tended to cause design problems in the system. In their study, the authors focused only on maintainability smells without considering other types of smells, such as robustness smells. Hence, in this thesis, we explore the robustness smells and also how they can be related to the maintainability smells.

2.5.2 Removal of Code Smells Through Refactoring

Multiple studies investigated the use of refactoring to remove code smells (Sousa *et al.* 2020, Oizumi *et al.* 2020, Bibiano *et al.* 2019, Cedrim *et al.* 2017, Lin *et al.* 2016, Rachow 2019). Cedrim *et al.* (Cedrim *et al.* 2017) investigated how often refactoring operations were applied to elements affected by code smells. They analyzed 13 types of maintainability smells along the version history of 23 software systems. They identifier that 80% of the refactorings were applied to these smelly elements. Unlike this study, this thesis focuses on the relationship between the maintainability smells and design problems.

Oizumi *et al.* (Oizumi *et al.* 2020) proposed a suite of heuristics to help developers apply refactorings to remove code smells. To propose these heuristics, the authors proposed using patterns of refactorings often associated with code smell removal. They identified 35 patterns for the maintainability smells Feature Envy, God Class, and Complex Class. Finally, based on an evaluation

with 12 developers, the authors proposed guidelines for the recommendation of refactorings to remove code smells. In this thesis we explore the combination of patterns of maintainability smells with robustness smells.

Bibiano *et al.* (Bibiano *et al.* 2019) explored 13 refactorings and their effect on 19 maintainability smells. They identified that refactorings only on the scope of a method are more efficient in removing the code smells. However, for smells such as Feature Envy, more interrelated code transformations are needed for its complete removal. In this study, the authors solely focus on the maintainability smells. This thesis explores how a set of refactorings can be used to remove multiple smells, hence removing the design problem.

2.5.3 Exception Handling and Software Robustness

Exception handling is a programming mechanism to ensure software's robustness by allowing developers to deal with unexpected events without crashing or producing incorrect results. The use of exception-handling by developers is extensively explored (Melo *et al.* 2019, Rocha et al. 2018, Ebert *et al.* 2015, De Padua et al. 2017, Kery et al. 2016, Nakshatri et al. 2016, Asaduzzaman et al. 2016, Cacho et al. 2014). Melo *et al.* conducted a qualitative analysis on the use of exception-handling guidelines by surveying 98 developers (Melo *et al.* 2019). The authors discovered that the majority of developers (70%) reported following some form of exception-handling guideline. However, these guidelines were found to be predominantly implicit and undocumented, indicating a lack of explicit guidance. This study highlights the need for improved documentation regarding exception handling. That led this thesis to explore new ways of documenting this for the developers, such as collecting the data available on OSSs.

Cacho *et al.* (Cacho et al. 2014) conducted a study on C# projects to investigate the relationship between software system changes and their robustness. The authors analyzed 119 software versions extracted from 16 systems in different domains. They identified that C# developers frequently prioritize maintainability over robustness in various program categories, often unconsciously. Additionally, the study revealed that changes made to catch blocks often resulted in the introduction of uncaught exceptions. Therefore, in this thesis, we aim to understand the relation that robustness and maintainability may have through the analysis of smells of each type.

Barbosa *et al.* (Barbosa et al. 2014) categorized the faults that happen in exception handling. They studied two open-source systems to identify these faults. The authors identified 10 categories of exceptional faults regarding

exception handling. In addition, they classified these faults into sub-categories, which, considering these sub-categories, led to a total of 18 categories. This thesis proposes presenting the common faults (robustness smells) regarding exception handling and suggesting implications that these robustness smells may have on a higher level of the software system. This thesis explores the relationship between robustness smells and design problems.

Other studies have also examined the faults and anti-patterns commonly associated with exception-handling in code, including Ebert *et al.* (Ebert *et al.* 2015) and De Lucia *et al.*(De Padua et al. 2017). Ebert *at al.* (Ebert *et al.* 2015) conducted an empirical study on exception-handling bugs, focusing on understanding the causes, frequency, severity, and difficulty of fixing them. For that purpose, they analyzed 220 bug reports related to exception handling and 154 responses to a survey regarding the theme. They identified that usually, organizations do not consider exception handling in the main phases of the system, such as the design phase. In addition, tests and documentation for exception handling are uncommon. In this thesis, we explore the impact of not considering exception handling on a higher level in the software system, such as the software design.

2.5.4 Developers' Perception on NFRs

A few studies explore the developers' perception of NFRs (Zou *et al.* 2017, Ameller *et al.* 2012, Camacho *et al.* 2016, Rastogi and Nagappan 2016). Zou *et al.* analyzed the perception of NFRs by exploring the Stack Overflow data, a common source of information used by developers (Zou *et al.* 2017). The goal of the author was to identify the NFRs that developers tend to focus on, the difficulties they face when dealing with such requirements, and the types of NFRs most discussed over time. The authors used a topic modeling approach to analyze the stack overflow data. Next, they applied LDA (Latent Dirichlet Allocation) to summarize the topics in the corpus created. They identified that developers focus mostly on *usability* and *robustness* NFRs. In their data, the *maintainability* and *efficiency* was less discussed. The study has the limitations of using only stack overflow data, which limits the generalization of the results. In addition, it is not clear the experience of the developers involved in the study since any developer can participate in such discussions on the platform. This thesis investigates the characteristics and perceptions of developers with diverse experience in software development. In addition, we considered developers working on both open and closed-source systems, looking for a generalization in our results.

Ameller *et al.* explored how software architects consider the NFRs in their systems (Ameller *et al.* 2012). That includes the challenges and methods involved when eliciting, documenting, and validating the NFRs. This study was performed through semi-structured interviews with 15 software architects from different companies. The authors identified that NFR elicitation is an interactive process and that the NFRs are often not documented, and when the documentation existed, it was often not precise. They also identified some challenges when dealing with NFRs elicitation, such as the difficulty of balancing the NFRs with functional requirements. Among its limitations, the study reported data from only 15 software architects, which may not be representative of the whole team. They miss other stakeholders involved in the project. This thesis gathers the perceptions of more than 40 different stakeholders with different roles in their companies (*e.g.* software architects, software engineers, and software developers)

Camacho *et al.* (Camacho *et al.* 2016) investigated the challenges agile times face in testing NFRs. The study used semi-structured interviews performed with 20 participants from a single company. Thematic analysis was used to analyze the data from the interviews, and the researchers identified seven main factors that influence non-functional testing in agile projects. They identified that agile teams face challenges mainly related to priority, culture, and awareness. In addition, they identified seven factors that influence non-functional testing: team experience, communication, customer involvement, tool support, test automation, NFR analysis, and non-functional testing strategy. This study has the limitations of having a small sample of participants from a single company and focusing only on performance and security testing. In this thesis, we performed a deeper qualitative analysis over a subset of these factors: NFR analysis, communication, and team experience. That allowed us to observe developers in different positions within their teams and from different companies. In addition, we investigate the different reasons that lead developers to address the NFRs.

3
Smell Patterns as Indicators of Design Degradation

A version of the work in this chapter appears in the **Proceedings of the 35th Brazilian Symposium on Software Engineering (SBES)**, (Oliveira *et al.* 2022).

As mentioned in Section 2.2, design problems result from one or more design decisions that negatively impact the NFRs of a system (Martin and Martin 2006). Degradation problems are design problems that gradually worsen over time as software evolves, potentially impacting the software's overall quality and adherence to NFRs. These problems are typically related to the long-term maintenance and evolution of a software system. They tend to affect key design elements, which include such as classes, methods, hierarchies, or even multiple software components. As degradation problems accumulate, they can lead to increased maintenance effort, higher error rates, and reduced system performance. Detecting and addressing degradation problems is essential to ensure that software systems continue to meet their NFRs.

It is worth noting that although we employ the term architectural problems throughout this chapter, they are fundamentally design problems. We focus on design problems that affect how the system will be modularized into software components and how these components interact with each other (hence, the use of the term "architectural"). Given their wide effect in the design, they represent a major threat to the system and they might also manifest when developers implement and refine design decisions along software changes.

When facing design problems related to the bad modularization of the system, developers must evaluate multiple elements, such as classes and packages, to identify and remove the problem. Therefore, this is a time-consuming and high-effort task, especially because the systems often do not have design documentation. As a result, developers need to use the only artifact available, the source code. In that case, developers can rely on code smells.

Certain groups of maintainability code smells, called smell patterns, may indicate specific design problems (see Section 2.5.1). This type of smell is a well-explored symptom of design problem (Sousa *et al.* 2020, Sousa *et al.* 2018). However, there is not yet an investigation on how developers can benefit from maintainability smell patterns in practice. Intending to investigate the

usefulness of smell patterns to help developers identify and remove design problems and avoid design degradation, we conducted a quasi-experiment with 13 professional developers.

This chapter presents this study, which is reported in the paper *Smell Patterns as Indicators of Design Degradation: Do Developers Agree?* (Oliveira *et al.* 2022). This study comprehends the first major contribution of this Ph.D. thesis, which can be characterized as follows: *how developers can benefit from the use of smell patterns in the identification and removal of design problems in practice* (see Section 1.4).

3.1 Introduction

The internal quality of a system depends on how it meets non-functional requirements such as maintainability (Ciupke 1999, Sousa *et al.* 2018). A design degradation problem happens when one or more design decisions have a negative impact on non-function requirements (Sousa *et al.* 2018). Degradation problems can impact isolated code elements (*e.g.*, a method or a class) or multiple code elements and structures (*e.g.*, hierarchies or components) (Sharma *et al.* 2020). The former is called *Implementation Problem* and is usually represented by code smells, which are code-level structures that are a surface indication of a design degradation (Fowler 1999). The latter is known as *Architectural Problem*, which is a degradation that affects how the system is decomposed into modules and how they communicate with each other (Garcia *et al.* 2009, Garcia *et al.* 2009b, Lippert 2006).

Studies have shown that degradation problems can be harmful to the software system (Curtis, Sappid and Szynkarski 2012, Ernst *et al.* 2015). When neglected, these problems hinder software maintenance and evolution, potentially harming business operations (Curtis, Sappid and Szynkarski 2012). Thus, developers need to remove these problems as early as possible. As the design documentation is usually unavailable, developers have to directly analyze the source code, using symptoms such as code smells and metrics to identify degradation problems (Sousa *et al.* 2018). In fact, the literature shows that developers are not only familiar with code smells, but they can also use them to identify degradation problems (Yamashita and Moonen 2013, Sousa *et al.* 2018). Developers use refactoring to remove or reduce degradation problems in the system (Fowler 1999). Since code smells are used in practice to identify low-level code refactorings (Fowler 1999), developers benefit from the use of a smell-based approach to reinforce their adoption of refactorings (Sousa *et al.* 2020).

Multiple studies investigated the relation between code smells and degradation problems (Fontana *et al.* 2019, Martins 2020, Oizumi *et al.* 2015, Oizumi *et al.* 2016). For instance, Sousa *et. al.* (Sousa *et al.* 2020) investigated patterns of smells that can potentially indicate refactoring opportunities (*e.g.*, the multiple smell pattern with *God Class* and *Complex Class*). However, there is no evidence on whether developers would benefit from these patterns.

In this paper, we investigated the use of smell patterns in practice, for two sub-groups of degradation problems, namely implementation and architectural problems. We conducted a quasi-experiment with 13 professional software developers. Then, we explored whether the smell patterns help developers to identify implementation problems and architectural problems in practice. This experiment allowed us to investigate to what extent smell patterns help developers to identify and refactor both degradation problems.

We confirmed in practice the findings of a previous study carried out by Sousa *et al.* (Sousa *et al.* 2020). We found evidence that multi-smell patterns are the most relevant for detecting architectural problems. We observed that they may be strong indicators of architectural problems such as *Concern Overload*, *Scattered Concern*, and *Fat Interface* (Brown *et al.* 1998). We also highlight some factors that should be considered to improve the identification of architectural problems with smell patterns. For instance, the identification of degradation problems was sensitive to the context of the system (*i.e.*, the degradation problem was considered inevitable due to the domain of the system). We also observed cases in which the developers stated that there was an implementation problem but their description of the problem was indicating an architectural problem. Hence, we observed that for the architectural problems, developers still need more support regarding the metrics and smells provided. Finally, we observed that refactoring associated with code smells contributes to the partial removal of degradation problems. With these findings, developers can better understand how the combined code smells lead them to not only identify the degradation problem but also remove it with refactoring operations. Finally, we also provided an automated tool to help developers to identify the refactoring opportunities.

3.2 Background

We discuss the here basic concepts and terminology of our work.

3.2.1 Design Degradation Problems

In this work, we considered two categories of design degradation: architectural problems and implementation problems.

An *architectural problem* is a degradation that negatively impacts high-level structures, such as components, abstractions, hierarchies of code elements, and other structures that are relevant to the software architecture (Bass et al. 2003). These problems affect, but are not limited to, (i) how the system is organized into subsystems and components, (ii) how and which code elements encapsulate process and data to address each functionality, and (iii) how the elements interact with each other and their execution environment (Garcia *et al.* 2009, Garcia *et al.* 2009b, Lippert 2006). These types of architectural problems are often harmful in software systems (MacCormack, Rusnak and Baldwin 2006, Schach *et al.* 2002). An example of an architectural problem is the *Fat Interface* (*i.e.*, an interface that provides multiple unrelated services) that usually harms quality attributes such as modifiability and extensibility (Martin and Martin 2006). Since this degradation problem can affect multiple components in the system, developers may need more support to identify and remove them.

An *implementation problem* is a degradation that negatively impacts fine-grained elements such as methods and classes (Sharma and Spinellis 2018). An example of an implementation problem is when a method is too long and complex to understand. In that case, the implementation problem could be signaled by the code smell *Long Method*. This problem negatively affects the readability of the source code, but since it is more localized, it might have less impact on the architecture.

3.2.2 Smell Patterns as Indicators of Refactoring Opportunities

Once identified the degradation problems, the next step is to remove them. Developers can use code smells as hints for the refactoring operations that can potentially remove the problem (Bourquin and Keller 2007, Fowler 1999). Smells not only indicate the presence of a degradation problem, but they can also indicate which refactoring operations the developers should apply (Fowler 1999). For instance, Sousa *et al.* found that smell patterns (*i.e.* groups of one or more types of smells) can indicate architectural patterns (Sousa *et al.* 2020).

Table 3.1 presents the two groups of smells patterns investigated by Sousa *et al.* (Sousa *et al.* 2020): Multi-Smells Patterns (`MSP`) and Single-Smell Pat-

Table 3.1: Architectural Problems and their Smell-Patterns

Architectural Problem	**Multi-Smells Pattern**
Ambiguous Interface	Long Method, Feature Envy, and Dispersed Coupling
Cyclic Dependecy	Intensive Coupling and Shotgun Surgery
Component Overload	Shotgun Surgery, Divergent Change, Feature Envy, God Class/Complex Class, Intensive Coupling, and Long Method
Concern Overload	Complex Class, Divergent Change, Feature Envy, God Class/Complex Class, Intensive Coupling, Long Method, and Shotgun Surgery
Fat Interface	Shotgun Surgery or Divergent Change, Dispersed Coupling, and Feature Envy
Misplaced Concern	God Class/Complex Class or Dispersed Coupling, Feature Envy, and Long Method
Scattered Concern	Dispersed Coupling, Divergent Change, Feature Envy, God Class/Complex Class, Intensive Coupling, and Shotgun Surgery
Unwanted Depedency	Feature Envy, Long Method, and Shotgun Surgery
Architectural Problem	**Single-Smell Pattern**
Incomplete Abstraction	Lazy Class
Unused Abstraction	Speculative Generality

terns (`SSP`). `MSP` indicates the patterns where the degradation problems can be identified through the use of multiple smells. For instance, the architectural problem *Cyclic Dependency* is indicated by the pattern composed of two smells: *Intensive Coupling* and *Shotgun Surgery*. When a single smell indicates a degradation problem, it has an `SSP`. For instance, the architectural problem *Incomplete Abstraction* can be indicated by a *Lazy Class* code smell. When these patterns appear, they indicate both the degradation problem and the refactorings that should be applied by the developer (Sousa *et al.* 2020). In that, case, we call this a refactoring opportunity. For instance, for the aforementioned `MSP`, the developers could apply Move Method, Extract Method, Move Field, and Inline Class refactorings. The full list of common refactorings for each smell explored in this study is detailed Appendix A.1.

The use of smell patterns has pros and cons. In the case of `SSP`, the developer only has to reason about a single smell to identify and refactor the problem. However, this single smell may not be enough to confirm the problem. Considering the `MSP`, the presence of smells in the pattern may increase the confidence regarding the presence of a degradation problem; thus, they can apply the refactoring suggested. However, reasoning about and connecting multiple smells is a time-consuming task. Yet, we do not have evidence of whether developers would benefit from the patterns in practice.

3.3 Study Design

We conducted a quasi-experiment (Shadish, Cook and Campbell 2001) asking developers to identify degradation problems in their software systems by analyzing different cases of smell patterns. The following subsections present the study settings.

3.3.1 Research Question

With this quasi-experiment, we have the goal of *understanding how the smell patterns can be used by developers to identify and refactor degradation problems in their systems*. With this goal in mind, we defined two research questions.

RQ1. *Are Smell Patterns Indicators of Degradation?*

First, we wanted to understand whether the smell patterns could be good indicators of degradation problems, as claimed by Sousa *et al.* (Sousa *et al.* 2020). To answer `RQ1`, we asked the participants of our study to identify degradation problems in their software systems. We asked them to identify these problems in three scenarios: using (i) Single-Smell Pattern (`SSP`), (ii) Multiple Smell Pattern (`MSP`), and (iii) other combination of smells (`Others`), which are smells different from the ones in the patterns.

Upon data collected, we conducted quantitative and qualitative analyses. The purpose of quantitative analysis was to provide an objective answer to our RQ. For that purpose, we applied two statistical tests: (i) One-Way ANOVA (Shaw and Mitchell-Olds 1993), and (ii) Chi-Square Test (McHugh 2013). After that, we applied qualitative analysis to understand the results in more details.

RQ2. *Can Smell-Patterns Indicate Refactoring Opportunities?*

Once identified the problems, the next step is to remove them, which can be done through the use of refactorings. Since the smell type can indicate the refactoring that should be performed (Fowler 1999), we wanted to understand how the smell patterns could help the developer in this task. For that purpose, we defined `RQ2`.

To answer `RQ2`, besides presenting information on metrics and code smells, the developers were provided with suggested refactorings for each smell (Section 3.2.2). We asked the participants if the suggested refactorings were sufficient to remove the identified degradation problems. With their answer, we aimed at understanding the scenarios where developers accepted, partially

accepted, or rejected the suggested refactorings. More information about the hypothesis and variables are available at Appendix A.2

3.3.2 Recruiting Participants

To select the participants, we applied convenience sampling by using the collaborators' network of contacts in the industry and also in academic groups. We also looked for potential participants in our social media (Twitter and LinkedIn). We defined criteria to select the final list of participants (*e.g.*, Intermediary knowledge about Java and Software Architecture). This information was given by developers through a characterization form. Detailed characteristics of the 13 participants are available at Appendix A.3.

We asked the participants to inform us which system they were familiar with, which was the basis to select the subject systems of our study. This familiarity is related to the contributions that the developer made to the system analyzed. Thus, we also collected information about the software systems selected by participants. Table 3.2 presents a summary of the software systems. In the last column, we show the ID of participants that analyzed each system. We identify each participant by an identification number (ID) that we will use to reference them.

Table 3.2: List of the Software Systems Analyzed by Participants

System	Type	Domain	Size	Part.
OPLA-Tool	OS	Academic/Tool/Model Optimization	Large	1
Fresco	OS	Mobile/Library/Media Management	Large	3
Fastjson	OS	Library/Parser	Large	6
Soot	OS	Academic/Tool/Compiler Optimization	Large	12
Couchbase Java Client	OS	Database driver	Medium	2, 10
JDeodorant	OS	Tool/Plugin/Source Code Analysis	Small	8
REST System	CS	REST API	Small	4
School Mgt System	CS	Web system/School Management	Small	5
Image Composition System	CS	Library/Media Manipulation	Small	11
Grace Language Compiler	OS	Academic/Compiler	Small	7
SportsTracker	OS	Desktop/Personal App	Small	9
Glide	OS	Library/Android/Caching/Media Loading	Medium	13

Type: OS = Open Source, CS = Closed Source.
Size (lines of code): Small = <100K, Medium = >100K and <499K, Large = >499k.

3.3.3 Study Procedures

The quasi-experiment to answer our research question has five main steps. All the participants had to follow them.

Step 1: Training. Participants went through basic training about study-related concepts, namely structural software quality, degradation problems, source code metrics, code smells and refactoring. The goal of this training was to make sure all the participants had an understanding of the terms used during the study.

Step 2: Environment Setup. To support our analysis, we collected information about the systems indicated by the participants (*e.g.*, internal structure metrics, smells, and smell patterns). For that purpose, we ran an adapted version of the Organic tool (Oizumi *et al.* 2018), which is a metric-based smell detector, in the participants' systems. We adapted the tool so we could suggest to the developers the degradation problems and the proposed refactorings to remove them. After analyzing the source code, the tool selects the code elements that the participants should analyze. The criteria for selecting code elements are the number of commits and the number of smells, in this priority order. We adopted such selection criteria because there is evidence that they are relevant for software maintainability (Oizumi *et al.* 2019).

Step 3. Experimental Task. After collecting all the required information, our adapted version of Organic provided six different cases to be analyzed by each participant. A case is composed by one or multiple code elements that are affected by code smells. The six cases are divided into three groups: (i) two affected by `MSP`, (ii) two affected by `SSP`, and (iii) two affected by `Other` smells.

Our adapted tool provides the information for each case according to the groups above. For example, suppose a participant is analyzing a case using one of the patterns. Then, our tool (i) provides information about code metrics (*e.g.*, coupling) and code smell(s), (ii) provides the list of smells in the pattern and a description of the possible degradation problem occurring in the case, and (iii) suggests the refactorings to remove the problem.

Each participant selected a different system to analyze. There were cases in which the tool was unable to detect all six cases involving the smell patterns and other combinations of smells. Therefore, in some scenarios the participant evaluated less cases (*e.g.* four) or evaluated more than two cases involving other combinations of smells. This happened since not all systems have the six cases. Thus, in these systems, the developers analyzed more than one instance of the same pattern.

Step 4: Case Analysis. The participants had to analyze each case according to the information provided by the tool. Then, they filled out a form where they should inform whether they agree there was a structural degradation problem or not. In the cases that participants considered there was NO degradation

problems, they had to provide a justification to their conclusion. They also had to rate the usefulness of the information provided by the tool to support their conclusion. More details about this form are available at Appendix A.6.

Step 5: Post-study Interview. At the end of the study, we invited each participant for an interview. We asked them about external factors that may have affected the experiment (*e.g.*, interruptions or problems with the tool). We also asked them about the desirable characteristics that a degradation detection tool should have.

3.3.4 Data Analysis Procedures

Below we present the procedures to conduct our analysis.

Quantitative Analysis. For the quantitative analysis, we considered the responses according to the three groups, namely Single Smell Pattern (`SSP`), Multiple Smell Pattern (`MSP`), and Other combination of Smells (`Others`). For this evaluation, we compared the precision of participants in identifying implementation problems and architectural problems. In this study, *Precision* is the number of cases classified by participants as representing degradation/architectural problems divided by the number of analyzed cases. To assess the statistical significance, we applied the One-Way ANOVA (Shaw and Mitchell-Olds 1993). This statistical test is useful for comparing the use of different treatments (`MSP`, `SSP`) and a control (`SSP`) by the same group of participants.

We also compared the different groups of responses concerning the proportion of smell pattern cases indicating architectural problems to the proportion of smell pattern cases indicating implementation problems. Hence, we aimed to understand in which granularity the patterns can be more precise. We also compared such proportions between the two different groups (`MSP`, `SPP`) and the control (`Others`) using the Chi-Square Test (McHugh 2013). Finally, we analyzed the precision of each smell pattern for indicating architectural problems.

Qualitative Analysis. The participants also provided descriptive responses to justify and explain each case analyzed. Therefore, we conducted a systematic analysis of descriptive responses. In this analysis, two collaborators worked together to categorize the responses into groups that help to explain the qualitative results. After this procedure, all collaborators discussed and improved the categorization of responses and the extraction of groups. Such analysis was inspired by procedures commonly used in qualitative research methods such as Grounded Theory and Content Analysis (Lazard *et al.* 2017).

For the categorization of responses, we defined five categories that reflect relevant scenarios for the evaluation of the use of smell patterns. In the first (C1) and second (C2) categories, we grouped the responses in which participants confirmed the existence of an architectural problem. C1 includes the cases in which the participants agreed that the suggested refactorings could remove the problem. C2, on the other hand, is the case where developers suggested that the refactorings would not completely remove the architectural problem. In the third (C3) and fourth (C4) categories, we included the participants' responses that agreed or not with the suggested refactorings, respectively. However, differently from previous categories, they considered that exists (a certain level of) degradation but not an architectural problem. In the last category (C5), we grouped the responses in which participants considered that there was no occurrence of degradation problems. This category aims to identify which factors lead participants to conclude that the analyzed smells do not represent degradation problems. The groups and categories created during this analysis will not be directly presented, but they are available in our replication package (1st Complementary 2022).

3.4 Results and Discussion

In this section, we present the results of our study for evaluating the use of smell patterns for finding refactoring opportunities.

3.4.1 Are Smell Patterns Indicators of Degradation?

Table 3.3 presents the results for the classification of the six cases analyzed by each participant. The last six columns show the precision of participants in each group between parentheses. The last table row summarizes the number of cases and precision of each group. Except for participants 4 and 11, all of them analyzed at least two cases in each group. The reason is that the systems analyzed by both participants did not have any instances of `MSP` cases.

When analyzing the precision of participants in the different groups (Table 3.3), it is possible to observe that the use of `MSP` tends to result in a higher precision when compared to `SSP` and `Others`. We observed an overall precision of 0.77 for `MSP`, 0.66 for `SSP`, and 0.60 for `Others` (last row). To compare the precision of participants in the different groups we applied the One-Way ANOVA Test.

Table 3.3: Number of Cases and Precision of Each Participant Using MPS, SSP and Others.

Id	# of MSP	# of SSP	# of Others	MSP-DP	SSP-DP	Others-DP	MSP-AP	SSP-AP	Others-AP
1	2	2	2	2 (1.00)	2 (1.00)	2 (1.00)	0 (0.00)	0 (0.00)	1 (0.50)
2	2	2	2	2 (1.00)	2 (1.00)	0 (0.00)	2 (1.00)	1 (0.50)	0 (0.00)
3	2	2	2	2 (1.00)	2 (1.00)	2 (1.00)	1 (0.50)	0 (0.00)	2 (1.00)
4	0	1	3	0 (-)	0 (0.00)	1 (0.33)	0 (-)	0 (0.00)	0 (0.00)
5	2	1	3	2 (1.00)	0 (0.00)	2 (0.67)	2 (1.00)	0 (0.00)	2 (0.67)
6	2	2	2	2 (1.00)	2 (1.00)	1 (0.50)	1 (0.50)	0 (0.00)	0 (0.00)
7	2	2	2	0 (0.00)	2 (1.00)	2 (1.00)	0 (0.00)	2 (1.00)	0 (0.00)
8	2	2	2	2 (1.00)	2 (1.00)	1 (0.50)	2 (1.00)	2 (1.00)	0 (0.00)
9	2	2	2	2 (1.00)	0 (0.00)	2 (1.00)	0 (0.00)	0 (0.00)	0 (0.00)
10	2	2	2	1 (0.50)	0 (0.00)	2 (1.00)	0 (0.00)	0 (0.00)	1 (0.50)
11	0	2	4	0 (-)	1 (0.50)	2 (0.50)	0 (-)	0 (0.00)	0 (0.00)
12	2	2	2	1 (0.50)	2 (1.00)	1 (0.50)	1 (0.50)	2 (1.00)	1 (0.50)
13	2	2	2	1 (0.50)	1 (0.50)	0 (0.00)	0 (0.00)	0 (0.00)	0 (0.00)
All	**22**	**24**	**30**	**17 (0.77)**	**16 (0.66)**	**18 (0.60)**	**9 (0.40)**	**7 (0.29)**	**7 (0.23)**

ID: Participant ID; **# of `MSP`, `SSP`, `Others`**: Number of cases with the patterns. **`MSP-DP`, `SSP-DP`, `Others-AP`**: Number of cases classified by participants as being affected by degradation problems; **`MSP-AP`, `SSP-AP`, `Others-AP`**: Numbers of cases classified by participants as being affected by architectural problems.

Before applying the test, we had to remove participants 4 and 11 since they did not use all treatments (*i.e.*, they did not analyze at least two cases in each group). After applying the statistical test with an alpha level of *0.05*, we observed a F-ratio of *0.242* and a p-value of *0.787*. Thus, we cannot confirm the difference between precision values of using smell patterns and other smells. The F-ratio is smaller than 1, which indicates that the use (or not) of smell patterns alone is not enough for explaining the variance of precision in the identification of degradation problems. Despite the results of the statistical test, we were able to find the factors that most influenced the precision of the different groups.

3.4.1.1 Factors that Impact the Use of a Smell-based Approach

These factors were described by the developers in the post-experiment interview. Next, we describe the most recurrent factors.

Context sensitive degradation detection. A recurrent observed factor is that certain degradation problems were considered inevitable in certain contexts. For example, participant 13 informed that the degradation indicated in the implementation by the *Message Chain* and *Long Parameter List* smells is common and acceptable in Android applications (*e.g.*, the high number of parameters and subsequent calls to a method were needed by the API used). In cases involving the pattern for the architectural problem *Incomplete*

Abstraction (*i.e.* when an element does not completely support a responsibility in its own component), we observed that participants frequently mentioned contextual factors to justify the fact that it does not represent any kind of design degradation.

Use of customized strategies and thresholds. For the detection of code smells, we adopted default detection strategies and thresholds from the literature (Lanza and Marinescu 2006). However, we observed that such strategies and thresholds were not sufficient for the developer to identify the degradation problem in all scenarios. For instance, we observed a case of *Long Parameter List* for which the participants did not agree with the number of parameters threshold. There were also cases of *Speculative Generality* for which the participants argued that the abstract methods of an abstract class were used in a sufficient number of sub-classes. This observation is corroborated by other studies (*e.g.*, (Hozano *et al.* 2017, Hozano *et al.* 2018)) that indicate the need for customized detection strategies for code smells. These analyses lead to the following finding:

> ***Finding 1:*** *Factors such as the context of the system and the strategies used for the detection of code smells may influence the perception of the developer regarding the presence of architectural problems.*

3.4.1.2 Cases That Smell Patterns Indicate Architectural Problems

When classifying a case as being affected by an architectural problem, the group with the highest precision was `MSP-AP` (0.40) (Table 3.3). Considering the cases classified as having a degradation problem, the highest precision was `MSP-DP` (0.77). The lowest precision for degradation problems was for `Others-AP` (0.60). Thus, the participants' precision in finding architectural problems was much smaller than their precision in finding any types of degradation problems, regardless of the use of patterns. This happened because there were cases in which the participants considered that there was a degradation problem, however, this problem was not relevant to the architecture of the system. According to them, there was (a certain level of) degradation but it was not an architectural problem.

To search for a possible association of the occurrence of patterns with the existence of architectural problems, we applied the Chi-Square Test. Table 3.4

Table 3.4: Results of the Chi-Square Test for the Association of Groups With Architectural Problems, Implementation Problems and no Problems.

	# Arch. Problems	**# Imp. Problems**	**# Non-Degradations**	**Row Total**
MSP	9 (6.66) [0.82]	8 (8.11) [0.00]	5 (7.24) [0.69]	22
SSP	7 (7.26) [0.01]	9 (8.84) [0.00]	8 (7.89) [0.00]	24
Other	7 (9.08) [0.48]	11 (11.05) [0.00]	12 (9.87) [0.46]	30
Column Total	23	28	25	76

presents the contingency table for this test. The 2^{nd} column presents, for each group (`MSP`, `SSP`, and `Others`), the number of cases that were classified as having implementation problems (*i.e.*, degradation problems that are not related to the architecture). The 3^{rd} column shows the number of cases classified as having architectural problems. The 4^{th} column presents the number of cases classified as not having any kind of degradation. In the last column, we show the total number of cases in each group. The last row shows the totals of each column. In the cells of 2^{nd}, 3^{rd}, and 4^{th} columns of Table 3.4, we also show, inside the parenthesis, the expected distributions according to the totals in each row and column. Values in square brackets represent how much each cell contributed to the Chi-Square statistic. The higher the difference between the expected value and the observed value, the higher the contribution to the Chi-Square statistic. For instance, the expected number of cases with architectural problems for the `MSP` group is *6.66*, but the actual number of cases was *9*, resulting in a contribution of *0.82* to the Chi-Square statistic.

The highest differences are observed in cases with architectural problems and cases without degradation (Table 3.4) for the `MSP` and `Others` groups. The number of `MSP` cases (9) with architectural problems is higher than expected (6.66), while the number of `Others` cases (7) with architectural problems is lower than expected (9.08). However, by applying the Chi-Square test with an alpha level of *0.05*, we obtained a Chi-Square statistic of *2.467* and a p-value of *0.650*. Thus, by the test, we can conclude that there is no correlation between the evaluated patterns and the existence of architectural or implementation problems. Given such a lack of correlation, in the following sub-sections, we present qualitative analysis that helps us to understand and explain the results.

3.4.1.3 Developers' Perspective on Architectural Problems

We observed contradictory cases that were classified as not being affected by architectural problems but the developers' justifications seem to reference

architectural problems. For example, when analyzing the `Couchbase Java Client` system, participant 10 classified one case involving a `MSP` as not having degradation. The MPS was intended to indicate a *Scattered Concern* problem involving the `CouchbaseAsyncBucket` class through the following combination of smells: *Dispersed Coupling*, *Feature Envy*, *God Class*, *Complex Class*, and *Intensive Coupling*. According to the open response presented by the participant, such a class would be affected by a functionality (concern) that is scattered across different classes.

Since this system is a well-structured open source system, we can hypothesize that the decision to keep the functionality scattered may have been conscious indeed. However, this does not mean that there is no architectural problem, since a scattered functionality may affect the extensibility and modifiability of the software system. Creating abstractions to isolate a feature can make the software design more complex. In this sense, developers may have decided to favor code design simplicity over other quality attributes.

We also observed cases involving patterns that were classified as implementation problems but the description given by the developer indicated architectural problems (considering the literature definition (Martin and Martin 2006)). For instance, when relying on a `MSP` to analyze the `Facebook Fresco` library, participant 3 indicated the existence of an implementation problem in the `FrescoController` Java interface and its implementations. This participant provided the following description of this problem:

"The classes are very dependent and coupled with the FrescoDrawback2 class. All of these classes and methods need to call various methods of that class to meet their responsibilities."

On our qualitative analysis, we identified how participant 3 proceeded with the analysis. In this case, the participant identified that (i) there was feature envies in some methods, (ii) the class had a high complexity (being a *Complex Class* smell), and (iii) through the metrics, the high coupling of that class. This information helped in the identification of a degradation problem, even if the participant does not have full knowledge about the architectural impact of such a problem. In this case, the MPS was intended to indicate the *Fat Interface* problem, which occurs when an interface exposes many functionalities that are not related to each other. The fact that one class is tightly coupled to another does not necessarily mean that there is a *Fat Interface* problem. However, high coupling is a negative characteristic that directly affects the architecture and may be indirectly related to the existence of a *Fat Interface*.

We believe that participant 3 may have initially conducted a shallow

analysis of the case, without considering a possible impact on the architecture. Nevertheless, we also believe that in this case, information about metrics and smells assisted the participant to unconsciously identify an architectural problem. Even though the participant thought that there was no architectural problem, a degradation problem related to high coupling was detected. The participant identified the problem through the combination of the smells and the presented system metrics.

Given the aforementioned analysis, smells and metrics can be a starting point for identifying whether there is a degradation problem, and in further analysis, identify and remove the architectural problem. These results confirm the findings of previous studies (Sousa *et al.* 2018, Sousa *et al.* 2020) that stated that the complete identification and removal of the architectural problem should be complemented by other information (*e.g.*, the concerns that are implemented in each class). In the case of participant 3, the information about concerns would be useful for identifying the presence of multiple unrelated services.

Other cases similar to those described above also happened for other participants. This indicates that the developers' understanding of what an architectural problem may not be aligned with the literature describes(Brown *et al.* 1998). We hypothesize that many developers only consider architectural problems to be those that were not intentionally introduced and those that explicitly affect many components of the architecture. This shows that, although smell patterns can indicate the existence of architectural problems in relevant scenarios, they still do not provide enough information to make evident the causes and the possible impact caused by the detected architectural problems. These results lead to the following finding:

> ***Finding 2:*** *Developers perceive architectural problems as those which they did not intentionally introduce and that explicitly affect many components of the architecture.*

3.4.1.4 Evaluating Specific Smell Patterns

To better understand when smell patterns are indicators of architectural problems, we analyzed individual types of patterns. Table 3.5 presents the results for the assessed pattern types. The 2^{nd} column shows the number of

evaluated cases of each pattern type. The 3^{rd} and 4^{th} columns show the number of cases with any kind of degradation problem and with only architectural problems, respectively. In both columns, we show within parenthesis the precision of each pattern type. Finally, the 5^{th} column presents the mean severity, on a scale from 1 (lowest) to 5 (highest), according to the classifications provided by participants.

Table 3.5: Number of Cases, Precision and Mean Severity of Each Pattern Type.

Pattern Type	# Cases	# Degradation	# Arch. Problems	Mean Severity
Multi-Smell Patterns				
Concern Overload	3	2 (0.66)	1 (0.33)	4.50
Fat Interface	6	6 (1.00)	3 (0.50)	3.50
Scattered Concern	6	5 (0.83)	2 (0.33)	4.60
Unwanted Dependency	7	4 (0.57)	3 (0.42)	3.25
Single-Smell Patterns				
Incomplete Abstraction	6	3 (0.50)	2 (0.33)	3.00
Unused Abstraction	17	13 (0.76)	5 (0.29)	3.38

Detection of Unwanted Dependencies. An *Unwanted Dependency* occurs when a dependency violates a rule defined on the system architecture (Perry and Wolf 1992). The precision of our *Unwanted Dependency* pattern was *0.57* for degradation problems and *0.42* for architectural problems. Given the lack of information about architectural rules, this precision can still be considered high. Unfortunately, architectural rules are not always defined and documented. In such cases, the pattern would be useful for the detection of *Unwanted Dependencies*. Participant 12 informed us in the post-experiment interview that, besides information about smells, the tool should provide explicit information about the dependencies that involve the affected code elements. These results lead us to hypothesize that the usefulness of the pattern for *Unwanted Dependency* is limited by the context of the system.

High Precision for the Fat Interface pattern. The pattern type with the highest precision was the one for *Fat Interface*, with *1.00* for degradation problems and *0.50* for architectural problems. The pattern for *Fat Interface* is composed by a *Shotgun Surgery* in the interface or *Dispersed Coupling* and *Feature Envy* in elements that are clients or implement the interface. We believe that one of the reasons for the high precision of this pattern is the fact that it makes clear the impact on hierarchical structures composed of interfaces and classes. In addition, smells like *Dispersed Coupling* and *Feature Envy* are strongly associated with the architectural concepts of high coupling and low cohesion. Although other patterns are also composed of smells that can reveal a relevant impact on the architecture, the pattern for *Fat Interface* makes it clear that multiple interrelated code elements are affected. Thus, the identification of

this problem can be simplified, otherwise, the developer would have to analyze these multiple code elements.

High severity for Concern Overload and Scattered Concern patterns. Participants classified cases involving the *Concern Overload* and *Scattered Concern* patterns with the highest severity. The mean perceived severity is *4.50* for *Concern Overload* and *4.60* for *Scattered Concern*. This indicates that, even though they were not classified as architectural problems in many cases, these patterns were considered to be highly severe. Both patterns involve the *God Class* and *Complex Class* smells. To corroborate, Sousa *et al.*(Sousa *et al.* 2020) observed that such smells were recurrent indicators of architectural refactoring opportunities. Thus, the high severity observed in this experiment is consistent with the results observed in their study. As we discussed above, the precision of these patterns could be higher if we had provided additional information about the dependencies and concerns of each code element. The results for these patterns lead us to the following finding:

> ***Finding 3:*** *The patterns for the architectural problems* `Fat Interface`*,* `Concern Overload` *and* `Scattered Concern` *are the most promising for identifying architectural refactoring opportunities. They become even more useful and precise if complemented with information about dependencies and concerns.*

Once we investigated how the smell patterns can help the developers in the identification of degradation problems, our next step was to understand whether these patterns can also help the developers to remove the problems through refactoring.

3.4.2 Can Smell-Patterns Indicate Refactoring Opportunities?

We asked the developers to inform us whether the refactorings suggested by our tool were sufficient to remove the degradation problem identified. They had to inform if they accepted, partially accepted, or rejected the suggested refactorings. Table 3.6 summarizes their answers.

For `SSP`, in most cases, participants either accepted or partially accepted the refactorings. The main reasons for the partially accepted refactorings were (i) the effort needed to conduct the suggested refactorings, and (ii) not fully understanding the design of the software system. In those cases, the architectural problem was *Unused Abstraction*. For such a problem, the participant

had to understand where the abstraction could be used. Therefore, deeper knowledge about the system design was necessary. To illustrate this recurrent observation, we may take as an example the justification of participant 2 for one of the cases:

"The refactoring suggestion, as useful as it may be, may not be just what is necessary to solve the problem. It potentially needs a reevaluation of the design itself to discover the real use of that abstract class, and why no other class uses it."

We observed that even if the developer understands the system design, the refactorings can be difficult. As suggested by participant 10, the refactorings indeed would reduce the complexity of a class, however, with a high effort cost. In that case, even though the participant recognized that there is an architectural problem, the participant thought the trade-off between the class complexity and the effort was not worthwhile. Besides analyzing the rejected and partially accepted refactorings, we also analyzed the cases when the participants accepted the suggested refactorings. In such cases, we noticed that the refactorings for simpler smells were more widely accepted. By simpler, we mean those smells that are easier to understand or, at least, those that participants are familiar with. Examples of such smells are *Long Method* and *Long Parameter List*.

Table 3.6: Number of Refactoring Suggestions that Were Accepted, Partially Accepted, or Rejected.

	SSP	**MSP**	**Others**
Accepted	9	6	9
Partially Accepted	6	11	6
Rejected	1	0	2

Regarding the `MSP`, we found that the participants partially accepted most of the suggested refactorings for `MSP` cases (Table 3.6). Among the causes for them partially accepting the refactorings, we observed (i) the complexity of the degradation problem, (ii) the use of design patterns, and (iii) the context of the software system.

Participants mentioned, for example, that multiple code elements with smells were presented, which made the analysis difficult. One of the obstacles was how to relate the smells and metrics of different code elements to identify the main problem. For that matter, they accepted only some refactorings since they were not fully aware of the whole architectural problem affecting these classes. Another cause for partially accepting the refactorings was related to the possible modification related to a higher-level abstraction. In these cases,

the participants mentioned that instead of only applying the refactorings they would also make more massive changes, such as introducing design patterns. For instance, one of the reasons mentioned by participant 10 was the context of the systems. Even though there was an architectural problem, the participant mentioned that this was needed due to the APIs used on the system. Hence, only some refactorings were accepted, since other cases of the smell were required for API usage. Following, we present an example of this scenario involving participant 10:

"This is a library that serves as an API for communication with a database and this is a central abstraction of the database. Thus, it is justified that this class adds a high amount of functionality so that (i) this abstraction is not spread over many classes, and (ii) API usability is also facilitated."

When the participants accepted all suggested refactorings, we observed that they used our tool to identify problems that at a glance could be considered as complex. For instance, participant 1 mentioned that some problems were hard to identify due to the complexity of the system. However, using the provided source code, smells, and metrics, the participant managed to identify and fully understand the problem.

In addition, in the accepted cases, some participants also mentioned the time constraints for the system development. Due to such constraints, they recognized the introduction of implementation and architectural problems during the development and evolution of the source code. For example, participant 5 provided the following justification in one of the cases:

"Due to development time, I simplified this implementation, adopting conditionals, instead of using polymorphism."

As reported by the participant, this simplification led the class to become a *Complex Class* and to have methods with *Message Chains*, *Long Methods*, and *Feature Envies*. Combined with other smells, they were indicators of the *Unwanted Dependency* problem.

For the `Other` category, we noticed that the participants did not accept the refactorings especially due to intentional decisions. For instance, participant 10 mentioned that even though the class analyzed was indeed big and complex, it was an intentional decision. In the `Other` cases where the participant partially accepted the refactoring, we noticed that this is related to the complexity of the problem. For instance, participant 3 suggested the implementation of a design pattern to start solving the problem. This was suggested since the analyzed class had duplicated functions. To implement this pattern, the participant suggested applying *Extract Class* to create new separated classes.

In this case, refactorings suggested to remove *long parameter lists* and *long methods* would not be enough. Similar to the case where they do not accept the refactorings, the domain of the system was also impacted when they only partially accepted the refactorings.

Based on the aforementioned results, we conclude that the refactorings associated with code smells in the smell patterns can help the developer to remove or, at least, partially remove degradation problems. Nevertheless, removing architectural problems tends to be more challenging. We have identified scenarios in which more complex refactorings (*e.g.*, introduction of a design pattern) were required. Therefore, we conclude that our smell-based approach is more useful when it provides automated support for performing the suggested refactorings. Hence, in this study, we developed an automated tool that, based on the smell patterns, suggests to the developer the degradation problems and the proposed refactoring remove them. To assist developers in removing architectural problems, we believe that the suggested refactorings in our tool could be customized and optimized using machine learning or search-based algorithms (Boukhdhir *et al.* 2014). This discussion leads to our last finding:

> ***Finding 4:*** *Refactorings associated with code smells contribute to the (partial) removal of degradation problems. However, more complex problems, such as the architectural ones, require complementary information (e.g., customized thresholds for the metrics of internal quality).*

3.5 Related Work

In a previous work, Sousa *et al.* investigated when smells are indicators of refactoring opportunities (Sousa *et al.* 2020). These authors analyzed 52,667 refactorings from 50 open-source systems. The main contribution of their study was the identification of smell patterns that are often associated with degradation problems. In this paper, we complement this related work by evaluating the use of smell patterns in practice. Our evaluation was based on a quasi-experiment involving 13 professional software developers, that helped us to reach new findings. For instance, we identified when patterns are actually useful in practice. We have also identified several factors that must be taken into account for developers to be more precise in identifying architectural problems through code smells. Next, we discuss other studies that are closely related to this work.

3.5.1 Code Smells and Degradation Problems

Code smells have been explored by multiple studies (Macia 2013, Macia *et al.* 2012b, Moha *et al.* 2010, Oizumi *et al.* 2016, Oliveira *et al.* 2019, Tufano *et al.* 2015). Tufano *et al.* (Tufano *et al.* 2015) explored when and why developers introduce code smells. They mined over 500k commits and manually analyzed 9,164 of them, identifying that the smells are introduced during the evolution tasks of a system. Some studies explored the perception of developers regarding the code smells (Palomba *et al.* 2014, Yamashita and Moonen 2013, Tufano *et al.* 2015). Palomba *et al.* investigated specifically whether developers recognized smells as degradation problems (Palomba *et al.* 2014). Through the analysis of 12 distinct smells and three open-source systems, they identified for most code smells, developers do not see as actual problems. Nevertheless, some smells were perceived as problems in the source code (*e.g.*, *Complex Class*, and *God Class*). However, these previous studies rely on surveys to investigate code smells. Some issues appear when using surveys to gather information (Easterbrook *et al.* 2008). For instance, the questions in a survey may not be designed in a way that yields useful and valid data. Another issue is to phrase the questions insomuch that all participants understand them in the same way, especially when the target population is diverse. Another limitation of these studies is that their results have not been grounded in practice.

Fontana *et al.* (Fontana *et al.* 2019) conducted an empirical study to evaluate if architectural smells are independent of code smells. Their results indicate that there is no strong correlation between architectural smells and code smells. However, they used an automated tool for detecting architectural smells. Thus, their architectural smells do not necessarily represent architectural problems, as there can be false positives and false negatives. Differently from them, we evaluated the use of a smell-based approach with professional developers on a quasi-experiment. We relied on the expertise of the developers to evaluate degradation problems. Therefore, our work evaluates the relationship between smells and both architectural and implementation problems from a different perspective.

3.5.2 Identification and Refactoring of Design Degradation Problems

Cedrim *et al.* (Cedrim *et al.* 2017) investigated the frequency that refactoring operations are applied to smelly elements. They found that almost 80% of refactoring operations are applied to smelly elements. In their study,

the authors focused on the relation between smells and refactoring. We focus on the relation between smells and design degradation problems, in which we conveniently use refactoring to find the relevant problems. In addition, differently from them, we conducted a quasi-experiment to assess the usefulness of a smell-based approach for finding refactoring opportunities.

Studies have explored how to identify design degradation problems (Ran *et al.* 2015, Sousa *et al.* 2018). For instance, Sousa *et al.* proposed a theory to describe the factors that influence how developers identify design degradation problems in practice (Sousa *et al.* 2018). However, these studies focus only on the identification of design degradation problems. They do not explore how these smells can be used in practice for both the identification and refactoring of these problems.

Different from the aforementioned studies, we performed a study with professional developers, where we investigated the use of a smell-based approach for finding refactoring opportunities in practice. Besides evaluating the use of a smell-based approach, we also developed an automated tool that is fully available to be extended and used by other researchers and developers. Details about this tool are available at Appendix A.5 and in our replication package (1st Complementary 2022).

3.6 Threats to Validity

The time allocated for the quasi-experiment can pose a threat to the validity of this study. To mitigate this threat, we performed a pilot study to adequate the number of cases and the time spent analyzing each case. We also present the cases of different groups in random order to avoid the cases of a certain group always remaining at the beginning or at the end of the experiment. Finally, we recorded the time spent analyzing each case and asked the participants if they felt tired during the experiment. The analysis of each case lasted an average of about 11 minutes. Most participants reported that they did not feel tired. Two participants were asked to partition the experiment into two parts to prevent tiredness.

As the experiment was carried out remotely, it was not possible to control external variables such as interruption by third parties or the occurrence of technical problems. We mitigate this threat by asking each participant to report the occurrence of any interruptions or technical problems. Two participants suffered interruptions caused by third parties and one of the participants' internet went down for 10 minutes. Nevertheless, the participants informed us that such problems did not hinder their performance during the

experiment.

The number of participants represents another threat to validity. We would need a larger sample of participants to further understand the results. To mitigate this threat, we complemented our quantitative analysis with systematic qualitative analysis. In fact, qualitative research requires the study of specific situations and people, complemented by considering specific contextual conditions (Yin 2015). We selected 13 professional developers with diverse experience, which are representative individuals of our target population. Thus, we consider that this threat was mitigated.

During the experiment we had some threats. We used an automated tool to collect metrics, detect smells and find patterns. Thus, our results are influenced by the accuracy and reliability of the tool. To reduce this threat, we extended a tool that was extensively used in related studies for collecting metrics and detecting code smells (*e.g.*, (Cedrim *et al.* 2017, Sousa *et al.* 2018, Oizumi *et al.* 2016)). We also conducted manual tests with multiple open-source systems to identify and remove possible defects in the tool. Finally, we conducted a qualitative assessment that helped us to identify cases in which the accuracy of the tool influenced the results.

Our tool provides an output with the degradation problem affecting the system, which could introduce a confirmation bias. To mitigate that, the developers analyzed cases that had patterns indicating the problem and cases that did not have any pattern. The developers' preference for the system analyzed could also introduce a confirmation bias. To mitigate that, the participants had different levels of familiarity with the systems.

3.7 Conclusion

To assess the use of smell patterns in practice, we conducted a quasi-experiment with 13 professional developers. Each participant evaluated multiple cases of smell patterns and other combinations of smells regarding their relation with architectural and implementation problems. This evaluation was conducted in the context of software systems they were familiar with. The participants also indicated whether the detected degradation problems were present at the architectural or implementation level. Finally, they indicated whether the refactorings associated with the smells would be sufficient to remove each degradation problem.

We observed that developers tend to agree that smell patterns are indicators of degradation problems in certain cases. This agreement is influenced by several factors. Examples of such factors are development platform (*e.g.*,

Android), type of functionality, and effort for refactoring the problem (see Section 3.4.1.1). In practice, developers can benefit from the use of patterns such as multi-smell patterns since they contain multiple smells that can be cumbersome to analyze apart. Our tool presents the information needed (*e.g.* smell patterns and suggested refactorings) to remove/reduce the degradation problem, likely saving developers time/effort. In future work, we intend to use the findings presented in this paper to improve our smell-based approach with new empirical studies. We also intend to improve our tool, based on the feedback that the developers gave us.

4
Considering Robustness Changes to Identify Design Problems

A version of the work in this chapter appears in the **Proceedings of the 20th International Conference on Mining Software Repositories (MSR)** (Oliveira *et al.* 2023).

In the previous chapter, we provided insights on how maintainability smell patterns can be used to effectively identify design problems (see Chapter 3). The quasi-experiment with professional developers showed that smell patterns could indicate design problems. The agreement among developers regarding the existence of design problems within the system is influenced by factors like the development platform, the type of functionality, and the level of effort required to refactor and remove the problem. There were scenarios where the maintainability smell patterns were not enough for the design problem identification, and the developers needed more information so they could have a more comprehensive understanding of the design problems. This occurred because depending solely on maintainability smells introduced a limitation related to the static analysis tools used for identifying these smells (see Section 3.4.1.1). These tools rely on metrics and thresholds that might prove inadequate for certain system contexts. This limitation related to static analysis tools is aggravated by developers' need for more contextual information about specific classes or components to identify and address design problems accurately.

Relying solely on a limited set of smells may result in an incomplete design problem identification. Considering that design problems negatively impact NFRs on the system, exploring a new symptom (*i.e.*, code smells) related to a new type of NFR can be an option to give the developer more information regarding a problem. Thus, in this chapter, we explore robustness smells. Robustness smells can complement the maintainability smells and reinforce the presence of design problems. Robustness smells are complementary, as they tend to appear in the exceptional code of the system (*i.e.* the code that handles unexpected behaviors in the system), while the maintainability smells appear commonly in the normal code (*e.g.* the code that deals with the expected behavior). Therefore, some information that the

maintainability smells may not capture can be complemented by the robustness smells.

In this context, we investigated how developers can use robustness smells combined with maintainability smells to identify design problems. To perform our analysis, we collected the commit history of more than 160k methods from various releases of 10 open-source software systems. We first investigated how the robustness changes (*i.e*, changes performed within the catch block) are correlated maintainability smells. Second, we explored whether robustness changes could be related to the number of maintainability smells in a class or method. Finally, we investigated how poor robustness changes (*i.e.*, the changes affected by robustness smells) could be used with the maintainability smells patterns for the identification of design problems.

This chapter presents the paper *Don't Forget the Exception! Considering Robustness Changes to Identify Design Problems* (Oliveira *et al.* 2023). This study comprehends the second major contribution of this Ph.D. thesis: *how robustness smells can be combined with the pattern of maintainability smells for the identification of design problems* (see Section 1.4).

4.1 Introduction

Exception-handling mechanisms, commonly utilized in modern programming languages, promote the robustness and stability of the software systems (Shah *et al.* 2010, Weimer and Necula 2008). The proper use of these mechanisms aims to guarantee the software integrity when unexpected events or behaviors happen (IEEE 1990). However, most software systems do not offer detailed documentation of the design decisions related to the exception handling implementation (Ebert and Castor 2013, de Lemos and Romanovsky 2001). This lack of information encourages developers to focus solely on the normal behavior of the software system (Robillard 2000), leaving the exception handling behavior poorly implemented (Reimer and Harini 2003, Shah *et al.* 2010) or even neglecting the exceptional code (de Lemos and Romanovsky 2001, Jakobus *et al.* 2015). This neglection can impact the software system's robustness and might also be a sign of problems in the design of the software system (Kechagia and Spinellis 2014, Coelho *et al.* 2015, Oliveira *et al.* 2016).

A design problem results from one or more design decisions that negatively impact the system's non-functional requirements (NFRs), which include robustness and maintainability (Garcia *et al.* 2009b, Li *et al.* 2015, Lim *et al.* 2012). The most critical design problems often affect how the system is modularized into components and how these components interact with each other.

To identify such design problems, developers have to analyze several elements (*e.g.*, classes and packages), which is a laborious activity. Thus, in this study, we focus on design problems related to system modularity. An example is the design problem *Concern Overload* that occurs when a component is responsible for realizing multiple concerns (Macia *et al.* 2012). This design problem can also make it difficult for developers to know which concerns they should focus on to create the proper exception-handling logic. Furthermore, when neglected, these design problems can lead the system to undesired consequences such as irrecoverability from the faults, increasing the maintenance cost, and speeding up software erosion (Gurp and Bosch 2002, MacCormack, Rusnak and Baldwin 2006, Curtis, Sappid and Szynkarski 2012).

Since these design problems can affect multiple NFRs, they must be identified and removed from the systems as soon as possible (de Mello *et al.* 2023). Multiple studies explored the use of maintainability smells as symptoms of design problems (Oizumi *et al.* 2016, Sousa *et al.* 2018, Coutinho *et al.* 2022). A recent study presented a catalog with patterns of maintainability smells that indicate multiple design problems (Oliveira *et al.* 2022). However, more smells may be considered, since developers may need more context regarding the class, component, or system. Moreover, this can cause an incomplete identification of the design problem. In addition, these studies do not explore how exceptional code (*i.e.*, code inside the catch block) can be combined with maintainability smells to identify design problems. Given the different natures of normal and exceptional code, it is possible that they can complement each other to identify design problems. Developers could benefit from tools that, besides detecting multiple symptoms, also combine them for revealing design problems (Sousa *et al.* 2018). Therefore, in this study, we aim to understand how the poor changes in exception handling can be used as symptoms of design problems and how they can be combined with maintainability smells to identify these problems.

For this study, we consider the changes related to robustness as the changes performed within the catch blocks, since the exceptional code is in this part of the implementations. We analyzed over 160k class methods from 10 open-source software systems. For our analysis, we collected (i) maintainability smells based on insights from a related study (Oliveira *et al.* 2022), (ii) robustness changes in methods, and (iii) robustness smells. In the first analysis, we explored how robustness changes could correlate with maintainability smells. Furthermore, we looked for maintainability smells that were introduced through robustness changes. Our goal was to identify how these two factors were correlated. In a second analysis, we identified whether robustness changes could have a negative impact on classes with methods

that underwent this kind of change. Hence, we could understand how these changes impacted the method's degradation. Finally, we investigated which poor robustness changes (signaled through robustness smells) could be used with patterns of maintainability smells to identify design problems.

We identified that a method with robustness changes would also be affected by a *Feature Envy*, *Dispersed Coupling*, or *Long Method* maintainability smells. By manually analyzing the methods with robustness changes, we identified cases where these changes introduced the maintainability smells, especially the *Feature Envy*. We also identified that classes with robustness changes had a higher density of smells when compared to classes without such changes. Hence, these robustness changes could indicate these classes' degradation. We also observed that the robustness smells *catch of generic exceptions* and *empty catch block* tended to co-occur with the patterns of maintainability smells that help in the identification of design problems such as *Concern Overload* and *Unwanted Dependency*.

Our results support the community in understanding how poor robustness changes can complement the information given by maintainability smells to identify design problems. In practice, developers can use this information to be aware that even minor modifications made to catch blocks can potentially affect or reveal underlying design issues within the system. Moreover, by identifying the robustness smells as a new symptom of design problems, tools can be developed to reinforce the presence of these problems.

4.2 Background

This section describes the concepts for understanding the relationship between poor robustness changes, and maintainability code smells, so they can be used to identify design problems.

4.2.1 Exception Handling

An *exception* is an unexpected event that occurs during the execution of a program, interrupting its normal behavior (Goodenough 1975). Usually, exceptions manifest through errors. When an error happens, the method where it appears creates an *exception object* including information, such as the program's state and details about the error. The *exception object* is then delivered to the runtime system, completing the initial routine that *throws an*

exception[1].

Developers use exception-handling mechanisms to ensure that the system will be in a consistent state, even after errors that occur at runtime (Goodenough 1975), assuring that the system will be robust. In Java, any code snippet likely to throw exceptions must be placed inside a `try-catch` block. The `try` block defines the `normal code` of a method. Respectively, each `try` is followed by one or more `catch` blocks, which handle specific exceptions thrown inside the associated `try`. The `catch` block has the code responsible for handling the specific exception types that can emerge from the enclosed instructions in the `try` block. In addition, the `catch` block encompasses the method's `exceptional code`. These blocks can also be followed by a `finally` block that always executes after the `try` and the `catch` if an exception is raised.

4.2.2 Robustness Changes and Code Smells

In this study, we consider as *robustness changes* those performed within the catch block, since the poor use of exception-handling mechanisms can harm the software robustness (Kechagia and Spinellis 2014, Coelho *et al.* 2015, Oliveira *et al.* 2016). We consider the *poor robustness changes* as the changes in the catch block that are affected by robustness smells. An example of a robustness smell is the *Empty Catch Block*, which occurs when a developer creates a catch statement but leaves its content empty. This is a problem because the catch block should be where the developer handles exceptions thrown by the system, which is not occurring. Unfortunately, developers tend to ignore these smells and only deal with them reactively when they face errors (Shah *et al.* 2010). Albeit ignored, robustness smells may indicate the decay of the software (Ebert *et al.* 2015). Other indicators of software degradation are the maintainability smells (Fowler 1999, Lanza and Marinescu 2006, Uchôa et al. 2020), that can also signal design problems (Sousa *et al.* 2017, Sousa *et al.* 2018, Oizumi *et al.* 2016, Oliveira *et al.* 2019, Oliveira *et al.* 2022). However, they are not always sufficient for this identification task. Hence, since both maintainability and robustness smells can indicate design problems, combining them could reinforce its presence.

To understand this relationship between poor robustness changes and maintainability smells, let us consider the following example illustrated by Figure 4.1[2]. This figure displays the HTTPHandler system, which implements an HTTP client. Figure 4.1(a) shows the layered architectural style followed

[1] *Java Documentation.* What Is an Exception? Available at `https://docs.oracle.com/javase/tutorial/essential/exceptions/definition.html`, last accessed on 2021-07-16.

[2] For simplicity, we adapted this example from one of the analyzed systems.

by the architecture of the system we analyzed, which consists of four layers: `Interface`, `Cache`, `Connection`, and `Control`. According to the architectural style, each layer is in charge of its responsibility, thus following the *Separation of Concerns* (SoC) principle (Dijkstra 1997, Parnas 1972).

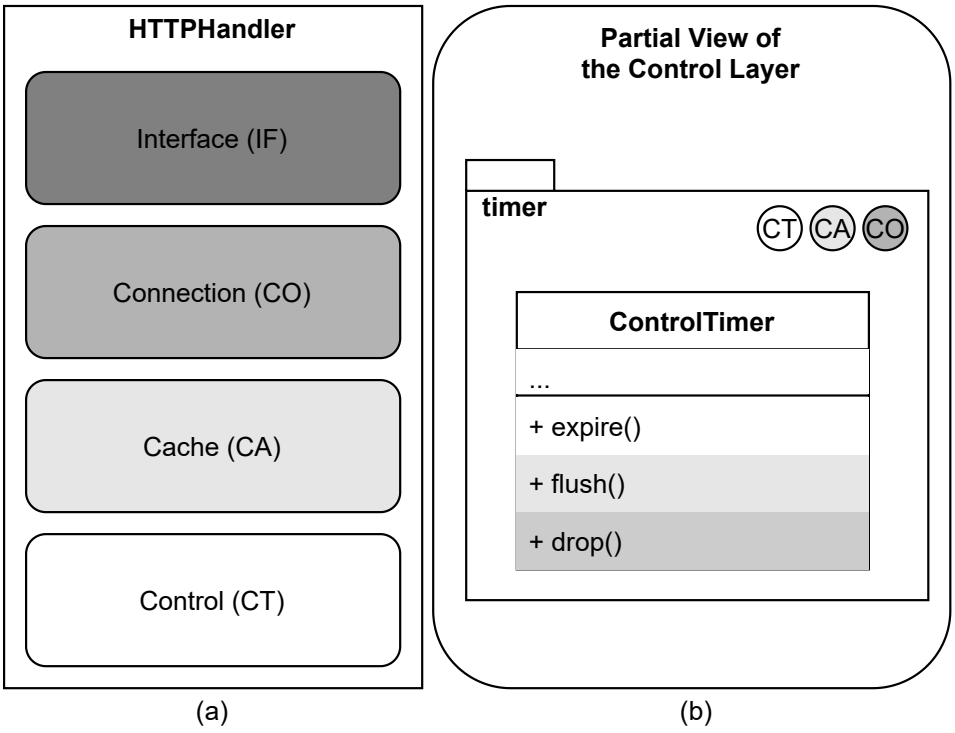

Figure 4.1: Partial View of the HTTPHandler System

Like other design decisions, one might expect that every exception handled in each layer is directly related to the responsibility (*i.e.*, the concern) implemented in the layer. However, this is not always the case. For example, Figure 4.1(b) shows the `ControlTimer` class with three methods, where `flush()` and `drop()` implement responsibilities from two other layers: `Cache` and `Connection`, respectively. When we look at the `expire()` method (illustrated in Figure 4.2), we see a code snippet that shows a change was performed in the class in which the developer tried to handle an error. However, as observed in the Figure 4.2, it catches a generic exception to handle the error, which is a bad practice since it hides the error that the catch block should have captured and handled. In addition, within the catch block, nothing is handled, only logged with a generic message.

We hypothesize that this happened due to the several concerns intermingled in the implementation of that class. This method is also affected by the code smell *Intensive Coupling*, which indicates that this class has a high coupling with the other classes in the module. At the time, the developer could not know which specific exception the block should handle. As afore-

```
public class ControlTimer{
  (...)
  public void expire(Control ct,
                     Connection conn){
    try {
      (...)
    } catch (Throwable t) {
      log.debug(ct.getServiceKey()
      + "and" + conn.getUrl())
    }
```

Figure 4.2: Example of a Method Catching a Generic exception

mentioned, the class implements concerns from other layers. Consequently, the many concerns caused the developer to neglect proper exception handling in the method. Furthermore, besides just logging the error instead of handling it, the log method also calls multiple methods from the other classes, introducing a *Feature Envy* and reinforcing the *Intensive Coupling* smell that the method already had. Additionally, these two maintainability smells are part of a pattern that can strongly indicate the design problem *Concern Overload* (Oliveira *et al.* 2022, Sousa *et al.* 2020). The patterns considered in our study are presented in Table 4.1. Therefore, in this example, the poor robustness change and the smell pattern together reinforced the design problem's presence. Moreover, this study explores the potential of using these co-occurring factors to identify design problems.

Table 4.1: Design Problems and their Smell-Patterns

Design Problem	**Smells Pattern**
Ambiguous Interface	Long Method, Feature Envy, and Dispersed Coupling
Cyclic Dependency	Intensive Coupling and Shotgun Surgery
Concern Overload	Divergent Change, Feature Envy, God Class/Complex Class, Intensive Coupling, Long Method, and Shotgun Surgery
Fat Interface	Shotgun Surgery or Divergent Change, Dispersed Coupling, and Feature Envy
Misplaced Concern	God Class/Complex Class, Dispersed Coupling, Feature Envy, and Long Method
Scattered Concern	Dispersed Coupling, Divergent Change, Feature Envy, God Class/Complex Class, Intensive Coupling, and Shotgun Surgery
Unwanted Dependency	Feature Envy, Long Method, and Shotgun Surgery

4.3 Related Work

Multiple studies have explored the relationship between maintainability smells and design problems (Sousa *et al.* 2018, Sousa *et al.* 2020, Oliveira *et al.* 2019, Oliveira *et al.* 2022, Oizumi *et al.* 2016, Oizumi *et al.* 2018, Macia *et al.* 2012). Oliveira *et al.* (Oliveira *et al.* 2019) identified the criteria that developers used to prioritize the classes that, with respect to their degradation, were most critical in a system. The authors found that developers tend to consider the quantity and diversity of maintainability smells in a class as an important factor in its prioritization. Thus, the developers should focus their effort on these degraded classes. The limitation of these studies regards using only the maintainability smells as the symptom of design problems. Sousa *et al.* (Sousa *et al.* 2018) developed a theory on how developers identify design problems. They identified the developers' use of multiple symptoms, including maintainability smells and violation of non-functional requirements (NFRs). Thus, our study explores the NFR robustness (by considering the robustness changes) and its combination with the symptom maintainability smell.

Studies have shown that maintainability smells can be indicators of design problems (Sousa *et al.* 2017, Macia *et al.* 2012a, Macia *et al.* 2012b, Coutinho *et al.* 2022). A pattern of maintainability smells (*i.e.* groups of one or more types of smells) can be a strong indicator of the presence of design problems (Oliveira *et al.* 2022, Sousa *et al.* 2020). For instance, when the maintainability smells *Feature Envy* and *Intensive Coupling* occur together, they indicate that the method, affected by these smells, is more interested in data from others, calling many methods from unrelated classes(Lanza and Marinescu 2006). Hence, these smells can indicate the presence of a *Scattered Concern* design problem (Garcia *et al.* 2013). However, these smells may not be sufficient to confirm the presence of design problems (Oliveira *et al.* 2022). Among the reasons, more information regarding the context of the method and class is needed, such as the system's domain. Thus, in this study, we expand the use of these smells with information on the exceptional code, which can introduce more context (*e.g.*, through the type of exception handled) about the method and class analyzed. Best of our knowledge, the relationship between robustness changes and code smells still needs to be explored.

The use of exception-handling by developers is extensively explored (de Mello *et al.* 2023, Rocha et al. 2018, Ebert *et al.* 2015, De Padua et al. 2017, Kery et al. 2016, Nakshatri et al. 2016, Asaduzzaman et al. 2016, Cacho et al. 2014). Melo *et al.* qualitatively analyzed the use of exception-handling guidelines by surveying 98 developers (Melo *et al.* 2019). The authors identified

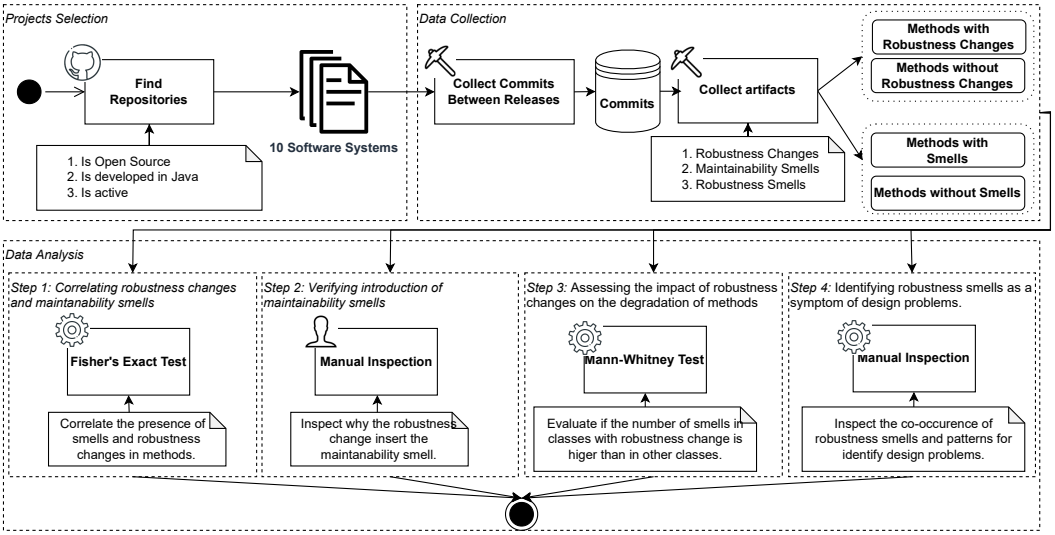

Figure 4.3: Workflow of our Study Design.

that in 70% of the developers' responses, there was a guideline to be followed. However, these guidelines tend to be implicit and undocumented. Cacho *et al.* (Cacho et al. 2014) presented a study with C# projects where they identify the relationship between software systems changes and their robustness. They analyzed 119 software versions extracted from 16 systems from different domains. They identified that C# developers often unconsciously traded robustness for maintainability in various program categories. Finally, it was identified that a high number of uncaught exceptions were also introduced when the catch blocks were changed. Other studies explored the faults and anti-patterns commonly related to the exception-handling in the code (Ebert *et al.* 2015, De Padua et al. 2017). In this study, we explore how poor changes in the exceptional code can impact the maintainability smells of the system.

4.4 Study Design

In summary, we analyzed commits from 10 open-source software systems. We started by collecting maintainability smells, robustness smells, and robustness changes in the commits between selected pairs of releases from those systems. We detail the remainder of the study design as follows. It is also summarized in Figure 5.1.

4.4.1 Research Questions

Our goal is to *understand how poor robustness changes can be combined with maintainability smells as complementary symptoms of design problems.*

With this goal in mind, we defined three research questions.

RQ_1: How often do robustness changes co-occur with maintainability smells?

We hypothesize that robustness changes and maintainability smells can be considered in combination (Section 4.2.2) and have the potential to reveal design problems. Therefore, we performed a statistical analysis of the correlation between robustness changes and maintainability smells using Fisher's exact test (Fisher 1922). We started by dividing methods regarding robustness changes they underwent between two releases and the presence of maintainability smells in these methods.

Alongside understanding whether those robustness changes and maintainability smells correlate, we also want to understand if these changes could introduce the maintainability smells. Thus, we selected the methods with robustness changes and verified whether the maintainability smells were introduced during those changes (see Section 4.4.4 - Step 1).

RQ_2: What impact can robustness changes have on the degradation of classes?

Once we answer RQ_1, we should have indications of whether robustness changes and maintainability smells are correlated and if these changes can introduce smells. After knowing that, we aim to understand whether performing robustness changes to methods can impact the degradation of classes. For RQ_2, similarly to previous work (Sousa *et al.* 2018, Oliveira *et al.* 2019, Oizumi *et al.* 2016), we considered degradation as the number of maintainability smells a method has (*a.k.a.* density of smells). Developers use this metric to prioritize classes with a high number of different smells when looking for design problems (Oliveira *et al.* 2019) (see Section 4.4.4 - Step 2).

RQ_3: How do robustness smells give evidence of design problems?

After identifying if robustness changes can have a negative impact on maintainability smells, we also want to investigate further how we can use poor robustness changes as symptoms of design problems. To answer RQ_3, we analyze when robustness smells co-occur with maintainability smells that

are part of patterns that indicate design problems (see Section 4.2.2). With this RQ, we aim to identify which robustness smells can complement these patterns of maintainability smells, reinforcing the presence of design problems (see Section 4.4.4 - Step 3).

Consequently, by addressing all three research questions, we will gain insights into how and which poor robustness changes can be used with the patterns of maintainability smells to assist developers in detecting design problems.

4.4.2 Defining Subject Software Systems and Releases

We firstly selected projects from a list of projects used in related studies (Coutinho *et al.* 2022, Oliveira *et al.* 2022, Barbosa et al. 2014, Oizumi *et al.* 2019). Then, we filtered the systems to be used in this study using the following criteria:

Open Source. We selected open-source software systems to allow full replication of this work, since access to closed software systems is usually very limited. Open-source software systems often rely on version control systems (*e.g.*, Git) to track the evolution of their source code. This gives us access to the complete history of changes (*i.e.*, commits) to the source code of a software system, allowing us to perform the multiple analysis required to address our three research questions.

Java Language. We consider systems written in Java since it provides an exception-handling mechanism designed to help developers to build robust systems (Barbosa et al. 2014). For instance, the use of checked exceptions forces developers to write handlers for certain errors. In addition, we selected projects from different domains that are more inclined to follow robustness requirements (*e.g.*, distributed computing, and big data processing). Moreover, Java has a wide availability of static analysis tools and libraries that can automatically identify source code problems such as Organic (Oizumi *et al.* 2018), PMD[3], and SonarQube[4];

Active Software Systems. To consider a software system *active* we considered four criteria: (i) its Git repository has more than 1,000 commits, (ii) should contain commits pushed a month before our data collection period, (iii) should have recent discussions on pull requests and issues, and (iv) should have recent releases. These criteria ensure that the systems still have a development activity and relevance to the developers participating. After

[3] *PMD*. Available at `https://pmd.github.io/`, last accessed on 01/19/2023

[4] *SonarQube*. Available at `https://www.sonarsource.com/products/sonarqube/`, last accessed on 01/19/2023

Table 4.2: Details on the Software Systems Analyzed

Project	Start Release	End Release	Commits		Methods
			With Any Changes	With Robustness Changes	
apm-agent-java	v0.5.0	v1.28.0	1,025	484	9,949
dubbo	dubbo-2.6.12	dubbo-3.0.0	1,382	1,069	26,613
elasticsearch-hadoop	v1.3.0.M1	v8.0.0	1,120	503	3,894
fresco	v1.0.0	v2.6.0	1,165	791	7,700
netty	netty-4.1.31.Final	netty-4.1.75.Final	995	694	19,067
okhttp	parent-2.3.0	parent-3.14.0	586	455	5,933
RxJava	v1.0.10	v3.1.0	1,169	799	28,004
spring-boot	v2.7.6	v3.0.0	1,380	618	19,330
spring-framework	v5.3.24	v6.0.0	1,119	794	38,429
spring-security	5.1.0.RELEASE	5.6.0.RELEASE	1,426	690	10,062

Commits with any changes: Number of commits with any kind of change
Commits with robustness changes: Number of commits with robustness changes
Methods: Total number of methods between releases

following these criteria, we settled on 10 software systems: APM Agent, Dubbo, Elasticsearch Hadoop, Fresco, Netty, Spring Boot, Spring Security, Spring Framework, RxJava, and OkHttp.

Given that the releasing strategy can differ across software systems, we selected start and end releases that span at least 1,000 commits. Our goal is to avoid the bias of having only a few robustness changes and thus being unable to perform reliable conclusions. Therefore, we can detect subtle patterns and correlations in the data that may not have been apparent with fewer robustness changes. This led us to more generalizable, accurate, and robust conclusions. In Table 4.2, we present the systems and the releases selected.

4.4.3 Collecting Artifacts Data

To answer our RQs, we collected robustness changes, maintainability smells, and robustness smells from 10 target systems. We first collected the maintainability smells using *Organic* (Oizumi *et al.* 2018). *Organic* is a static code analyzer that collects software metrics (Lanza and Marinescu 2006) for maintainability smells detection. For our first two RQs, we only consider method-level smells (*e.g.*, Feature Envy, and Dispersed Coupling). We selected these smells since they are part of the patterns that help identify design problems (see Table 4.1). When evaluating those design problem patterns (*i.e.*, RQ_3), we also consider class-level maintainability smells (*e.g.*, God Class, and Complex Class).

To collect robustness changes, we developed a Python script that calculates the difference between two commits and identifies any change within the catch block of a method body. We also filtered out the test code. Finally, we collected nine robustness smells (*e.g.*, empty catch block, and catch generic

exception), using *PMD*, which is a static code analyzer, often used to find flaws in source code. Details about the smells and the script developed can be found at Appendix B.1.

4.4.4 Data Analysis

In this section, we present the steps to the analyses executed to answer our research questions as follows (see Figure 5.1).

Step 1: Correlating robustness changes and maintainability smells: To answer RQ_1, we first divided methods as follows: (i) methods with at least one robustness change, (ii) methods that changed but did not have any robustness change, (iii) methods that were affected by at least one maintainability smell, and (iv) methods that were unaffected by maintainability smells. Considering this division, we defined the following pair groups to be used in Fisher's exact test (Fisher 1922).

- **Smelly + Changed (SML + CH)**: Methods with maintainability smells and robustness changes
- **Smelly + Not Changed (SML + NoCH)**: Methods with maintainability smells without robustness changes
- **Not Smelly + Changed (SML + CH)**: Methods without maintainability smells and with robustness changes
- **Not Smelly + Not Changed (NoSML + NoCH):** Methods without maintainability smells and without robustness changes

With this statistical test, we can define whether robustness changes performed to a method are related to the presence of maintainability smells in that same method. To understand how these two factors could be correlated, we performed a manual inspection analysis of randomly selected 206 methods (equally distributed between the collaborators) with maintainability smells and robustness changes. First, we selected the methods in which robustness change and maintainability smell were present. Then, for each analysis, the participants filled out a form detailing how the robustness changes could be related to the maintainability smells. All participants have experience with exception handling and code smell detection and at least a Master's degree in Software Engineering. Whenever collaborators had even a slight doubt about the validity of a case, they took note of it, and another author was asked to confirm their findings. This process allowed a comprehensive review of each case, ensuring that mistakes were avoided and the verdicts were reliable.

Details about this manual inspection and the protocol used can be found in our replication package (2nd Complementary 2022).

Step 2: Verifying the introduction of maintainability smells. In this step, we complemented the analysis of RQ_1 with a qualitative analysis now looking at cases in which maintainability smells were introduced through robustness changes. For that purpose, we identified the methods by which this event occurred and inspected them, looking for why the robustness change introduced the smells. For this analysis, we prioritized cases with a higher quantity of smells introduced. Furthermore, we analyzed 30 methods, equally divided between the software systems. To select these methods, we divided them into quartiles, considering the number of smells introduced in that commit. Furthermore, four collaborators analyzed the methods presented in the 1^{st}, 2^{nd} and 3^{rd} (together), and 4^{th} quartiles.

Step 3: Assessing the impact of robustness changes on the degradation of methods: To answer RQ_2, we evaluate whether the number of maintainability smells in a class with methods that underwent robustness changes is significantly higher than the number of smells in classes that do not have methods with this kind of change. First, we compared the number of smells in both groups of classes. Next, we considered the total number of maintainability smells a class has in the end release, considering the smells on the method-level. Finally, to analyze the statistical significance of our results, we applied the Mann-Whitney test (Sheskin 2003).

Step 4: Identifying robustness smells as a symptom of design problems. To answer RQ_3, we analyzed the methods that underwent robustness changes between the pair of releases defined (see Section 4.4.2), resulting in 4,758 methods. First, we collected the robustness smells for each method and verified whether the robustness smells co-occurred with the patterns for identifying design problems. For each smell, we verified how many times a pattern co-occurred with the robustness change. Finally, we analyzed 80 cases in which this event occurred to understand whether this robustness change and the pattern could be related to a possible design problem.

4.5 Analysis and Results

This section presents and discusses the results to answer our research questions. First, we explored the correlation between robustness changes and maintainability smells. Next, we explored the density of smells compared to classes with robustness changes and without this kind of change. Finally, we explored the type of robustness changes that lead to the presence of design

Table 4.3: The Relation of Maintainability Smell (S) and Robustness Changes (C) - ($p < 0,05$)

Software System	Odds Ratio	SML + CH	SML + NoCH	NoSML + CH	NoSML + NoCH
apm-agent-java	4.755	50	941	99	8,859
dubbo	6.995	219	1,956	385	24,053
elasticsearch-hadoop	2.377	33	139	338	3,384
fresco	5.243	25	35	916	6,724
netty	6.902	45	63	1,778	17,181
okhttp	3.351	32	75	658	5,168
RxJava	5.938	48	105	1,991	25,860
spring-boot	2.192	10	62	1,320	17,938
spring-framework	4.504	31	73	3,302	35,023
spring-security	2.458	50	234	782	8,996

Odds Ratio: Odds ratio identified through the Fisher test
SML + CH: # of methods with maint. smells and robust. changes
SML + NoCH: # of methods with maint. smells and without robust. changes
NoSML + CH: # of methods without maint. smells and with robust. changes
NoSML + NoCH: # of methods without maint. smells and robust. changes

problems.

4.5.1 How Often do Robustness Changes Co-occur with Maintainability Smells?

To answer $\mathbf{RQ}_1$, we analyzed if there is a correlation between robustness changes and maintainability smells. For that purpose, we first relied on Fisher's exact test (Fisher 1922) (see Section 4.4.1). Table 4.3 provides an overview of this analysis.

Chances of methods with robustness changes being affected by maintainability smells. Considering the commits between the two releases, we computed a correlation for the methods' robustness changes and maintainability smells. In our analysis, all systems had $p < 0.05$, meaning that there is an association between the occurrence of robustness change and maintainability smell in methods. The odds ratio assumes values from 0 to infinity. When the OR is greater than 1, it indicates that methods that underwent robustness changes will likely be affected by maintainability smells.

In this test, we used the robustness change as a predictor and maintainability smell as the outcome. For instance, for the system Dubbo, the Fisher test computed an odds ratio of 6.995. This means that the odds of a maintainability smell occurring are 6.995 higher in cases where the robustness change occurs than in cases where this change does not affect the method. Thus, this indicates a strong correlation between robustness change and maintainability smells. In this example, 26,613 methods were changed between the releases, thus considered in the analysis. From these methods, 604 had robust-

Table 4.4: The Relation of Specific Smells (S) and Robustness Changes (C). ($p < 0,05$)

Software System	Odds Ratio							
	Brain Method	Dispersed Coupling	Feature Envy	Intensive Coupling	Long Method	Message Chain	Long Parameter List	Shotgun Surgery
spring-security	NR	11.002	2.477	NR	6.291	NR	NR	11.206
apm-agent-java	NR	6.952	8.153	9.058	13.431	2.482	NR	NR
dubbo	10.330	14.171	8.198	10.191	9.305	3.168	1.936	10.330
elasticsearch-hadoop	NR	7.214	2.968	4.607	3.971	3.002	NR	8.032
fresco	NR	9.299	8.878	NR	7.205	NR	2.780	NR
netty	NR	13.033	6.126	9.930	14.293	NR	2.461	NR
okhttp	NR	3.862	5.933	9.743	9.059	2.929	NR	9.011
RxJava	NR	10.049	9.820	NR	11.272	NR	3.387	21.686
spring-boot	NR	NR	NR	NR	NR	NR	3.682	NR
spring-framework	NR	15.411	4.517	NR	5.447	3.425	4.124	NR

NR: Not statistically relevant

ness changes (219 with maintainability smells and 385 without them). Also, we discuss the cases in systems with the highest odds ratio. For instance, in Table 4.3, the lowest odds ratio is 2.192 in the spring-boot system; hence methods with robustness changes are at least twice as likely to have maintainability smells compared to methods without such changes. Therefore, developers should consider these changes when introducing or changing exceptional code.

> ***Finding 1.*** *When robustness changes are performed, developers should be aware of the maintainability smells introduced or already present in the method.*

To better understand which maintainability smells correlated with the robustness changes, we did an analysis specifically for each smell. Table 4.4 presents the results for this analysis. Each column represents a maintainability smell. Each cell represents the odds ratio for the specific smell. When we did not reach statistically relevant results, we filled the cell with 'NR' (Not statistically Relevant).

We can highlight three smells: *Feature Envy*, *Dispersed Coupling*, and *Long Method*. The results of these three smells were statistically significant in all software systems, except for *spring-boot*. Dispersed Coupling had an odds ratio higher than five in eight systems, while Feature Envy and Long Method had in six and eight systems, respectively. That means a strong correlation between the robustness change and the presence of these smells on methods. The smell Shotgun Surgery had an odds ratio of 21.686 on *RxJava*. However, the results for this smell were relevant only for five systems.

> ***Finding 2.*** *Methods that underwent robustness changes mostly co-occur with the maintainability smells Dispersed Coupling, Feature Envy, and Long Method.*

We decided to manually evaluate cases involving the three maintainability smells mentioned with the highest odds ratio. For that, we selected a sample of 206 methods. To select them, we chose the commit in which the class had methods affected by the maintainability smells and had the robustness change. On a manual analysis, we identified that in at least 74 (35.91%), the maintainability smell was directly related to the robustness change, meaning that either the robustness change introduced the smell or worsened the smell already present in the method. More details about this validation can be found in our replication package (2nd Complementary 2022). Besides these direct cases, we observed that this relation could also be indirect.

Indirect relation between robustness changes and maintainability smells. One example is case in the method `drainLoop` from *RxJava* (RxJava 2023). We identified that the *Feature Envy* and *Message Chain* smells were related to the operation performed, which was a parsing that resulted in new exceptions being raised. Therefore, there is an indirect relation between the smells and the exceptional change. The indirect relation can also be related to *Long Methods*. In this scenario, it is natural that the excess of code statements will be more complex, leading to multiple catch implementations of generic catch blocks or even empty catch blocks.

Developers can use the logging mechanism as a source of information for minor exception-handling improvements. Changes in the logging regarding the errors handled were also constant in our analysis. We observed that 37 (from 206) cases were related to logging in the catch block. At first sight, this may be seen as a bad practice, as apparently nothing is handled. However, the log messages can serve as documentation for the developers, since they can provide information to the developers that will maintain this exceptional code. Furthermore, when a developer inserts a log message with proper information regarding the exception, it can indicate that he intends to improve the code in the future. However, in our analysis, we observed that the inclusion of *Feature Envy* was mainly related to changes in the log messages, as we described above on a method from *dubbo*. Hence, developers need to be careful when adding these messages since they can add new maintainability smells. For instance, the developer should give more context in the log message, which can be reinforced using more specific exceptions rather than generic ones. In addition, the developer needs to be sure about the log level for the messages, which would avoid smells such as the *Feature Envy*.

Maintainability smells can be included during robustness changes. We identified that from the 4,758 methods with robustness changes, 774 (16.26%) co-occurred with the introduction of maintainability smells.

Therefore, we analyzed the cases in which this happened. We focused on cases with the introduction of *Feature Envy*, *Dispersed Coupling*, and *Long Method*, which had statistically relevant results in 9 out of 10 systems (see Table 4.4). On *dubbo*, we observed a case in which the developer added log messages to the catch block. However, these messages heavily relied on calls to methods from other classes, adding a *Feature Envy* and a *Dispersed Coupling*. We observed a similar case on *RxJava*, but in which the catch block introduced the smell *Dispersed Coupling*. In addition, the block called multiple classes to close the resources used. On *okhttp*, we identified a case of bad practice on exception handling that led to the introduction of a *Long Method*. In this class, the developer added multiple catch blocks for different exceptions. However, the handling code was the same in every case, which led to duplicated code, making the method unnecessarily long. In that scenario, the developer should use the same code on the same block, since the Java language allows the developer to do that. These changes also can impact the robustness of the software system in the future, making it difficult for the developer to understand the source of that problem.

> ***Finding 3.*** *Robustness changes can introduce smells such as Feature Envy and Dispersed Coupling on the methods, which can negatively impact their maintainability.*

We observed that these cases in which the robustness changes come along with the introduction of maintainability smells happened when the method was introduced in the commit. Therefore, both the change and the smell are inserted simultaneously. Still, the robustness change can indicate the presence of these smells in the normal code. For instance, let us consider the method `onBeforeExecute` from *apm-agen-java* (APM Agent Java 2023). This method is part of the *ElasticsearchRestClientInstrumentation* class, which provides the instrumentation for the Elasticsearch Java Rest Client. The method is called before a request is executed using this client. In this case, the catch block in the method was empty, and the developer left a comment warning that nothing should be handled there. Through this message, it can be the case that the exception is not in the correct class. This can be directly related to the presence of the *Feature Envy*, indicated through the multiple calls to data from the `Span` class. This excessive calling also introduced the smells *Intensive Coupling*, *Long Method*, and *Message Chain*. Therefore, even though it was a simple change in the catch block, it signaled the other maintainability smells.

> ***Finding 4.*** *Even small changes in the exceptional code (e.g., a comment in an empty catch block) can be an indicator of maintainability smells such as Feature Envy and Intensive Coupling.*

In summary, in our first RQ, we observed the maintainability smells that commonly co-occur or are introduced with robustness changes performed in the code. Hence, developers should be aware of these smells when performing this kind of change on a method, even when small. This way, it is recommended that the developers plan ahead the changes that will be performed. Furthermore, these changes should be considered even in the early stages of software development, when the decisions regarding exception handling should be defined appropriately. Developers could also benefit from using tools that could warn them about the possible maintainability smells that could be introduced with that robustness change. Alternatively, even warning them about possible smells already present in that method, giving the change performed, is informative.

4.5.2 What Impact Can Robustness Changes Have on the Degradation of Classes?

To answer $\mathbf{RQ}_2$, we computed the density of maintainability smells in methods with and without robustness changes (see Section 4.4.4. Figure 4.4 presents box plots representing the density of smells per method group and software system. Inside each one, we have white dots representing the mean density. In parentheses, we show the systems in which $p - value < 0.05$, meaning that the results were statistically significant for the Mann-Whitney statistical test applied. With this test, we want to confirm whether the density of maintainability smells in classes with methods that underwent robustness changes is greater than the density of smells in classes without these methods.

Density of maintainability smells in classes with and without methods that underwent robustness changes. Looking at the box plots, we can observe that the density median was higher in six systems, while the median was the same in the other four. In addition, the mean density of smells is higher in seven systems. Thus, we can observe through the mean and median that classes with methods that underwent robustness changes tend to have a higher density of maintainability smells than classes without these methods. More details about the statistical values are available at Appendix B.2.

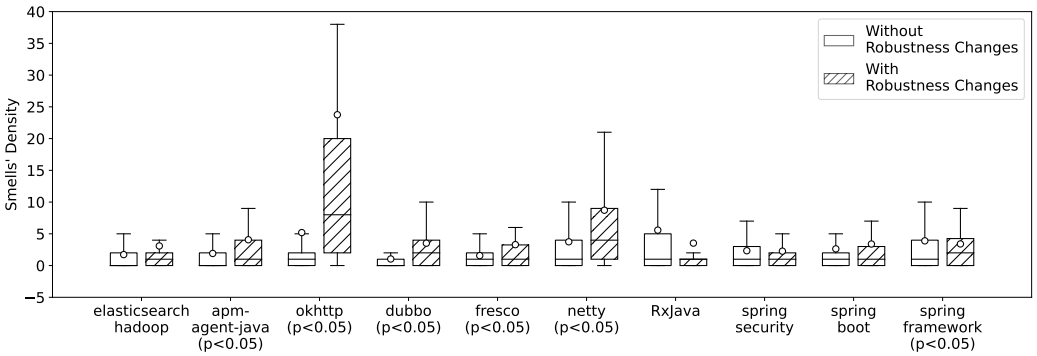

Figure 4.4: Smells Density in Classes per Software System.

> ***Finding 5.*** *Classes with methods that underwent robustness changes tend to have a higher density of maintainability smells compared to classes without methods with this kind of change.*

Robustness changes can worsen the already existing maintainability smells. In our analysis, we observed a typical scenario for all systems. The introduced catch block does not originate the smells but, in some cases, contributes to their worsening (*e.g.*, making a method more coupled than earlier). It suggests that the robustness changes occur in methods with a high density of smells. For instance, we identified that catch blocks contribute to amplifying the smelly nature of *Long Methods*, *Message Chains*, and *Dispersed Couplings*. Therefore, the catch block contributes to expanding the smelly structure since the same practices of the smelling code are – to a minor or a large extent – also replicated for the exception handling code.

Well-written robustness changes may reduce maintainability smells. We observed that methods with a high number of smells had catch blocks with poor or no exception handling at all. For instance, on *elasticsearch*, we observed that the excess of maintainability smells on the methods could be related to the lack of proper error handling. Multiple instances of Feature Envy were found in methods without exception handling. These methods exhibited this code smell more frequently than the methods with catch blocks in the same class. We also observed similar cases on *Fresco* and *dubbo*. Since we are considering the system's version on the last release (*e.g.*, the last version of the class), we can conjecture that the methods in which the developers did not handle errors degraded more when compared with the classes that had proper error handling.

Table 4.5: Cases in which Robustness Smells Co-occur with Pattern of Maintainability Smells

Pattern	Robustness Smell								
	Catch Generic Exception	Method Throws Exception	Empty Catch Block	Catch NPE	Rethrows Exception	Throw New Instance Of Same Exception	Throw Exception In Finally	Exception As Flow Control	Throw NPE
Concern Overload	212	1	54	2	7	0	3	7	3
Cyclic Dependency	74	5	6	0	6	0	3	7	1
Fat Interface	171	2	6	0	1	0	0	1	1
Misplaced Concern	394	5	199	3	8	1	0	1	7
Scattered Concern	35	0	1	0	0	0	0	1	0
Unwanted Dependency	274	2	81	2	7	0	0	1	4
Total	1160	15	347	7	29	1	6	18	16

> ***Finding 6.*** *Robustness changes can worsen maintainability smells such as Long Methods, Message Chains, Dispersed Couplings, and Feature Envy, hampering the modularity of the system.*

To sum up, with this RQ, we observed that the robustness changes could have a negative impact on classes with methods that underwent such changes. This impact is signaled through the high density of maintainability smells. Developers commonly use this density to prioritize classes likely to have design problems (Oliveira *et al.* 2019, Sousa *et al.* 2018). Furthermore, the introduction and worsening of smells were mainly related to the maintainability smells that signal design problems related to the system's modularity (Table 4.1). Hence, we highlight that when performing robustness changes, developers should be aware of the smells introduced or worsened since it may indicate deeper problems in the system. In addition, tools could use these metrics (*i.e.*, presence of robustness change and high density of specific smells) to signal possible design problems in the system.

4.5.3 How do Robustness Smells Give Evidence of Design Problems?

To answer $\mathbf{RQ}_3$, we analyzed if robustness smells co-occurring with patterns of maintainability smells could help in the identification of design problems. Thus, we first identified when the robustness smells appeared in methods together with the maintainability smells that indicate design problems (see Table 4.1). Table 4.5 presents the results. Each cell represents cases where the robustness smell co-occurs with the maintainability smells, forming a pattern. We can observe a high number of cases where *catch generic exception* (1,160 cases) and *empty catch block* (347 cases) co-occurred with the maintainability smells forming the patterns. We manually analyzed the patterns that happened the most with the robustness smells. They are highlighted in gray.

Generic and empty catches can indicate the presence of maintainability smell patterns. The robustness smell was not directly related to a design problem from the cases we analyzed. However, they could signal that maintainability smells were affecting the method. Let us consider the case of the method `getAsync` from fresco (Fresco 2023). This method has nested try/catch blocks due to excess of verification performed. There are also catch generic exceptions that only return `Null` instead of handling the exceptions. We can hypothesize that this happened for two reasons: (i) the method was fulfilled with multiple concerns; hence the developer did not know which exception should be handled, or (ii) the exception was handled on the wrong class. This can hint that this method could not be in the correct class. When we look at the maintainability smells, this method has a *Feature Envy* since it calls multiple methods from other classes, which also causes a *Dispersed Coupling*. Since the method uses data from other classes, it also causes a *Long Method*. Together, these smells can indicate the design problems *Misplaced Concern* or *Scattered Concern*. Both design problems suggest that this method should be moved to a more appropriate class.

Developers can be induced to introduce generic catches. Let us consider a case analyzed on *dubbo* (Dubbo 2023) in the method `getProxy`, which was affected by a *catch generic exception* (*i.e.*, a *Throwable*). This is a bad practice since the exceptions should be adequately handled depending on their type. As the scope of *Throwable* is too broad, it may hide runtime issues that should be better handled. However, the developer may be induced to introduce these bad practices. This may occur on methods affected by a *Long Method*, combined with the fact that the developer does not know the software system's design well. Thus, he/she uses this generic handler to catch any exception.

Furthermore, this harmful practice can be the source of new maintainability smells, as we observed in this method, which was affected by *Dispersed Coupling*, *Feature Envy*, *Message Chain* and was part of a *Complex Class*. This happens because the developer tries to implement multiple concerns in this method. To do that, he/she may need to call multiple methods from other classes, hampering the modularity of the software system. In addition, the long methods also tend to have more dependencies on external elements. Thus, the class also has a higher chance of having exceptions handling errors from multiple contexts, which can explain why developers use generic catches. In this example, both maintainability and robustness smells could indicate the presence of a *Misplaced Concern* or *Scattered Concern*.

Generic catches indicating unwanted dependencies. We observed

that developers tended to determine that all exceptions without a specific type would only be logged and lead to the system crash afterward. This happened when they were handling generic exceptions or in cases with an empty catch block, where only a comment was left in the handler. These smells could appear due to some projects' status as frameworks (*e.g.*, spring-framework and RxJava), meaning that the hot spot classes (*i.e.*, those that the framework's users will directly use) might have a different form of exception handling than the rest of the project. The design problem *Unwanted Dependency* seemed to appear for the same reason: the role of the classes as hot spots. Upon analysis, we noticed that these classes served as "import hubs". Each class imported several other package classes, allowing the user to have all the package's functionality by importing a single class. However, this bad practice leads to potentially unwanted dependencies among the classes. In addition, this caused the presence of the maintainability smells *Feature Envy*, *Shotgun Surgery*, and *Long Method*.

> ***Finding 7.*** *The robustness smells empty catch block, and catch generic exception can indicate the presence of maintainability smells mainly related to the design problems Concern Overload, Misplaced Concern, and Unwanted Dependency.*

In conclusion, when the code is too complex and affected by code smells on both normal and exceptional code, we observed that developers tend to modify only the normal code (the one inside the try block). This can happen since the exceptional code tends to be more complex due to the global nature of exceptions. This complexity is related to how the exceptions can traffic between system modules. Thus, when changing the exceptional code, the developer needs to be careful about how this change will impact the modularity of the software system. In addition to the complexity, the IDEs tend to provide refactoring suggestions only for the normal code. Due to its different nature, it would be ideal to have specific refactorings for the exceptional code. For instance, in some cases, the IDEs constrain the implementation of exception-handling mechanisms. The IDE does not let the developer choose which class should handle the exception. Therefore, the developer must be free to choose how to properly handle the exception before applying the refactoring. Hence, the refactoring needs to be manually performed, which can discourage the developer from implementing the improvement change.

We hypothesize that this behavior was also due to the need for proper refactoring recommendations by IDEs. These tools provide better refactoring

options only for the normal code. They do not consider the nuances the exception handling code can have, such as which class could or should handle the exceptions. Thus, these changes co-occurring with the code smells can indicate to developers that robustness refactoring should be done at that time. When neglected, these robustness changes can also increase the degradation of the method. This can complicate the code in the future, making it more complex and discouraging developers from maintaining the code.

As observed, the robustness smells can indicate the presence of patterns of maintainability smells that signal design problems related to the system modularity. These design problems are challenging to identify since they can be scattered through multiple software system components. By analyzing maintainability smells, developers may not be able to identify a design problem accurately. However, developers can find more reliable results when using both maintainability and robustness smells. Thus, the indicator given by the co-occurrence of robustness smells and these patterns can help the developer to identify this design problem and refactor the code to remove or reduce this problem. Hence, the developer will save time and effort on this identification.

4.6 Threats to Validity

Our analyses were performed on a set of 10 software systems, which could be a threat. However, our selection was based on meticulously defined criteria to find software systems (see Section 4.4.2) relevant to our research questions. In addition, our criteria also focused on reproducibility, allowing other researchers to replicate our steps. Therefore, our study can be the starting point for other researchers to explore the types of relationships explored in this paper with other software systems with other purposes and domains.

Another threat is the detection strategies (and their specific thresholds) for code smells. To mitigate this threat, we selected a tool (Organic) that uses a set of strategies and thresholds commonly used by the software engineering community (Lanza and Marinescu 2006). Moreover, this tool has been successfully used in multiple studies about software design (*e.g.*, (Oizumi *et al.* 2016, Cedrim *et al.* 2017, Sousa *et al.* 2018, Oliveira *et al.* 2022)). Finally, we mitigated this threat by performing a manual inspection analysis with experienced experts. For each analysis, the participants filled out a form detailing how the robustness changes could be related to the maintainability smells.

The selection of releases can also be another threat. However, we carefully selected these releases based on the number of commits that the two releases had between them. More specifically, we selected a pair of releases spanning at

least 1000 commits. That allowed us to cover a significant part of the history of software systems. Furthermore, there is no bias in the selection of releases, since this metric of commits was used indistinctly from the software system.

The use of exception handlers affects the size and complexity of methods, potentially threatening validity. Therefore, we validated our dataset to ensure we were not performing unfair comparisons between methods with and without exception handlers. Hence, we observed that the presence of getters/setters was balanced between these two subsets per software system, accounting for ∼11% of getters and setters composing our complete dataset.

4.7 Conclusion

We explored how the combination of poor robustness changes and maintainability smells indicate design problems. For that purpose, we analyzed over 160k methods from 10 open-source systems. We identified that the robustness changes could introduce or worsen maintainability smells such as *Dispersed Coupling*, *Feature Envy*, and *Long Method*. We also observed that classes with methods that underwent robustness changes tend to have a higher density of maintainability smells compared to classes without these methods. Finally, we observed that the robustness smells *empty catch block* and *catch generic exception* could be used with the maintainability smells to help the identification of design problems such as *Concern Overload* and *Unwanted Dependency*.

Understanding the relationship between robustness changes and maintainability code smells can help developers identify design problems' presence. Such problems are harmful to the software and hard to identify. Using the knowledge about this relationship can be the first step toward their identification. Additionally, these problems can be the target of refactoring operations, thus improving the systems' maintainability and robustness. Developers can now be aware that even small changes in the catch blocks can impact or indicate possible design problems in the system. Moreover, based on our findings, tools can be developed to better assist the maintainability and evolvability of systems.

In future work, we plan to explore other symptoms of design problems (*e.g.*, smells related to other NFRs). In addition, we will investigate the evolution of methods (*i.e.*, further changes in later versions) that once underwent robustness changes. Thus, we will be able to understand how poor robustness changes can further impact the quality of the method along with its evolutionary changes. Finally, we plan to explore cases where only the exception

part of the method has changed and observe its impact on the normal code unchanged.

5
Understanding How Developers Deal With Non-Functional Requirements

In the two previous chapters, we explored how developers may use and combine two symptoms of design problems: maintainability smells and robustness smells (see Chapters 3 and 4). Another possible symptom of design problems is the non-conformity with NFRs (see Section 2.3.3). For instance, observing that a system is becoming more difficult to change over time, indicates non-conformity with the system's maintainability. Hence, the stakeholders may consider that a possible design problem is causing this high effort to maintain the system. However, as presented in the earlier chapters, identifying design problems is not a trivial task, especially for developers with less experience.

Developers who understand how to properly address NFR concerns within the system are likely to prevent the system from facing design problems. Hence, understanding who these developers are and how they deal with NFRs can be a key step in identifying and avoiding design problems. In addition to understanding who these developers are, we also need to understand how they discuss and perceive NFRs within their systems. Understanding how experienced developers deal with NFRs can give us insights into how experienced developers can share their knowledge with less-experienced developers on NFR tasks.

Despite its importance, the discussion of NFRs is often ad-hoc and scattered through multiple sources, limiting developers' awareness of NFRs. Pull Request (PR) discussions on open-source systems are a common mechanism used by developers for mentioning design and requirements concerns, thus providing a centralized platform for comprehensive NFR knowledge exchange.

In this study, we report an investigation on how developers discuss and perceive NFRs within their systems. First, we investigated NFR discussions available in PRs of repositories of the Spring ecosystem. We collected, manually curated, and analyzed 1,533 PR discussions addressing four categories of NFR: maintainability, security, performance, and robustness. Through this characterization, we identified and investigated the most engaged developers in NFR discussions.

To complement this analysis, we applied a survey with 44 developers to gather their perceptions on NFRs discussions. In addition, we also considered developers from closed-source systems, looking for similarities and differences between the developers' strategies on open-source systems. By observing how developers approach NFRs and participate in discussions, we documented the best practices and strategies newcomers can use to address NFRs effectively.

This chapter presents the paper *Understanding Developers' Discussions and Perceptions on Non-Functional Requirements: The Case of the Spring Ecosystem*. The study addresses the third major contribution of this Ph.D. thesis: *how do developers discuss and perceive NFRs within their systems* (see Section 1.4).

5.1 Introduction

Non-Functional Requirements (NFRs) determine desired qualities or attributes a software system should have, such as security, maintainability, and performance NFRs. The specification of NFRs on software development is essential to establish the technical constraints in which software systems should run (Casamayor *et al.* 2010). Consequently, NFRs support developers in making architectural and design decisions that will drive the implementation of a software system (Bashar 2001).

Over the software life cycle, developers are expected to discuss and document the impact of NFRs on the system. Particularly during software maintenance activities, the lack of organized and updated information concerning NFRs has several negative consequences (Slankas and Willians 2013, Saadatmand *et al.* 2012). For example, not having a proper source of information developers rely on when reasoning about NFRs can result in misinterpretation of the maintenance tasks, delays in solving issues, and increased risk of introducing new NFR problems (*e.g.*, decreasing performance) (Yamashita and Moonen 2012, Yamashita and Moonen 2013, Chung and Nixon 1995).

Despite its relevance, the specification of NFRs is commonly neglected or informal (Bhowmik *et al.* 2019, Chung *et al.* 2012, Hoskinson 2011, Scacchi 2009). We observe this behavior in open-source systems (OSSs), where developers typically discuss NFRs on demand during maintenance tasks, using unstructured communication such as mailing lists (Noll and Liu 2010, de Souza *et al.* 2005). Since the information about the system NFRs is unavailable or non-organized in a structured and specific/dedicated document, developers need to find alternatives to map and understand the systems' NFRs, such as analyzing scattered textual content (Noll and Liu 2010).

A few studies have investigated alternatives for automating the identification of NFRs on available documentation (Slankas and Willians 2013, Kurtanović and Maalej 2017, Casamayor *et al.* 2010, Cleland-Huang *et al.* 2007). These investigations explore detection approaches ranging from keyword strategies to Natural Language Processing through Machine Learning. However, automatically detected NRFs are typically limited to small datasets of requirement specification documents (*e.g.*, the PROMISE dataset (Baker *et al.* 2019, Chatterjee *et al.* 2021, Kaur 2022, Jindal 2021)). Furthermore, a set of studies use closed-source systems (Mohammed and Alemneh 2021, Chatterjee *et al.* 2021, Handa *et al.* 2022, Wang *et al.* 2018, Tóth, 2019), and do not make their artifact available for reproducibility. Yet, to the best of our knowledge, no previous work explored the identification of NFRs in PR discussions.

Pull Requests (PRs) are a common resource employed by developers for discussing the need for new system features and software maintenance and evolution (Soares *et al.* 2015, Gousios *et al.* 2014, Gousios *et al.* 2015, Jiang *et al.* 2021). A recent study (Jiang *et al.* 2021) on 900 GitHub projects reveals that more than 54% of projects produce their release notes with PRs (*e.g.*, details of recent changes). Given the nature of PR discussions, their content can be a valuable source of information on NFRs. However, first, one needs to identify which PRs contain NFR discussions and characterize such types of discussions. This characterization aims to understand how the NFR discussions happen and how the developers are involved. Therefore, in this paper, we report an empirical study on characterizing NFRs that emerged from PR discussions and understanding the characteristics of developers engaged in these discussions. This characterization lets us gather insights into how the developers discuss the NFRs within the Spring ecosystem and can show us relevant features when building NFR classification mechanisms.

For the purpose of analyzing NFR discussion, we developed our own dataset, where we manually classified 1,533 PRs obtained from three Open-Source Systems (OSSs) within the Spring ecosystem (Spring, 2022). We selected the Spring ecosystem due to the diversity of functionalities and services delivered (Cosmina *et al.* 2017). Besides, the Spring community is notably active in discussing code changes through PR discussions. We focus on four NFR types, namely *maintainability*, *robustness*, *performance*, and *security*. The NFRs chosen were the ones more prominent in the systems analyzed.

To select the sample of PRs to be manually classified, we applied a pre-classification using keywords linked to NFRs by previous studies (Cleland-Huang *et al.* 2007, Slankas and Willians 2013). Then, we followed a well-structured procedure for the manual classification of each PR. We classify PR

discussions in terms of *(i)* the presence of the NFR type addressed, *(ii)* the location in the PR where the discussions about NFR are triggered, *(iii)* the keywords mentioned in the discussion, and *(iv)* the main message addressing the NFR. This classification allowed us to characterize the PR discussions and the developers discussing NFRs.

Through the analysis of the dataset created, we identified 63 developers that frequently engaged in PRs addressing NFRs. After investigating their characteristics and participation in the discussions, we found that most of these developers play roles related to NFRs (*e.g.*, Security Engineer Senior) in their companies, indicating their expertise in certain NFRs types. This suggests that these NFR discussions can attract attention from experienced developers within the OSS community. In addition, these developers are commonly involved with key tasks in the software system (*e.g.*, commits and reviews). Compared to other developers, their participation in the NFR discussions regarding these tasks is also prevalent. By analyzing the PR discussions classified, we found that the discussions about NFRs typically are triggered by the PR title or description (77%). This shows developers opening these discussions are already concerned with the system NFRs.

To better understand how developers perceive and address NFRs within their systems, we ran a survey with 44 developers. Through this survey, we identified that developers engage in NFR discussions, since they acknowledge their significant influence on software quality. We also observed how the developers address NFRs throughout the entire software development lifecycle, employing multiple methods, including PR discussions and rigorous testing, to ensure the software complies with the systems' NFRs. The participants also highlighted the collaborative aspect of dealing with NFRs during the software evolution, going from meetings between stakeholders to suggestions made by more experienced developers and architects.

We make the following four achievements:

1. A characterization of how four common types of NFRs are discussed in PRs. This characterization gives the first glimpse into the nature of NFR discussions in the OSS community. It can be further employed to support the building of new technologies for the automatic classification of NFR discussions and warning developers in charge of dealing with NFRs.

2. We highlight prevalent developers' characteristics while discussing NFRs. By identifying these characteristics, we can support project managers in allocating team members who ideally are more qualified to solve an NFR problem (Joblin *et al.* 2017). In addition, by understanding

which developers are involved in NFR discussions, newcomers can more effectively collaborate with those developers and work together to address NFRs.

3. A set of strategies on how to address NFRs. Through the survey applied, we gathered the perceptions of developers regarding NFRs and how they address such requirements in their systems. This information enabled us to formulate an initial set of strategies aimed at assisting developers in effectively addressing NFRs. These strategies include the use of specific technologies and practices (*e.g.*, continuous integration) allied with testing and the use of benchmarks to guarantee the NFRs defined for the system meet the expectations.

4. A curated dataset composed of 1,533 PRs addressing *maintainability*, *robustness*, *performance*, and *security*, which is publicly available[1]. Details about the dataset is available at C.1. We also provide a subset of common keywords related to NFRs derived from manual analysis. The full list of keywords is available at Appendix C.2. This content improves previous studies (Cleland-Huang *et al.* 2007, Slankas and Willians 2013) and supports further investigations on conceiving, training and validating NFR classification technologies.

Audience. Project managers shall benefit from our analysis by identifying developers more frequently involved with NFRs. We also provide a documented set of strategies to better address NFRs in software systems based on knowledge gathered through our survey. In addition, newcomers can learn effective strategies and best practices for addressing NFRs by observing experienced developers' approaches and participating in discussions. Moreover, although our analysis is focused on the systems from the Spring ecosystem, the insights gained from our findings can be explored in practice for other ecosystems.

5.2 Background and Related Work

In this section, we explore *(i)* documentation of NFRs, *(ii)* discussions on OSSs, and *(iii)* developers perceptions on NFRs.

[1]Available at `https://github.com/devs-discussions-perceptions/devs_discussions_perceptions_paper`.

5.2.1 The Documentation Gap of NFRs

Based on requirements specification, developers are expected to manage and validate the implementation of NFRs through the software development life cycle. However, requirement specification documents usually focus on functional requirements, lacking a sufficient description of the system's NFRs (Cleland-Huang *et al.* 2007). In the case of OSSs, even producing a formal specification of the functional requirements is unusual (Bhowmik *et al.* 2019). Instead, content addressing the system requirements is often scattered through many documents related to the software system, such as mailing lists, message boards, and ad-hoc discussions (Noll and Liu 2010, Slankas and Willians 2013). The lack of proper specification and documentation of the system's NFRs is an obstacle to its maintenance and evolution (Robiolo *et al.* 2019). For instance, newcomers in charge of evolving systems that lack documentation about NFRs will likely spend considerable time identifying/inferring them. Lack of information may result in an insufficient understanding of the system's NFRs, which may cause design problems (Sousa *et al.* 2018).

To overcome the problem of insufficient NFR documentation, studies explored different approaches to automatically identify NFRs on the available textual artifacts of the system. Binkhonain and Zao (Binkhonain and Zhao 2019) performed a review on machine learning techniques used for the identification and classification of NFRs. In their reviews, the authors observed a series of limitations on the existing techniques, such as the lack of (i) labeled datasets, (ii) standard definition, classification, and representation of NFRs, and (iii) reporting standards.

Studies have investigated the use of Natural Language Processing (NLP) combined with Machine Learning to automate the identification and classification of NFRs on specification documents. Casamayor *et al.* proposed a semi-supervised learning approach, combining the algorithm Expectation Maximization with Naive Bayes (Casamayor *et al.* 2010). The authors empirically evaluated their approach over the PROMISE dataset. The proposed semi-supervised learning approach reached an accuracy rate higher than 75%, surpassing 85% for some NFRs. Slankas and Willians techniques, such as k-NN, Multinomial Naïve Bayes, and Sequential Minimal Optimization (SMO) to identify NFRs from the PROMISE dataset combined with the available documentation (Slankas and Willians 2013). SMO reached 74% precision and 54.4% recall. However, due to the nature of the additional documents composing the dataset, most of the NFRs detected address only legal requirements.

The aforementioned works rely on the PROMISE dataset. However, this

dataset comprises only 625 sentences describing requirements specifications (255 functional and 370 non-functional), which may be considered a limited learning scope. Furthermore, this dataset is highly unbalanced. Considering the four NFRs we explore in the OSSs, it has 74 instances of maintainability, 24 of performance, 25 of security, and none of robustness. Therefore, in our study, we could not rely on state-of-the-art techniques to identify the discussions automatically. After creating a model based on the PROMISE dataset, we reached accuracies below 40%. That led us to create our own dataset for the purpose of our analysis. We detail this dataset in Section 5.3.2.

5.2.2 Open-Source Discussions

OSSs using distributed version control systems tend to use features that support developers in tracking and discussing changes (Gousios *et al.* 2014). For example, PRs are one of the most used features in GitHub (Github 2022). PRs allow developers to communicate with other team members about changes to be performed (Github Pull Requests 2023). Once a developer opens a PR, it may discuss the acceptance/rejection of change with other team members. A PR discussion is composed of *(i)* a title, *(ii)* a description, and *(iii)* comments (Liu *et al.* 2019). The PR author creates both the PR title and description. The PR comments result from the system team members' discussion.

Software system discussions represent an important and abundant resource available in OSSs. These discussions may run in the context of reported issues and PRs. Our study focuses on PR discussions once they often support requirement and design discussions among developers (Gousios *et al.* 2014). Through investigation of software system discussions, we can identify the developers who consistently address NFRs within these discussions. By observing their characteristics, we can have an understanding of how the team addresses NFRs during the software evolution. In addition, since these discussions often include the decisions taken to address the NFR, they can serve as documentation to help future developers understand how to properly handle such requirements within the systems.

5.2.3 Developers' Perception on NFRs

A few studies explore the developers' perception of NFRs (Zou *et al.* 2017, Ameller *et al.* 2012, Camacho *et al.* 2016). However, these studies lack characterizing how developers discuss NFRs. Zou *et al.* investigated the

perspective of developers on NFRs by exploring 21.7 million posts and 32.5 million comments on Stack Overflow (Zou *et al.* 2017). For this purpose, the authors applied topic modeling to identify the topics related to NFR discussed in the posts and comments. The authors identified that developers focus more on usability and reliability and are less concerned about efficiency and maintainability. They also investigated the difficulty of a topic to be discussed. The authors found that maintainability is the most difficult NFR type for developers to discuss.

Ameller *et al.* investigated how software architects perceive and address NFRs in their projects (Ameller *et al.* 2012). The authors conducted a semi-structured interview with 20 software architects from academia and industry. They found that the software architects perceive NFRs as key elements for the success of software development. The authors also identified factors that impact how to address the NFRs, such as project size, team composition, and stakeholder involvement.

Camacho *et al.* investigated the perceptions of agile team members regarding the importance of testing NFRs (Camacho *et al.* 2016). The authors applied semi-structured interviews with 20 participants from a single company. They observed that several factors, including experience, culture, and awareness, influence the identification of NFRs' testing needs. The authors provide recommendations to address this kind of test better. For instance, code review practices and all roles within the team should work with quality in mind, being aware of the NFRs.

Despite their contributions, these studies have limitations. Zou *et al.* do not investigate the reasons behind the developer's perceptions and focus on questions from Stack Overflow. Hence, it is not clear the experience of the developers involved in the study. Although Ameller *et al.* focus only on software architects, they miss other stakeholders involved in the project. In their study, Camacho *et al.* explore stakeholders of a single company. Our study surveyed more than 40 developers in different positions within their teams and from different companies. In addition, we investigate the different reasons that lead developers to address the NFRs.

Characterizing the developers who deal with NFRs includes identifying, among others, the type of development tasks in which they are more frequently engaged in OSSs, which may indicate their interests and skills. Gonzalez *et al.* investigated the developers' profile based on their productivity (*e.g.*, number of commits) and code quality (*e.g.*, number of refactorings) (González *et al.* 2021). They evaluated 77,932 commits from 33 OSSs, clustering 2,460 developers using the k-means algorithm. The authors identified three profiles of

developers: *(i)* the cleaner (who documents and fixes issues), *(ii)* the average developer, and *(iii)* the dirty (who introduces complex functions that often need to be refactored). Even though this study followed a more qualitative approach to identifying developer profiles, the authors did not explore whether the developers' profiles deal with NFRs.

5.3 Study Design

To understand PR discussions related to NFRs, we first manually curated a dataset with such discussions. We classified 1,533 PR candidates to address NFRs. Through this classification, we built a dataset to characterize the NFR discussions and identify the developers who discuss NFRs. We selected 63 developers to investigate their characteristics and strategies for dealing with NFRs. Finally, we conducted a survey with developers from private companies to compare their perceptions with the results found in the PRs analysis. The following subsections present the study settings.

5.3.1 Research Questions

With this study, we have the goal of *understanding developers' discussions and perceptions on NFRs*. Thus, our study is guided by the following research questions (RQs).

RQ1. *What are the characteristics of the discussions in PRs addressing NFRs?*

To answer `RQ1`, we manually classified NFRs on the PR discussions of three systems from the Spring ecosystem. We focus on analyzing PRs designed to propose changes in the codebase. PRs typically involve several members with different roles, which may lead to a broader view of NFR discussions. For each PR, we first identified the locations (*i.e.*, PR title or PR description) where the contributors address NFRs. With this information, we intend to identify how NFR issues are typically reported in PRs, triggering the discussions. Then, we characterized the NFRs' discussions concerning the topics discussed. We aimed to determine whether it was possible to generalize NFR topics in these discussions. From PR discussions, we could identify the developers most engaged in discussing NFRs (see RQ2).

RQ2. *Who are the developers that engage in NFR discussions?*

By characterizing the developers' expertise and how they contribute to the projects, we expect to understand their role in NFR discussions. We intend

to point out the desired characteristics of developers that fit roles related to a specific NFR in a team. Once we understand the developers who mainly engaged in NFR discussions in OSSs, we now want to understand developers' perceptions of open-source and closed-source systems. For this purpose, we defined the following research question:

RQ3. *How do developers perceive and address NFRs in their daily work?*

To answer this RQ, we conducted an opinion survey (Linaker *et al.* 2015) with 44 developers working with multiple closed-source systems. The survey is composed of questions ranging from the early stages of the software system to its continuous maintenance. Based on the insights gathered, we intend to point out the best strategies that developers tend to use when addressing NFRs.

5.3.2 Study Steps

To answer our RQs, we performed the steps described and presented in Fig. 5.1.

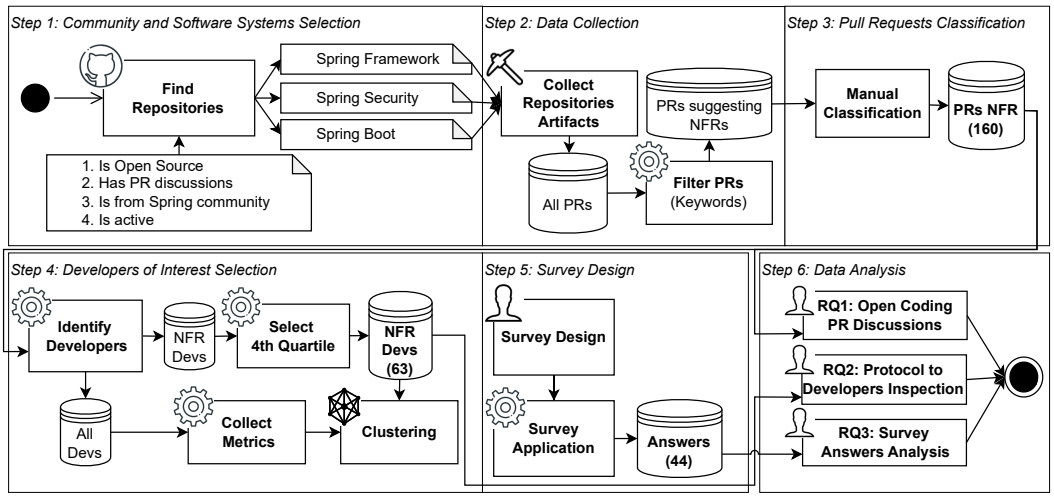

Figure 5.1: Workflow of our Study Design.

Step 1. Software Systems Selection: To select the systems within the Spring ecosystems that are suitable for this study, we followed the criteria adopted by Barbosa *et al.* (Barbosa *et al.* 2020): *(i)* systems that use PRs as a mechanism to discuss code changes, *(ii)* the system Git software system must have over 1,000 commits, *(iii)* systems that are at least five years old; and *(iv)* systems currently active. Notice that applying such selection criteria is essential for being able to perform reliable conclusions (Kalliamvakou *et al.* 2016). We selected the three that provided the diversity, considering the entire effort required for the analyses.

Table 5.1: Software Systems from the Spring Ecosystem

System	# Commits	# Pull Requests	Time Span
spring-boot	40,071	5,319	2012-2022
spring-framework	25,522	3,850	2008-2022
spring-security	11,671	1,954	2004-2022

We selected three systems, namely Spring Security, Spring Framework, and Spring Boot, from a wide and well-known open-source community available at Git: Spring (Spring Community 2022). Table 5.1 describes the characteristics of each system. We conducted this study over a particular ecosystem to allow us to investigate the dynamics of the NFRs discussions in more depth.

Step 2. Data Collection: We used the GitHub API (Github Rest API 2022) to collect the PR discussions. For each NFR type, we defined a set of keywords that can indicate its incidence in PRs. The set of keywords used in our study is based on previous works (Cleland-Huang *et al.* 2007, Slankas and Willians 2013). We collected the keywords on different PR discussion levels (*e.g.*, title, description, and discussion) and the commit messages related to each PR. With this set of keywords, we performed a pre-classification of PR discussions by the NFR type addressed. Based on this pre-classification, we randomly selected a subset of PRs for each NFR type to be manually inspected and classified by experts. The keywords considered in our study are found in Table 5.2.

Step 3. PRs Classification: First, we defined the NFRs to include in our dataset by identifying the most prominent NFRs within the systems through an analysis of the PR labels. This analysis highlighted four NFRs: *security*, *maintainability*, *robustness*, and *performance*. Since we wanted to characterize the discussions related to NFRs, one of the steps followed was to identify where the mentions of NFRs happen in PRs. Hence, we divided the PR discussion into *(i)* title, *(ii)* description (*i.e.*, first comment), *(iii)* discussion comments, and *(iv)* review comments. We manually classified PRs from each system to identify whether the discussion was related to an NFR. For each PR, the following information was identified:

- The NFR types identified (*i.e.*, security, maintainability, robustness, or performance).
- The sentence (or sentences) that led to each NFR identification.
- The keywords related to each NFR, from single words to short sentences.
- The location of each NFR keyword: title, description, comments, or review comment.

Table 5.2: Non-Functional Requirements and its Keywords

NFRs	**Keywords**
Maintainability	maintainability, maintenance, reliability, serviceability, accordance, measures, requirements, index, update, release, production, addition, budget, integration, operation, comprehension, readable, readability
Performance	performance, rate, bandwidth, cpu, time, latency, throughput, channel, instruction, response, process, communication, space, memory, storage, peak, compress, uncompress, runtime, perform, execute, dynamic, offset, reduce, response, longer, fast, slow, maximum, capacity, scale, cycle
Robustness	robust, robustness, inputs, error, failure, network, error, reliability, serviceability, fault, tolerance, exception, bug, recover, handl, fail with, crash, unexpect, NPE, null, stack, swallow, reliable
Security	access, author, ensure, data, authentication, security, secure, malicious, prevent, incorrect, harmful, state, exception, vulnerability, vulnerable, malicious, harmful, attack, expose, compromised, authenticator, encrypt, cookie, encrypted, ssl, authenticate, integrity, virus, encryption,

– The participants involved in the PR discussion.

Seven collaborators performed the manual classification individually. In cases of uncertainty about the classification, a second researcher was consulted. In case of disagreement, the collaborators discussed it until they reached a consensus. As the output, we have all the PRs tagged with the (set of) NFRs they are discussing. We also used the GitHub API to collect relevant metrics about the developers' participation in NFR discussions. The full list of metrics is presented at Appendixes C.3 and C.4.

Step 4. Developers of Interest Selection: First, we selected the most engaged developers in NFR discussions. Next, we distributed them into quartiles by the frequency of their participation in NFR discussions. Hence, we defined three participation levels: Low (1^{st} quartile), Average (2^{nd} and 3^{rd} quartiles) and High (4^{th} quartile). We selected the developers in the 4^{th} quartile as the developers of interest since they are the most participative in NFR discussions.

Aiming at also evaluating developers who engaged less frequently in discussions but were more focused on discussing NFRs, we selected another set of developers. For this purpose, we used the following metrics: *(i)* developers who mentioned at least one keyword related to an NFR in at least 25% of their messages within the repository, *(ii)* developers who actively participated in a minimum of three PR discussions, and *(iii)* developers who had less than 50 messages in the repository. Following these rules, we could find developers who are less active regarding discussions, but their discussions were more focused on NFRs.

To characterize the developers, we computed metrics addressing their background, activities, and code quality (*e.g.*, experience in years). We com-

puted the metrics by gathering raw data through the GitHub API and performing aggregations to compound more complex metrics. The list of metrics can be found in at Appendix C.3.

Next, we applied a clustering algorithm to group the developers based on their metric similarities. We ran a dimensionality reduction in the metrics using PCA (Abdi and Williams 2010). Then, we conducted the silhouette analysis (Rousseeuw 1987) to estimate the ideal number of clusters that fit our data. Finally, we applied the k-means (MacQueen 1967). For each software system, we obtained a set of clusters representing similar developers. We combined these steps to select developers of interest to be investigated and analyze their characteristics. We selected 15 developers from the fourth cluster and 48 from the other clusters.

Step 5. Survey Design: We designed a survey to answer RQ3. The survey population is composed of software developers with experience in dealing with NFRs in their systems. The survey is at Appendix C.6. The survey questionnaire consists of 19 questions, divided into four blocks, as follows:

1. Eight questions to characterize the developers regarding their age, gender, academic experience, experience in software development, role position, and proficiency in programming languages.

2. Eight questions aimed at gathering insights into the practices of the developers when dealing with NFRs.

3. Two multiple-choice questions, in which we wanted to understand the developers' strategies to address NFRs in their systems. The developers also had the opportunity to add new options if necessary. In addition, we asked them to complement their answer, explaining how these strategies were applied.

4. One open question designed to gather insights on the learning and improvement opportunities that the developers experienced when dealing with NFRs.

We conducted the opinion survey in August 2023, extending a 15-day window for responses. We used social media such as Twitter and LinkedIn to publicize the survey. We also publicized with other research groups that we had previously worked with. After this recruitment, the survey questionnaire was answered by 44 developers.

Step 6. Data Analysis: To answer `RQ1`, on the dataset created, we identified the parts of a PR discussion in which the NFRs appeared (*i.e.*,

title, description, messages, and/or review). To analyze the content of the discussions, we selected a sample of around 15% of the total number of PR discussions in each system, a total of 160 PR discussions for qualitative analysis. We ensured that the discussions selected were equally distributed for each NFR, avoiding bias, and ensured a representative sample for each NFR type. For this analysis, we followed some steps based on Grounded Theory procedures (Corbin and Strauss 2014). We applied the *open coding* procedure, which consists of *breakdown, analysis, and categorization* of the data. To develop the coding scheme, we identified the subcategories of each NFR and developed an initial set of codes based on these subcategories (see Section 5.4.1). Then, we refined the categories by applying the codes to the data and revising as needed. For cases in which the code did not meet any category, we created new ones. The four researchers involved in the open coding received a standard definition of each NFR type to support their activities. A second author reviewed each code and category created. In case of disagreement, the collaborators discussed it until they reached a consensus.

To answer `RQ2`, we investigated 63 engaged developers (Step 4). Since the profiles of these developers were not available in a specific document in the projects' repositories, we created a protocol to perform this analysis, in which four collaborators participated. This protocol includes *(i)* the analysis of the developers' GitHub profile, *(ii)* the identification of other repositories they collaborate, and *(iii)* the manual analysis of their profiles available on the Spring team page (Spring Team 2022). The entire protocol is available in our complementary material[1]. We followed the Grounded Theory procedure to analyze this information, as in `RQ1`, categorizing and grouping information to find clusters of developers with similar characteristics.

To answer `RQ3`, we examined the survey answers. We applied descriptive statistics to analyze the numerical and multi-option questions. We also used visualizations (*e.g.*, bar charts and segmented charts) to observe some tendencies in the results. To evaluate the open questions, we applied qualitative analysis, categorizing the answers to observe the topics most discussed by the developers. Five collaborators participated in this last analysis; at least two analyzed each answer. In case of disagreement in the categorization, a third author was asked to participate in a discussion until the team reached a consensus.

5.4 Results

In this section, we present the results of our study to characterize NFR discussions and understand how the developers perceive and address NFRs in

Table 5.3: Distribution of Mentions to the NFRs on Pull Requests Discussions

NFR	Title	Description	Messages	Review	Total
Maintainability	120 (59.11%)	71 (34.97%)	35 (17.24%)	0 (0%)	203
Security	91 (69.46%)	34 (25.95%)	26 (19.84%)	0 (0%)	131
Robustness	58 (51.78%)	45 (40.17%)	29 (25.89%)	2 (1.78%)	112
Performance	38 (58.46%)	32 (49.23%)	13 (20%)	1 (1.53%)	65

their systems.

5.4.1 What are the Characteristics of the Discussions in PRs Addressing NFRs?

To characterize the PR discussions related to NFRs, we first inspected where those discussions started and evolved within each PR. For that purpose, in our manual classification (see Section 5.3.2 - Step 3), we identified which part of the discussion mentioned the NFR keyword (*e.g.*, title, description, discussion messages, and/or review). Table 5.3 presents the number of times the NFR mention occurred on the PR location. Between parenthesis, we show the percentage of mentions compared to all mentions. The last column shows how frequently we observed the NFR in our dataset. For instance, we observed `Maintainability` in 203 PRs. In 120 (59.11%) of them, the mention occurred in the PR title. Notice that, in some cases, the mention appear in multiple locations (*e.g.*, title and description).

By analyzing Table 5.3, we observe that the PR *title* and the *description* mainly trigger discussions. For all four NFR types, their mention is on the *title* in more than 50% of the instances. When we look at the description, the percentage is lower, even reaching only 25.95% for `Security`. Considering that the *title* and *description* commonly contain information about NFRs, we conducted a more in-depth analysis of their utilization.

Complementary nature of *title* and descriptions. The *title* and *description*, combined, are the pieces of information that the developer must provide to open the PR. Therefore, we also considered when the keyword was in the PR *title* or *description*. We observed that the NFR with the lowest occurrence was `Robustness`, with 77.67% of cases with an NFR mentioned either on the title or *description*. `Maintainability`, `Security`, and `Performance`, reached 83.74%, 84.73%, and 86.15%, respectively. To better understand this relation, consider the example of a PR opened on Spring Framework (Spring Framework 2022). In the PR *title*, the developer shortly described the change performed:

"Avoid unnecessary sorting overhead"

From the mention of the term "overhead", this PR seems related to a change in `Performance`. However, the change is still unclear, in the change description, the developer states:

> *"This PR avoids some unnecessary sorting overhead (e.g. if the collection is too small) for methods that are repeatably called and where collection sizes of <= 1 are fairly common (e.g. for the ProducesRequestCondition)."*

This description clarifies the change performed, allowing other developers to discuss this improvement. In the remaining discussion within this PR, a second developer agreed with the changes and accepted them to be integrated into the system.

Considering the presence of the keywords related to the NFRs in the title and/or description, we observe that in at least 77% of the PRs, the developers were aware of the NFR before opening the PR. This result indicates that developers have NFR concerns during other activities such as tests, maintenance, or external use. We can summarize our findings as follows.

> ***Finding 1:*** *Developers usually mention the NFRs when creating PRs. Thus, developers are aware of and concerned about these NFRs even before opening the PR. This indicates that NFRs are a primary concern during software maintenance and evolution.*

To better understand the content of NFR discussions, we focused on analyzing the titles and descriptions within PR discussions. For that purpose, we randomly selected 160 PRs for qualitative analysis (see Section 5.3.2 - Step 6). Through open coding, we generated 35 codes and 9 categories, allowing us to understand the content of these titles and descriptions. The list of codes and categories are presented in our complementary material[1].

NFR problem identification and resolution. Among the codes identified, 25 were associated with the category `NFR Problem Identification` and/or `NFR Problem Resolution`. These two categories describe cases where the developers identify and describe a change related to NFR. The developers either provide a solution themselves or identify the problem and leave it for other developers to solve. Let us consider the example from Spring Boot (Spring Boot 2022). In that case, the developer described a solution to improve the `Performance`. Following, we provide a snippet from what the developer stated:

> *"As described in Issue #16401 there is an optimization opportunity in SpringIterableConfigurationPropertySource when checking for CacheKey equality[...]. This is*

Table 5.4: NFR Types and Sub-categories Resulting From the Open-coding on PR Title and Description.

NFR Type	Subcategory
Maintainability	Documentation (28), Code Simplification (18), Feature Enhancement (11), Readability (3), Encapsulation (1), Extensibility (1), Move Component (1), and Setup (1)
Performance	Complexity (6), Concurrency (2), Memory Usage (2), Response Time (1), and Synchronization (1)
Robustness	Error Recoverability (22), Error Representability (9), and Error Scope (4)
Security	Feature Addition (13), Inconsistent Behavior (9), Information Protection (7), and Parameters Customization (1)

due to the fact that the internal key inside CacheKey is copied in order to fix PR #13344 and can thus not benefit from a == comparison[...]. The idea of this PR is to introduce a flag that effectively disables the copying of the internal key under certain circumstances[...]. With the applied changes, I see major improvements compared to an M2[2] *baseline"*

In the title, this developer described *"optimizing CacheKey handling"*. It is a case where the title and description complement each other. The snippet provided above summarized the PR description. In this example, the developer gave a detailed description of the problem context, where he/she found an optimization opportunity on a class in the system. Then, the developer describes how the problem was solved, thus improving performance. Another developer reviewed the code and accepted the PR change in the following discussion. However, this type of long description did not happen so often in our dataset.

NFR-related topics on the PR title and description. We observed multiple codes related to sub-categories concerning the four NFRs, as shown in Table 5.4. In parentheses, we show the frequency of each subcategory. For each NFR, we subdivided them according to the type of change that was mentioned. We aimed to understand the different topics the title and description address. For instance, we identified 64 cases for `Maintainability`, divided into eight sub-categories: `documentation, code simplification, feature enhancement, readability, encapsulation, extensibility, move component, and setup`. `Documentation` was the most recurrent topic (28 cases) topic for PRs that discuss `Maintainability`. In general, those changes are related to the addition of missing documentation, fixing inconsistencies between the documentation and code behavior, typo corrections, and minor improvements. Due to the space limitation, we will only describe one of the sub-categories. The remaining are detailed at Appendix C.5

[2]M2 are the initials to reference a milestone in the system

For `Performance`, we identified 12 cases divided into five sub-categories: `complexity, concurrency, memory usage, response time, and synchronization`. For this NFR, the most recurrent sub-category was `complexity`, which describes cases where the change increased (or decreased) the computational complexity.

For `Robustness`, we identified 35 codes, divided into three sub-categories related to the NFR: `error recoverability, error representability, and error scoping`. The most recurrent sub-category for `Robustness` was the `error recoverability` (22 cases). This error happens when there is an improper exception handling and clean-up actions after the system raises an exception.

Finally, for `Security`, we identified 30 cases divided into four sub-categories: `feature addition, inconsistent behavior, information protection, and parameter customization`. While the different categories had similar cases, we can observe that `feature addition` was the most frequent category identified in the coding process. Those cases usually relate to adding a wide range of new security-related features, such as new types of encryption, new authentication methods, etc. With these examples and the data presented in Table 5.4, we can define our next finding as follows.

> ***Finding 2:*** *The PR titles and descriptions tend to clearly communicate the intention and the scope of changings addressing NFRs.*

Upon the analyses and findings described above, we can characterize the NFR discussions as being triggered mainly by their title and description and usually discussing the identification and resolution of problems related to the NFR. In addition, these titles/descriptions also encompass multiple sub-categories of the NFRs explored.

By understanding how the PR discussions occur, researchers can explore these characteristics in systems from other communities and domains. Researchers may optimize their efforts based on our findings. For instance, instead of considering the whole discussion to identify a particular NFR discussion, we may consider only the title and description of the PR discussions. It can also avoid noise in the identification process.

In addition, we found in software systems analyzed that developers may use the PRs titles, descriptions, and code changes as a source of documentation about the system's NFRs. In this way, newcomers can use this content to understand how developers address NFRs during the development process. Having this source of documentation is particularly useful in OSSs, where the

Table 5.5: Summary of Developers' Characteristics

Task/ID	D1	D2	D3	D4	D5	D6	D7	D8	D9	D10	D11	D12	D13	D14	D15
PRs Opened	15	3	7	1	16	10	3	11	3	0	4	0	7	2	2
PR Comments	115	3	152	60	11	7	39	125	34	49	47	69	18	12	3
PR Reviews	500	36	419	9	2	0	10	595	71	6	4	19	15	24	17
PR Commits	90	10	111	5	17	10	4	86	58	0	9	2	34	4	0

NFR documentation is often scattered through multiple sources. Considering the importance of understanding how these discussions occur, it is also important to understand who are the developers engaged in NFR discussions. Therefore, next, we investigate the characteristics of developers who discuss NFRs.

5.4.2 Who are the Developers that Most Engage in NFR Discussions

To understand who deals with NFR, we first selected 15 developers by identifying their degree of participation in NFR discussions (see Section 5.3.2 - Step 4). The table respectively presents, for each developer, the number of PRs opened, comments made, reviews, and commits. Table 5.5 summarizes the characteristics of these 15 developers regarding their main tasks on the software system. For instance, developer `#D1` *(i)* opened 15 PRs, *(ii)* made 115 comments, *(iii)* made 500 review comments, and *(iv)* committed 90 times.

We applied the k-means clustering technique for each software system to observe whether developers had similarities regarding their activities. Then, we verified in which cluster each selected developer is classified. For each software system, the clustering results led to four clusters. Notice that although the number of clusters was equal between the three software systems, they didn't exhibit identical characteristics within those clusters. However, we observed similarities between the clusters across repositories, as follows.

- **Newcommers**: It is composed of two clusters, containing around 80% of the total number of developers. Developers in these clusters tend to be newcomers with less experience or other developers who are less active. This is an expected result since the most significant part of contributors is not regular.
- **Active Contributors**: Contains developers who are newbies or recent contributors but highly active. This cluster usually has less than 10% of developers.
- **Experienced Contributors**: They are highly active and experienced contributors on the software system, having top metrics in multiple

tasks (*e.g.*, commits, PRs, refactorings, comments, and reviews). Less than 10% of the developers belong to this cluster.

We observed that the 15 developers initially analyzed belong to cluster *Experienced Contributors*. They are part of the small group of core contributors active in the software system, not only addressing questions related to NFRs. Therefore, this leads us to our third finding.

> ***Finding 3:*** *Developers more actively discussing NFRs are also core contributors having high participation on multiple tasks in the software system (e.g., commits, reviews, and refactorings).*

These findings show the activity of developers on the software system. However, we want to understand what makes these developers discuss NFRs. For that purpose, we analyzed other sources of information available (*e.g.*, GitHub profile and Spring Team page) that support us in understanding what makes developers discuss specific NFRs. Our manual analysis through open coding generated 17 codes and six categories, allowing us to understand these developers. More details about these codes and categories are found in our complementary material[1].

Developers participation on other projects. In our analysis, we could observe that all 15 developers participated in multiple OSSs. Many of them, such as `#D1`, have collaborated in more than 10 software systems. For these developers, it is common that most of the systems they participate in are part of the *Spring* ecosystem. Multiple developers have participated in more than 10 software systems from *Spring*. They also tend to participate in repositories related to NFR, the most discussed in the PRs. For instance, `#D1` participates in multiple software systems related to `Security`, whereas `#D2` participates in several software systems related to documentation, and `#D3` participates in software systems related to `Performance`.

Where developers work. We identified where the developers work through a manual analysis. Almost all developers (12 of 15) work for a company that supports the *Spring* ecosystem. One developer works for a company focused on developing browser and mobile games. Two developers do not have their companies identified. Hence, the developers generally work for a company that directly maintains the software system or for a company that presumably uses *Spring* products.

These results suggest that most of these developers are knowledgeable about the software system since *(i)* they work for the company that maintains

the software system, *(ii)* they describe themselves as engineers for the software system, and *(iii)* they are active in the *Spring* repositories. Through these observations, we see why these developers engage more in NFR discussions. Since the companies where they work are closely related to the *Spring*, they have a strong incentive to ensure that the NFRs are properly addressed. Their high activities in tasks such as opening PRs and reviewing others' PRs suggest that they not only have a good understanding of the NFRs of the system, but they are also proactive in addressing them.

Developers' expertise based on their position. Based on the developers' positions in companies, we could infer their expertise on certain NFRs. For instance, the developer `#D1` has high participation in multiple tasks related to `Security` in the software system. On the software system team page, we observed that `#D1` is *Security Senior Engineer*, a position closely related to the NFR he most discusses. Therefore, due to the nature of this position, we can infer that this developer has high expertise in Security.

A similar scenario happened with the developer `#D2`. This developer mainly revised code related to `Maintainability`. By looking at the PR discussions in which this developer participated, it is also mainly related to the documentation of Spring Security. We observed that this developer is *Senior Technical Writer* of his company. Hence, it makes sense for his main contributions to be related to `Maintainability`, with a focus, especially on documentation.

To understand developers who discussed less but were more focused on discussing NFRs, we selected 48 developers to investigate from the three systems. Initially, we selected 20 of each system, equally distributed considering the four NFRs. However, some developers appeared in more than one of the three systems analyzed, leading us to analyze 48 developers. However, we could not find any specific information about most of them (*e.g.*, they did not have information on their GitHub page).

We can highlight that at least six are open-source enthusiasts and active contributors to multiple open-source projects. Four are main contributors to projects within the *Spring* community. Three of them are closely related to security open-source projects. In this last case, the developers contributed mainly to the Spring Security project. Combined, this information shows similar characteristics from the group of 15 developers early analyzed. These results give us the first glimpse of the characteristics of developers dealing with NFRs on open-source systems. With these aspects identified, we can summarize our next findings as follows:

Table 5.6: Participants Characterization

	Average	Median	Mode
Age	36.13	33.50	32
Experience in Years	12.06	10	3, 5 and 20
Number of Projects Participated	16.61	8.50	2 and 8

> ***Finding 4:*** *Developers who discuss NFRs tend to be part of companies closely related to the Spring community or repositories related to the NFR they most discuss, justifying their engagement in addressing NFRs in OSSs.*

By understanding these developers' characteristics, system managers and leaders can allocate specialists on NFRs to specific tasks or positions in the company. In addition, newcomers can benefit from the knowledge that may be acquired by understanding how the more experienced developers deal with NFRs. Moreover, the team managers can also allocate these developers to review decisions regarding the NFRs and avoid future violations of these requirements. Since these violations are symptoms of design problems in the system (Sousa *et al.* 2017), this concern with the NFRs is a key activity to keep the software system healthy through its evolution.

5.4.3 How Developers Perceive and Address NFRs in Their Systems

To have a broader view of the perception of developers regarding NFRs, we ran a survey with 44 developers from different systems, mainly from closed-source systems. By observing both closed and open-source developers' perspectives, we aim to generalize our findings. Next, we present the results of this survey.

Participants characterization. In Table 5.6, we provide an initial characterization of participants in your survey. Concerning their experience in years and the number of projects, we can observe that both more experienced and less experienced developers participated in your survey. When we look at their experience in years, we can observe that the developers vary in experience, but through the mean and median, we can observe high levels of experience.

Considering the education level, most participants had a master's degree (17), followed by a bachelor's degree (12) and a PhD degree (8). Regarding the roles in their companies, most of them were software developers (16) and/or software engineers (15). Concerning the programming languages that they were

proficient in, Java (27), Python (25), and JavaScript (19) stand out. Notice that the questions about the roles and programming languages were multiple-choice. By observing these demographics, we see a diversity in the profile of the participants.

Why developers discuss NFRs. In our survey, we asked what makes the participants engage in discussions related to NFRs. Most of them answered that their primary motivation is (i) the impact on the quality of the software they use (37), (ii) the alignment with the software objectives (30), and (iii) the potential risks that may arise if the NFR is not addressed (22). These motivations suggest that the participants value the discussion since they perceive the NFRs as key aspects of the software quality. This is indicated once the majority of participants (42 out of 44) stated that they consider NFRs a crucial aspect for the success of the software system. These answers are the first insights into why developers invest time and effort in addressing NFRs.

When developers discuss NFRs. We asked the developers which phase of the software lifecycle they discussed NFRs. Their responses were diverse, mentioning almost all phases: development (31), test (25), and design (3). The developer `#P11` even mentioned that during the early development phase, the NFRs are easily identified when the team is giving due importance to identifying and analyzing the NFRs.

The developer `#P6` mentioned using the PR discussions within their project for revision purposes. In his/her company, PR discussions are used to discuss performance improvements. `#P6` also stated the tests being applied at the end of the day to check non-conformities with the *Performance*, which are used to fix problems that could appear in production. `#P36` even mentioned a more sophisticated way to address the NFRs. `#P36` noted that the project had a spreadsheet showing the system's NFRs, which a software architect filled out. Altogether, developers cited multiple approaches to ensure the system meets NFRs: pen tests for Security, continuous integration using Static Application Security Testing (SAST), Dynamic Application Security Testing (DAST), and stress testing for Performance. We have the following finding by understanding why and when developers discuss NFRs in their systems.

> ***Finding 5:*** *Developers discuss NFRs since they recognize their impact on the software quality, identifying such requirements as key to the success of the software systems. They discuss the NFRs throughout*

the software's whole lifecycle, using diverse methods to ensure compliance, such as PR discussions and tests.

How developers perceive and address NFRs. We asked developers about their strategies to guarantee that the NFRs were adequately applied during the software development. Among their answers, two approaches stand out: collaboration with other team members to gather insights regarding the NFRs (35) and review and refine the definitions of NFRs in the early stages of the development process (31).

We observed developers employ specific NFR analysis tools (19). In these cases, we noticed that teams tend to address NFRs reactively. For instance, `#P17` explicitly mentioned that the NFRs were addressed only when specific metrics were below acceptable. In this example, automated tools were used to ensure that the NFR was being met. (*e.g.*, use of SonarQube for minimum coverage of automated tests). In addition, `#P17` mentioned that their approach was proactive only if the NFR was a requirement defined early by the team of requirements. That reactive approach appeared when they expressed no defined process to deal with the NFRs.

Other developers mentioned specific strategies to address the NFRs. `#P13` mentioned that the team uses developers' previous experiences to define particular NFRs for the system, and then they are reevaluated if necessary during the review phases. `#P12` stated that they first looked for NFR documentation available. However, in cases where this was not available, they asked for help from more experienced team members. If there is still no solution, a meeting is scheduled between the stakeholders involved. `#P12` even mentioned that if it reaches the point of needing to do this meeting, it was proof of a failure in requirements collection and analysis. Indeed, the lack of proper documentation was the most mentioned in our questions to understand the challenges in addressing NFRs (20 cases). We can resume these results in the following finding.

Finding 6: *Developers primarily address NFRs through collaborative efforts, often resorting to reactive approaches, focusing on NFRs only when automated tools indicate possible problems. This happens because the missing documentation is one of the most significant challenges.*

Which developers address NFRs. We asked the developer whether there is someone specifically dealing with the NFRs in the system. Most develop-

ers (26) answered that there was no specific role to handle such requirements. Nine developers mentioned that the entire team is responsible for the NFRs, depending on what is needed. For instance, `#P4` stated that the most experienced architects and/or developers were most concerned with the NFRs.

Since, in multiple cases, no developer focused on only solving NFR problems, some practices are followed in the companies to ensure that the NFRs are being adopted. `#P30` stated that before joining the company, the team members were trained to understand and keep the NFRs security and maintainability of all activities carried out within it. `#P33` also mentioned that within the company, they had specialists in certain NFRs, such as security. However, they are not exclusive to only one project. Instead, they work on all of the company's projects.

Strategies to address NFRs. We gathered information on the best strategies to address NFRs within the developers' systems. We categorized these strategies based on their answers, as follows.

- **Prioritization and planning:** NFRs should be treated with as much priority as other requirements. They should be planned in advance and reviewed throughout the project.
- **Identification and discussion:** NFRs should be identified and discussed early in the development process, ideally in the design phase. During the evolution of the software, these NFRs should be revisited if necessary and discussed with stakeholders the possible changes that will be done to keep the NFR adequate.
- **Used of technologies allied with testing:** The adequacy of the NFR can be verified through technologies already approved by the market, where their non-functional requirements satisfy the project's complexity. In addition, the processes of *continuous integration* and *continuous delivery* (CI/CD) can help in the identification of such problems.
- **Benchmarks:** Using benchmarks to simulate the behavior of a piece of code or algorithm under different conditions is recommendable, enabling the review and refactoring of the code when it is not meeting the project-specified NFRs.
- **Documentation and best practices:** By keeping the NFRs well-documented, developers will have a starting point to address an NFR problem when it appears. In addition, it is good to have these best practices on handling NFRs documented since each system may have its own particularity.

– **Long-Term mindset:** Properly addressing NFRs will enhance the software's lifetime. To guarantee this, a system (i) should have a good user experience, (ii) should be designed to scale, and (iii) should be easy to maintain by future developers. These characteristics rely on NFRs such as performance, robustness, security, and maintainability.

By investigating how developers perceive and handle NFRs in open-source and closed-source systems, we obtained initial insights into their discussions regarding these system requirements. Through an analysis of their perspectives and typical tasks, we compiled a documented overview of best practices for addressing NFRs to ensure the long-term health of software systems. As one survey participant aptly said: *"Paying attention to NFRs can mean the difference between the success or failure of a software project"*.

5.5 Threats to Validity

This section discusses the threats to the *internal* and *external* validity of our study as follows.

Threats to Internal Validity: *Labeling process.* The procedure adopted for the manual identification of NFRs in PR can be posed as a threat. We adopted a well-structured process to overcome such a threat to the study's validity. Besides all collaborators being experienced with the NFRs types investigated, we also provided complementary material with details about the NFRs to avoid misclassification. In addition, our qualitative analysis ensured that the author evaluated a PR classified by other collaborators. Finally, when there was any doubt about the classification, a second author also classified the PR to ensure the reliability of the classification. *Clustering algorithm.* Regarding clustering, the k-means (MacQueen 1967) algorithm expected an explicit indication of the number of clusters for grouping the data. We mitigate that threat by applying the Silhouette analysis (Rousseeuw 1987) that enables us to determine the most appropriate number of clusters for our data. Furthermore, the set of metrics supplied to the clustering algorithm to group developers could be a limitation. For this reason, we manually reviewed the metrics in the feature engineering process to ensure they represent the proper dimensions that characterize the developer's tasks in the repository.

Threats to External Validity: *System ecosystem generalization.* We selected three systems for our analysis that might be a threat. However, our selection was based on a set of criteria (Section 5.3.2) to ensure that we are selecting project repositories with proper characteristics for our study. In this way, we decided to focus on OSSs from the same ecosystem to perform an in-

depth analysis and understand this scenario in specific. Even though the generalization of the study findings is limited to the Spring ecosystem, we provided a detailed research methodology, allowing the replication of the study steps to investigate other software (eco)systems. *Developers' generalization.* The number of developers analyzed in *RQ2* can be another threat. However, since our goal was to analyze only the developers dealing with NFRs, we focused on two steps to select them. First, we divided them into quartiles regarding software system activity and their presence in NFR-related discussions. Then, we first selected the developers who were present on the 4^{th} quartile, meaning they were highly active on the software system and regarding the NFRs. Second, we selected developers from the other quartiles but with a higher focus on NFR in the discussions. Finally, we applied the survey to complement the information gathered from these first developers.

5.6 Conclusion

To understand how developers discuss NFRs on OSS, we first characterized these discussions on three systems from the Spring ecosystem. For that purpose, seven collaborators of this paper performed a manual classification of 1,533 PR discussions. We considered four NFRs for this classification: maintainability, security, performance, and robustness, which were the NFRs most present on the software systems analyzed. Through this classification, we built a dataset of PR discussions regarding the presence of NFRs. With this dataset, we characterized the NFR discussions and identified the developers discussing these requirements.

We observed that in more than 77% of the PRs, discussions related to NFRs are triggered in the title and description, both provided by the developer opening the PR. That lets us understand that the developer who opens the PR knows the NFR that should be discussed. These discussions' content is mainly related to introducing and resolving NFR problems. When we investigated the developers discussing NFRs, we observed that they have high participation in multiple tasks in the software system (*e.g.*, commits, and reviews). They also usually have a role in the company that they work for, closely related to a specific NFR (*e.g.*, Spring Security Senior Engineer).

As a contribution, we provide the dataset created by our manual classification, which can be used to develop new and improved automated classification mechanisms for NFRs. The characterization we provided of PR discussions can be used as a starting point for developing tools for the automatic detection of NFRs. Furthermore, the mapping of developers who discuss NFR can be

used by system managers who want to understand specialists' characteristics in certain NFRs. Hence, he can allocate his team based on these skills. In future studies, we plan to explore the impact of the identification of NFRs on PR discussions can have on the system's design and expand our analyses with more ecosystems.

6
Conclusions and Future Work

A design problem is the result of one or more design decisions that have a negative effect on non-functional requirements (NFR) of a system (*e.g.*, maintainability and robustness). Identifying and removing these problems is essential as allowing them to prevail in the system implementation can result in high maintenance costs and even lead to the system discontinuation. Current studies focus solely on considering maintainability code smells as the only driving symptom for the design problem identification, which can lead to incomplete contextual information to support developers during the identification process. Hence, there is a need to explore new types of information that developers can use to identify design problems.

In this thesis, we explored how developers can leverage the use of robustness smells in combination with maintainability smells to identify design problems effectively. We identified the benefits and challenges of relying on these two types of symptoms. Through our initial research steps, we realized that only developers who are able to somehow discuss NFRs can be able to identify design problems. As a consequence, only these developers would be able to make proper meaning of maintainability and robustness smells (and other NFR-related smells) along design problem identification.

Developers deal with NFRs during discussions in OSSs. To complement the investigation, we performed a survey with developers from both open and closed-source systems. We searched for similarities and differences in developers' approaches to identify and address NFRs. Through this analysis, we provided a documentation of best practices and strategies followed by experienced developers when dealing with NFRs. With this guidance, non-experienced developers can learn how to address NFRs within their systems and potentially avoid future design problems related to these NFRs.

6.1
Thesis Contributions

The main goal of this thesis is to *support developers in the identification of design problems by relying on the use of multiple and diverse symptoms.* To reach our goal, we performed three main studies to gain insights into how

developers can use and combine symptoms of design problems to identify and address them.

In our first study (see Chapter 3), we aimed to understand how developers could use patterns of maintainability smells to identify design problems. For that purpose, we conducted a quasi-experiment with 13 professional developers. We explored to what extent the use of maintainability smell patterns could assist them on identifying design problems in practice. This first study was divided into two main steps. First, we asked the participants to identify design problems in their software systems. They used a tool we developed to show the candidates of design problems in the systems. Second, we asked the participants whether the information about the smell patterns was enough to identify and remove the design problems.

We identified the factors that led the developers to perceive a design problem in the system. One factor was the *context of the system*, where the developer perceived the problem as inevitable. Another factor was the *use of customized strategies and thresholds* for the identification of maintainability smells. In that case, the developers mentioned false positives being detected, such as a long method that was not necessarily long in the context of the system analyzed.

We observed that the smell patterns for the design problems *Fat Interface*, *Concern Overload*, and *Scattered Concern* are highly promising as indicators of their occurrences. However, their effectiveness can be enhanced when complemented with information about dependencies and concerns (related to the relevant symptoms) within the software system. In this context, the first contribution of this thesis is the following.

> **1st Contribution.** Developers can benefit from using maintainability smells patterns as a practical approach to identify design problems.

Developers tend to perceive design problems as issues that they did not intentionally introduce and that have a noticeable impact on multiple software architecture components (see 3). Then, we analyzed some factors that induced developers' perceptions regarding the presence of design problems. These factors included the context (*e.g.* use of APIs that led to design problems) of the system they worked on and the strategies used to detect the maintainability smells. Without such information, developers were not able to complete the design problem identification. Developers mentioned that complementary information would be needed to identify more complex design

problems, such as those affecting the system's modularity. In this context, we performed our second study.

In our second study (see Chapter 4), we aimed to understand how robustness smells could be used as (complementary) symptoms of design problems. For that purpose, we analyzed over 160k methods from 10 OSSs. We divided this study into three main steps. First, we explored how any kind of robustness change (*i.e.*, change made within the catch block) could be correlated with maintainability smells. Second, we explored whether these robustness changes could harm methods (by considering the density of maintainability smells) that underwent such a change. Finally, we investigated how robustness smells (*i.e.* poor robustness changes in the catch block) could be combined with the maintainability smells (already explored in Chapter 3) for more effective identification of design problems.

In our analyses, we observed that methods that underwent robustness changes tended to co-occur with the maintainability smells *Dispersed Coupling* and *Feature Envy*. By performing a manual analysis, we observed that these two maintainability smells were introduced in the methods when the robustness changes were performed. We also observed that even small changes in the exceptional code, such as a comment left in an empty catch block, can indicate (or reinforce) maintainability smells in the method.

By performing even small changes in the exceptional code, developers should be aware that the target code can be affected by maintainability smells. Furthermore, developers can proactively address emerging design problems by taking into consideration the maintainability smell patterns studied (see Chapter 3). In addition, we observed that classes with methods that underwent robustness changes have a high density of smells, confirming the negative impact of these changes.

By exploring the combination of robustness smells and maintainability smells patterns, we observed that some robustness smells (*empty catch block* and *catch generic exception*) exhibit a close relation to system modularity problems. For instance, when exceptions are caught in a generic manner, as indicated by these two smells, it can imply that these smelly methods are dealing with multiple concerns. In the face of multiple unrelated concerns, developers can be forced to deal with the exceptions in this generic way. Hence, by observing the presence of these robustness smells and combining them with maintainability smells patterns, developers can be more confident in identifying design problems. In this context, we have our second contribution to this thesis.

> **2nd Contribution.** Developers can benefit by combining both robustness smells and maintainability smell patterns to identify design problems more effectively.

Understanding how developers can rely on different types of code smells is fundamental for the identification of design problems. The non-conformity with NFR is another symptom of design problems. Understanding and documenting strategies and techniques developers use when dealing with NFRs can support less experienced developers to identify and properly address design problems. Through the adoption of different symptoms, developers will be more likely to maintain the code and prevent unwanted designproblems in the future. To understand developers who deal with NFRs and their strategies, a third study was performed (see Chapter 5)

This study was divided into four steps. In the first step, we created a dataset of pull request (PR) discussions. In the second one, we characterized the NFR within the PR discussions; we focused on understanding the main characteristics of these discussions and how they occur. In the third step, we investigated the developers' engagement in these discussions. Finally, we surveyed the developers to understand how they identify, perceive and address NFRs within their systems.

For this study, we created a dataset of 1,533 PR discussions labeled with their respective NFR. That allowed us to characterize the discussions and identify the developers participating. On our characterization, we identified that developers tended to mention NFR-related keywords in the title and the description of the PR, which reinforces that NFRs are the developers' primary focus during the software's maintenance and evolution. We also observed that developers describe multiple NFR-related keywords, showing their clear intention on the PR opened and helping define the scope of the change pointed out or made. Through analyzing these discussions, we also selected 63 developers who showed the highest level of involvement in NFR discussions.

In our investigation, we observed that these developers were the central contributors to the systems, having high participation in multiple tasks, such as commits, reviews, and refactorings. By observing their GitHub profile and social media, we also noticed that they usually had roles in their companies related to the NFRs they discussed. For instance, a highly active developer in discussing *security* was the *Senior Security Engineer* of the company where he worked. That shows the paramount importance of their practical expertise in addressing these NFRs in the systems.

To enhance our comprehension on how they address NFRs, we surveyed 44 developers. We observed that developers discuss NFRs often because they are essential to the software evolution according to their perception. These discussions were made especially to keep the NFRs aligned with the project objectives. These discussions took place during the whole software development life cycle, going from the design phase to the maintenance phase. They highlighted the importance of discussing NFRs, especially in the design phase, with the presence of the clients.

The participants also pointed out the use of practices to guarantee that the NFR was being satisfied and reaching the expectations of the clients, for example, the use of continuous integration and continuous delivery. For this purpose, they relied on specific techniques to test each type of NFR, such as stress testing to measure the performance. All participants stated that they have experience addressing NFRs. Nevertheless, the developers chosen to address NFR-related issues were often the most experienced members within their teams, such as the software engineers and architects.

Based on our investigation with 63 developers from the Spring ecosystem and 44 developers surveyed, we found the strategies they tend to apply to address the NFRs. For example, they highlighted the importance of prioritizing and planning the NFR in the early stages of software development. They also mentioned the use of static analysis tools combined with testing to help address NFRs. The use of benchmarks to identify when the NFRs were not met in the system was also mentioned to help address the NFRs properly. Additionally, developers mentioned the need for easy access to the documentation. These strategies help developers improve the software system's overall quality; and by addressing NFR problems in the system, future problems, such as design problems, can be avoided. Based on these findings, we have our next contribution.

> **3rd Contribution.** Understanding how developers identify and address NFRs within their systems can help in the prevention of design problems.

With these three contributions, we take a step further in helping developers deal with design problems. We are advancing the state-of-the-art by showing how the developers use symptoms of design problems in practice and how they can combine multiple symptoms to identify such problems. In this thesis, we analyze how the combination of maintainability and robustness smells can assist developers in this task. In particular, we assess how combining both

smells can provide more information regarding the design problem, increasing its identification accuracy. In addition, looking for a more practical perspective, we showed how developers perceive NFRs within their systems. We also presented developers address NFRs' non-conformities that may arise during the system's lifecycle, documenting their best practices.

Figure 6.1 depicts a methodology grounded on the findings of this thesis for assisting developers in the identification of design problems. In this methodology, we employed a specific notation:

- Gray squares represent the phases.
- Blue squares represent the actions that need to be executed.
- Hexagons represent the artifacts utilized during these actions.
- Arrows represent the sequence of steps to be followed.

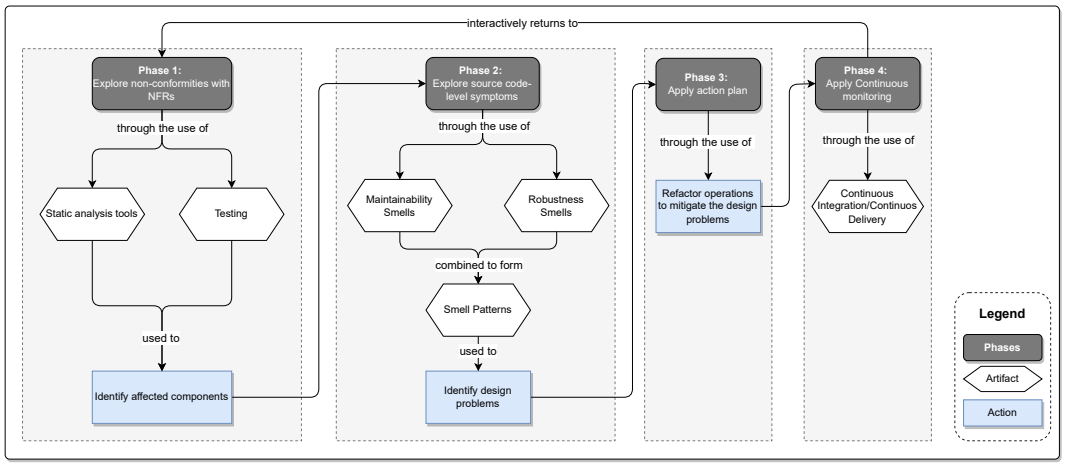

Figure 6.1: Workflow on Design Problem Identification.

This methodology is divided into four major phases, described as follows.

Phase 1. Explore the non-conformities with NFRs: First, the developer should identify non-conformities with the NFRs defined for the software system. This step is an essential initial activity as design problems negatively affect these requirements. As observed in Chapter 5, developers address the NFRs when identifying such non-conformities (*e.g.*, when a component has issues with the code readability and/or is tightly coupled with other component(s)). Developers can rely on static analysis tools and testing to identify such non-conformities. For instance, static analysis tools can measure code complexity metrics and pinpoint components that are hard to maintain or prone to errors. Considering the tests, developers can evaluate the test coverage and tests that assess the software's ability to meet NFRs related to

Table 6.1: Smells Patterns Considering Maintainability and Robustness Smells

Design Problem	Maintainability Smells	Robustness Smells
Concern Overload	Complex Class, Divergent Change, Feature Envy, God Class/Complex Class, Intensive Coupling, Long Method, and Shotgun Surgery	Empty Catch Block, Catch Generic Exception
Scattered Concern	Dispersed Coupling, Divergent Change, Feature Envy, God Class/Complex Class, Intensive Coupling, and Shotgun Surgery	Empty Catch Block, and Catch Generic Exception

robustness, such as fault tolerance and error handling. This phase serves as a way to identify these components that may be negatively affecting the NFRs in the system. Then, the developer should focus on such a component for deeper analysis.

Phase 2. Explore source code-level symptoms: Once the developer has identified the components to analyze, they can use the patterns discussed in this thesis. In Chapter 3, specifically in Section 3.2.2, we present patterns that focus exclusively on maintainability smells (see Table 3.1). Furthermore, in Table 6.1, we provide the expanded patterns that now include robustness smells. Developers can rely on both types of patterns when they deem necessary, based on the systems they are identifying the design problems. Through the use of these patterns, developers can be more confident regarding the presence of a design problem.

Phase 3. Apply action plan: Once the developer has identified the design problem, they must address it. Developers can use refactoring operations to remove or reduce the impact of design problems in the system. Code smells play a significant role in identifying opportunities for these refactorings. In Chapter 3, we investigated how the smell patterns can be combined with the refactorings to mitigate the design problem. In Appendix A.1, we provide the full list of refactorings for each respective code smell. Therefore, by using the patterns in this thesis, the developers can apply such refactorings to mitigate the design problem.

Phase 4. Apply continuous monitoring: Once the developer removes the design problem, the next step is to keep monitoring the system. That is needed since new problems may emerge during the development phase. For monitoring, the methodology proposed suggests using continuous integration/continuous delivery (CI/CD) processes. CI/CD tools generate reports and logs for each build and deployment. Developers should monitor these reports for any non-conformities with NFRs. The CI/CD pipeline should prevent the deployment of the code to production until the non-conformities are addressed. Failing the

build provides immediate feedback to the development team and enforces NFR compliance. As identified in Chapter 5, the developers apply such approaches to keep informed whether the NFRs meet the expectations, avoiding non-conformities with such requirements. Therefore, this phase has a link with the first phase, as this is an iterative process that must be performed through the system lifecycle.

By following this methodology, the developer can systematically address design problems in the system. This methodology helps ensure that the software system maintains its quality and evolves over time while meeting the desired NFRs. Developers and researchers interested in improving code quality can explore this thesis as a valuable resource to understand better how to use different symptoms effectively as indicators of design problems. Finally, by following the documented strategies outlined in this thesis, they can apply and adapt them to their own specific scenarios, thus proactively preventing and mitigating design problems. This thesis opens up multiple possibilities, such as incorporating machine learning techniques to automatically detect code quality issues, such as design problems. We discuss these possibilities in the next section.

6.2 Future Work

Empirical studies with the Smell Patterns tool. In our first study, we created a tool for the developers to identify the possible design problems within their systems (see Chapter 3). This tool can be enhanced with new types of symptoms (and patterns) to complement our current catalog of smell patterns (see Section 4.5.3) The catalog can also be enhanced with the combination of robustness smells and maintainability smell patterns (see Chapter 4).

Combination of symptoms. Multiple types of symptoms can be used for the identification of design problems. In this thesis, we focused on some of them as justified in previous chapters. In future work, we can explore other types of symptoms. For instance, the detection of violations of design patterns (Gamma *et al.* 1995) could be used together with code smells. Their co-occurrence may indicate a design pattern was misused in a particular context of the project. Moreover, based on how we explored the combination of maintainability smells and robustness smells, new studies can be performed to explore new combinations of symptoms.

Tools to automatically identify NFRs. In Chapter 5, we created a manually curated dataset of 1,533 PR discussions labeled with their respective

NFR discussed. In this dataset, we provided *(i)* the NFR identified, *(ii)* the phrase that contains the keywords related to the NFR, *(iii)* the keywords related to the NFR, and *(iv)* the location where the keyword appeared. Moreover, we provided several keywords related to the NFRs. Tool builders and researchers can use this dataset as input to build natural language processing (NLP) models aiming to identify NFRs automatically.

Investigation of the strategies used by experienced developers to address NFRs. In Chapter 5, we documented the strategies and perceptions that developers use to address NFRs. Researchers can explore these strategies to understand what can be applied in practice to guarantee software quality and avoid design problems. It is worth investigating whether systems adhering to these practices experience fewer NFR-related issues and exhibit smoother evolution (*e.g.* reduced maintenance costs) compared to systems that do not adopt such practices.

Expertise Assessment Model: In Chapter 5, we highlighted strategies that developers can apply to address NFRs. Through these results, a machine learning model can be developed for assessing and quantifying the expertise of software developers or teams in handling NFRs. This could involve considering various factors, including technical knowledge, experience, soft skills, and the ability to balance conflicting NFRs. In addition, studies can be carried out considering the correlation between the presence of NFR experts in a development team and project success metrics.

Assessing Developer Proficiency in Addressing NFRs: In Chapter 5 we introduced a preliminary study focused on identifying how software developers deal with NFRs through their engagement in Pull Request (PR) discussions. In future work, we aim to provide automatic support for getting information about the developer's level of expertise in solving NFR issues based on their historical contributions. This could involve analyzing developers' contributions, code changes, and discussions. We plan to incorporate a scoring system that assigns weights to different competencies and skills based on their relevance and importance within specific NFR domains. Thus, it can help recruiters and managers assess an overall score for the developers' NFR expertise and assign who could be more able to identify and fix a design problem.

Bibliography

[1st Complementary 2022] COMPLEMENTARY MATERIAL, C. .. https://github.com/andersonjso/sbes22-smell-patterns-indicators, 2022.

[2nd Complementary 2022] OLIVEIRA, A.. **Complementary Material, Chapter 4**. https://github.com/robustnessdp/robustness-changes, 2023.

[APM Agent Java 2023] **Elasticsearchrestclientinstrumentation.java at apm-agent-java**. https://github.com/elastic/apm-agent-java/blob/32a364/apm-agent-plugins/apm-es-restclient-plugin/apm-es-restclient-plugin-5_6/src/main/java/co/elastic/apm/es/restclient/v5_6/ElasticsearchRestClientInstrumentation.java. (Accessed on 01/11/2023).

[Abdi and Williams 2010] ABDI, H.; WILLIAMS, L. J.. **Principal component analysis**. Wiley interdisciplinary reviews: computational statistics, 2(4):433–459, 2010.

[Ameller *et al.* 2012] AMELLER, D.; AYALA, C.; CABOT, J. ; FRANCH, X.. **How do software architects consider non-functional requirements: An exploratory study**. In: 2012 20TH IEEE INTERNATIONAL REQUIREMENTS ENGINEERING CONFERENCE (RE), p. 41–50. IEEE, 2012.

[Asaduzzaman et al. 2016] ASADUZZAMAN, M.; AHASANUZZAMAN, M.; ROY, C. K. ; SCHNEIDER, K. A.. **How developers use exception handling in java?** In: 2016 IEEE/ACM 13TH WORKING CONFERENCE ON MINING SOFTWARE REPOSITORIES (MSR), p. 516–519. IEEE, 2016.

[Baker *et al.* 2011] BAKER, A.; VAN DER HOEK, A.; OSSHER, H. ; PETRE, M.. **Guest editors' introduction: Studying professional software design**. IEEE Software, 29(1):28–33, 2011.

[Baker *et al.* 2019] BAKER, C.; DENG, L.; CHAKRABORTY, S. ; DEHLINGER, J.. **Automatic multi-class non-functional software requirements classification using neural networks**. In: 2019 IEEE 43RD ANNUAL

COMPUTER SOFTWARE AND APPLICATIONS CONFERENCE (COMPSAC), volumen 2, p. 610–615, 2019.

[Barbosa et al. 2014] BARBOSA, E. A.; GARCIA, A. ; BARBOSA, S. D. J.. **Categorizing faults in exception handling: A study of open source projects**. In: 2014 BRAZILIAN SYMPOSIUM ON SOFTWARE ENGINEERING, p. 11–20. IEEE, 2014.

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

[Bashar 2001] NUSEIBEH, B.. **Weaving together requirements and architectures**. Computer, 34(3):115–119, 2001.

[Bass et al. 2003] BASS, L.; CLEMENTS, P. ; KAZMAN, R.. **Software Architecture in Practice**. Addison-Wesley Professional, 2003.

[Bhowmik *et al.* 2019] BHOWMIK, T.; DO, A. Q.. **Refinement and resolution of just-in-time requirements in open source software and a closer look into non-functional requirements**. Journal of Industrial Information Integration, 14:24–33, 2019.

[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: 2019 ACM/IEEE INTERNATIONAL SYMPOSIUM ON EMPIRICAL SOFTWARE ENGINEERING AND MEASUREMENT (ESEM), p. 1–11. IEEE, 2019.

[Binkhonain and Zhao 2019] BINKHONAIN, M.; ZHAO, L.. **A review of machine learning algorithms for identification and classification of non-functional requirements**. Expert Systems with Applications: X, 1:100001, 2019.

[Booch *et al.* 2005] BOOCH, G.; RUMBAUGH, J. ; JACOBSON, I.. **The Unified Modeling Language User Guide**. Addison-Wesley, Boston, 2005.

[Boukhdhir *et al.* 2014] BOUKHDHIR, A.; MAROUANE, K.; SLIM, B.; JOSSELIN, D. ; BEN, S. L.. **On the use of machine learning and**

search-based software engineering for ill-defined fitness function: A case study on software refactoring. In: Le Goues, C.; Yoo, S., editors, SEARCH-BASED SOFTWARE ENGINEERING, p. 31–45, Cham, 2014. Springer International Publishing.

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

[Brown *et al.* 1998] BROWN, W.; MALVEAU, R.; III, H. M. ; MOWBRAY, T.. **Refactoring software, architectures, and projects in crisis**. Google Scholar Google Scholar Digital Library Digital Library, 1998.

[Brunet *et al.* 2014] BRUNET, J. A.; MURPHY, G. C.; TERRA, R.; FIGUEIREDO, J. ; SEREY, D.. **Do developers discuss design?** In: PROCEEDINGS OF THE 11TH WORKING CONFERENCE ON MINING SOFTWARE REPOSITORIES, MSR 2014, p. 340–343, New York, NY, USA, 2014.

[Buschmann *et al.* 1996] BUSCHMANN, F.; MEUNIER, R.; ROHNERT, H.; SOMMERLAD, P. ; STAL, M.. **Pattern-Oriented Software Architecture - Volume 1: A System of Patterns**. Wiley Publishing, 1996.

[Cabral and Marques 2007] CABRAL, B.; MARQUES, P.. **Exception handling: A field study in java and. net**. In: EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, p. 151–175. Springer, 2007.

[Cacho et al. 2014] CACHO, N.; CÉSAR, T.; FILIPE, T.; SOARES, E.; CASSIO, A.; SOUZA, R.; GARCIA, I.; BARBOSA, E. A. ; GARCIA, A.. **Trading robustness for maintainability: an empirical study of evolving c# programs**. In: PROCEEDINGS OF THE 36TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, p. 584–595, 2014.

[Camacho *et al.* 2016] CAMACHO, C. R.; MARCZAK, S. ; CRUZES, D. S.. **Agile team members perceptions on non-functional testing: influencing factors from an empirical study**. In: 2016 11TH INTERNATIONAL CONFERENCE ON AVAILABILITY, RELIABILITY AND SECURITY (ARES), p. 582–589. IEEE, 2016.

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

[Cedrim *et al.* 2017] CEDRIM, D.; GARCIA, A.; MONGIOVI, M.; GHEYI, R.; SOUSA, L.; DE MELLO, R.; FONSECA, B.; RIBEIRO, M. ; CHÁVEZ, A.. **Understanding the impact of refactoring on smells: A longitudinal study of 23 software projects**. In: PROCEEDINGS OF THE 2017 11TH JOINT MEETING ON FOUNDATIONS OF SOFTWARE ENGINEERING, ESEC/FSE 2017, p. 465–475, New York, NY, USA, 2017. ACM.

[Chatterjee *et al.* 2021] CHATTERJEE, R.; AHMED, A.; ROSE ANISH, P.; SUMAN, B.; LAWHATRE, P. ; GHAISAS, S.. **A pipeline for automating labeling to prediction in classification of nfrs**. In: 2021 IEEE 29TH INTERNATIONAL REQUIREMENTS ENGINEERING CONFERENCE (RE), p. 323–323, 2021.

[Chen *et al.* 2005] CHEN, X.; DAVARE, A.; HSIEH, H.; SANGIOVANNI-VINCENTELLI, A. ; WATANABE, Y.. **Simulation based deadlock analysis for system level designs**. In: PROCEEDINGS OF THE 42ND ANNUAL DESIGN AUTOMATION CONFERENCE, DAC '05, p. 260–265, New York, NY, USA, 2005. ACM.

[Chung and Nixon 1995] CHUNG, L.; NIXON, B. A.. **Dealing with non-functional requirements: Three experimental studies of a process-oriented approach**. In: 17TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, ICSE '95, p. 25–37, New York, NY, USA, 1995. Association for Computing Machinery.

[Chung *et al.* 2012] CHUNG, L.; NIXON, B. A.; YU, E. ; MYLOPOULOS, J.. **Non-functional requirements in software engineering**, volumen 5. Springer Science & Business Media, 2012.

[Ciupke 1999] CIUPKE, O.. **Automatic detection of design problems in object-oriented reengineering**. In: PROCEEDINGS OF TECHNOLOGY OF OBJECT-ORIENTED LANGUAGES AND SYSTEMS - TOOLS 30 (CAT. NO.PR00278), p. 18–32, Aug 1999.

[Cleland-Huang *et al.* 2007] CLELAND-HUANG, J.; SETTIMI, R.; ZOU, X. ; SOLC, P.. **Automated classification of non-functional requirements**. Requirements engineering, 12(2):103–120, 2007.

[Coelho *et al.* 2015] COELHO, R.; ALMEIDA, L.; GOUSIOS, G. ; VAN DEURSEN, A.. **Unveiling exception handling bug hazards in android based on github and google code issues**. In: 2015

IEEE/ACM 12TH WORKING CONFERENCE ON MINING SOFTWARE REPOSITORIES, p. 134–145. IEEE, 2015.

[Corbin and Strauss 2014] CORBIN, J.; STRAUSS, A.. **Basics of qualitative research: Techniques and procedures for developing grounded theory**. Sage publications, 2014.

[Cosmina *et al.* 2017] COSMINA, I.; HARROP, R.; SCHAEFER, C.; HO, C.; COSMINA, I.; HARROP, R.; SCHAEFER, C. ; HO, C.. **Introducing spring**. Pro Spring 5: An In-Depth Guide to the Spring Framework and Its Tools, p. 1–18, 2017.

[Coutinho *et al.* 2022] COUTINHO, D.; UCHÔA, A.; BARBOSA, C.; SOARES, V.; GARCIA, A.; SCHOTS, M.; PEREIRA, J. ; ASSUNÇAO, W. K.. **On the influential interactive factors on degrees of design decay: A multi-project study**. In: 2022 IEEE INTERNATIONAL CONFERENCE ON SOFTWARE ANALYSIS, EVOLUTION AND REENGINEERING (SANER), p. 753–764. IEEE, 2022.

[Curtis, Sappid and Szynkarski 2012] CURTIS, B.; SAPPIDI, J. ; SZYNKARSKI, A.. **Estimating the size, cost, and types of technical debt**. In: PROCEEDINGS OF THE THIRD INTERNATIONAL WORKSHOP ON MANAGING TECHNICAL DEBT, MTD '12, p. 49–53, Piscataway, NJ, USA, 2012. IEEE Press.

[De Padua et al. 2017] DE PADUA, G. B.; SHANG, W.. **Studying the prevalence of exception handling anti-patterns**. In: 2017 IEEE/ACM 25TH INTERNATIONAL CONFERENCE ON PROGRAM COMPREHENSION (ICPC), p. 328–331. IEEE, 2017.

[Dijkstra 1997] DIJKSTRA, E. W.. **A Discipline of Programming**. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 1997.

[Dubbo 2023] **dubbo/stubproxyfactorywrapper.java apache/dubbo**. `https://github.com/apache/dubbo/blob/0e66de1c6/dubbo-rpc/dubbo-rpc-api/src/main/java/org/apache/dubbo/rpc/proxy/wrapper/StubProxyFactoryWrapper.java`. (Accessed on 01/16/2023).

[Easterbrook *et al.* 2008] EASTERBROOK, S.; SINGER, J.; STOREY, M.-A. ; DAMIAN, D.. **Selecting Empirical Methods for Software Engineering Research**. Springer London, London, 2008.

[Ebert and Castor 2013] EBERT, F.; CASTOR, F.. **A study on developers' perceptions about exception handling bugs**. In: 2013 IEEE INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE, p. 448–451. IEEE, 2013.

[Ebert *et al.* 2015] EBERT, F.; CASTOR, F. ; SEREBRENIK, A.. **An exploratory study on exception handling bugs in java programs**. Journal of Systems and Software, 106:82–101, 2015.

[Ernst *et al.* 2015] ERNST, N.; BELLOMO, S.; OZKAYA, I.; NORD, R. ; GORTON, I.. **Measure it? manage it? ignore it? software practitioners and technical debt**. In: 10TH JOINT MEETING ON FOUNDATIONS OF SOFTWARE ENGINEERING, ESEC/FSE 2015, p. 50–60, New York, NY, USA, 2015. ACM.

[Fisher 1922] FISHER, R. A.. **On the interpretation of χ2 from contingency tables, and the calculation of p**. Journal of the Royal Statistical Society, p. 87–94, 1922.

[Fontana *et al.* 2019] FONTANA, F. A.; LENARDUZZI, V.; ROVEDA, R. ; TAIBI, D.. **Are architectural smells independent from code smells? an empirical study**. J. Syst. Softw., 154:139 – 156, 2019.

[Fowler 1999] FOWLER, M.. **Refactoring: Improving the Design of Existing Code**. Addison-Wesley Professional, Boston, 1999.

[Fresco 2023] **Buffereddiskcache.java at facebook/fresco**. `https://github.com/facebook/fresco/blob/cc75c3/imagepipeline/src/main/java/com/facebook/imagepipeline/cache/BufferedDiskCache.java`. (Accessed on 01/16/2023).

[Gamma *et al.* 1995] GAMMA, E.; HELM, R.; JOHNSON, R. ; VLISSIDES, J.. **Design Patterns: Elements of Reusable Object-oriented Software**. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[Garcia *et al.* 2009] GARCIA, J.; POPESCU, D.; EDWARDS, G. ; MEDVIDOVIC, N.. **Identifying architectural bad smells**. In: CSMR09; KAISERSLAUTERN, GERMANY. IEEE, 2009.

[Garcia *et al.* 2009b] GARCIA, J.; POPESCU, D.; EDWARDS, G. ; MEDVIDOVIC, N.. **Toward a catalogue of architectural bad smells**. In: Mirandola, R.; Gorton, I. ; Hofmeister, C., editors, ARCHITECTURES FOR

ADAPTIVE SOFTWARE SYSTEMS, p. 146–162, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[Garcia *et al.* 2013] GARCIA, J.; IVKOVIC, I. ; MEDVIDOVIC, N.. **A comparative analysis of software architecture recovery techniques**. In: PROCEEDINGS OF THE 28TH IEEE/ACM INTERNATIONAL CONFERENCE ON AUTOMATED SOFTWARE ENGINEERING; PALO ALTO, USA, 2013.

[Github 2022] GITHUB. **Github: Where the world builds software · github**. `https://github.com/`, 2022. (Accessed on 10/21/2022).

[Github Pull Requests 2023] GITHUB. **Available: `https://docs.github.com/en/github/collaborating-with-issues-and-pull-requests/about-pull-requests`**, 2023.

[Github Rest API 2022] GITHUB. **Github rest api - github docs**. `https://docs.github.com/en/rest`, 2022. (Accessed on 10/21/2022).

[Glinz 2007] GLINZ, M.. **On non-functional requirements**. In: 15TH IEEE INTERNATIONAL REQUIREMENTS ENGINEERING CONFERENCE (RE 2007), p. 21–26. IEEE, 2007.

[Godfrey and Lee 2000] GODFREY, M.; LEE, E.. **Secrets from the monster: Extracting Mozilla's software architecture**. In: COSET-00; LIMERICK, IRELAND, p. 15–23, 2000.

[González *et al.* 2021] GONZÁLEZ, C. A.; ZUMEL, L. A.; ACERO, J. A.; LENARDUZZI, V.; MARTÍNCZ-FERNÁNDEZ, S. ; RODRÍGUEZ, S. R.. **A preliminary investigation of developer profiles based on their activities and code quality: Who does what?** In: 2021 IEEE 21ST INTERNATIONAL CONFERENCE ON SOFTWARE QUALITY, RELIABILITY AND SECURITY (QRS), p. 938–945. IEEE, 2021.

[Goodenough 1975] GOODENOUGH, J. B.. **Exception handling: Issues and a proposed notation**. Communications of the ACM, 18(12):683–696, 1975.

[Gousios *et al.* 2014] GOUSIOS, G.; PINZGER, M. ; DEURSEN, A. V.. **An exploratory study of the pull-based software development model**. In: PROCEEDINGS OF THE 36TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, p. 345–355, 2014.

[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: 2015 IEEE/ACM 37TH IEEE INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, volumen 1, p. 358–368. IEEE, 2015.

[Gurp and Bosch 2002] VAN GURP, J.; BOSCH, J.. **Design erosion: problems and causes**. Journal of Systems and Software, 61(2):105 – 119, 2002.

[Handa *et al.* 2022] HANDA, N.; SHARMA, A. ; GUPTA, A.. **Framework for prediction and classification of non functional requirements: a novel vision**. Cluster Computing, 25(2):1155–1173, 2022.

[Hoskinson 2011] HOSKINSON, C.. Available: `http://www.politico.com/news/stories/0611/58051.html`, 2011.

[Hozano *et al.* 2017] HOZANO, M.; GARCIA, A.; ANTUNES, N.; FONSECA, B. ; COSTA, E.. **Smells are sensitive to developers! on the efficiency of (un)guided customized detection**. In: 2017 IEEE/ACM 25TH INTERNATIONAL CONFERENCE ON PROGRAM COMPREHENSION (ICPC), p. 110–120, 2017.

[Hozano *et al.* 2018] HOZANO, M.; GARCIA, A.; FONSECA, B. ; COSTA, E.. **Are you smelling it? investigating how similar developers detect code smells**. Inf. Softw. Technol., 93:130–146, 2018.

[IEEE 1990] COMMITTEE, I. S. C.; OTHERS. **Ieee standard glossary of software engineering terminology (ieee std 610.12-1990). los alamitos**. CA: IEEE Computer Society, 169, 1990.

[ISO-IEC 25010 2011] INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. **ISO-IEC 25010: 2011 Systems and Software Engineering-Systems and Software Quality Requirements and Evaluation (SQuaRE)-System and Software Quality Models**. ISO, 2011.

[Jakobus *et al.* 2015] JAKOBUS, B.; BARBOSA, E. A.; GARCIA, A. ; DE LUCENA, C. J. P.. **Contrasting exception handling code across languages: An experience report involving 50 open source projects**. In: 2015 IEEE 26TH INTERNATIONAL SYMPOSIUM ON SOFTWARE RELIABILITY ENGINEERING (ISSRE), p. 183–193. IEEE, 2015.

[Jiang *et al.* 2021] JIANG, H.; ZHU, J.; YANG, L.; LIANG, G. ; ZUO, C.. **Deeprelease: Language-agnostic release notes generation from pull**

requests of open-source software. In: 2021 28TH ASIA-PACIFIC SOFTWARE ENGINEERING CONFERENCE (APSEC), p. 101–110. IEEE, 2021.

[Jindal 2021] JINDAL, R.; MALHOTRA, R.; JAIN, A. ; BANSAL, A.. **Mining non-functional requirements using machine learning techniques**. e-Informatica Software Engineering Journal, 15(1), 2021.

[Joblin *et al.* 2017] JOBLIN, M.; APEL, S.; HUNSEN, C. ; MAUERER, W.. **Classifying developers into core and peripheral: An empirical study on count and network metrics**. In: 2017 IEEE/ACM 39TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE), p. 164–174. IEEE, 2017.

[Kalliamvakou *et al.* 2016] KALLIAMVAKOU, E.; GOUSIOS, G.; BLINCOE, K.; SINGER, L.; GERMAN, D. M. ; DAMIAN, D.. **An in-depth study of the promises and perils of mining GitHub**. Empirical Software Engineering, 21(5):2035–2071, 2016.

[Kaminski 2007] KAMINSKI, P.. **Reforming software design documentation**. In: 14TH WORKING CONFERENCE ON REVERSE ENGINEERING (WCRE 2007), p. 277–280, Oct 2007.

[Kaur 2022] KAUR, K.; KAUR, P.. **Sabdm: A self-attention based bidirectional-rnn deep model for requirements classification**. Journal of Software: Evolution and Process, p. e2430, 2022.

[Kechagia and Spinellis 2014] KECHAGIA, M.; SPINELLIS, D.. **Undocumented and unchecked: exceptions that spell trouble**. In: PROCEEDINGS OF THE 11TH WORKING CONFERENCE ON MINING SOFTWARE REPOSITORIES, p. 312–315, 2014.

[Kery et al. 2016] KERY, M. B.; LE GOUES, C. ; MYERS, B. A.. **Examining programmer practices for locally handling exceptions**. In: 2016 IEEE/ACM 13TH WORKING CONFERENCE ON MINING SOFTWARE REPOSITORIES (MSR), p. 484–487. IEEE, 2016.

[Kurtanović and Maalej 2017] KURTANOVIĆ, Z.; MAALEJ, W.. **Automatically classifying functional and non-functional requirements using supervised machine learning**. In: 2017 IEEE 25TH INTERNATIONAL REQUIREMENTS ENGINEERING CONFERENCE (RE), p. 490–495. Ieee, 2017.

[Lanza and Marinescu 2006] LANZA, M.; MARINESCU, R.. **Object-Oriented Metrics in Practice**. Springer, Heidelberg, 2006.

[Lausen 2002] LAUESEN, S.. **Software requirements: styles and techniques**. Pearson Education, 2002.

[Lazard *et al.* 2017] LAZAR, J.; FENG, J. ; HOCHHEISER, H.. **Research methods in human-computer interaction**. Morgan Kaufmann, 2017.

[Lee *et al.* 1990] LEE, P. A.; ANDERSON, T.; LEE, P. A. ; ANDERSON, T.. **Fault tolerance**. Springer, 1990.

[Li *et al.* 2014] LI, Z.; LIANG, P.; AVGERIOU, P.; GUELFI, N. ; AMPATZOGLOU, A.. **An empirical investigation of modularity metrics for indicating architectural technical debt**. In: PROCEEDINGS OF THE 10TH INTERNATIONAL ACM SIGSOFT CONFERENCE ON QUALITY OF SOFTWARE ARCHITECTURES, QoSA '14, p. 119–128, New York, NY, USA, 2014. ACM.

[Li *et al.* 2015] LI, Z.; AVGERIOU, P. ; LIANG, P.. **A systematic mapping study on technical debt and its management**. Journal of Systems and Software, 101:193–220, 2015.

[Lim *et al.* 2012] LIM, E.; TAKSANDE, N. ; SEAMAN, C.. **A balancing act: What software practitioners have to say about technical debt**. IEEE software, 29(6):22–27, 2012.

[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 2016 24TH ACM SIGSOFT INTERNATIONAL SYMPOSIUM ON FOUNDATIONS OF SOFTWARE ENGINEERING, FSE 2016, p. 535–546, New York, NY, USA, 2016. Association for Computing Machinery.

[Linaker *et al.* 2015] LINAKER, J.; SULAMAN, S. M.; HÖST, M. ; DE MELLO, R. M.. **Guidelines for conducting surveys in software engineering v. 1.1**. 2015.

[Lippert 2006] LIPPERT, M.; ROOCK, S.. **Refactoring in Large Software Projects: Performing Complex Restructurings Successfully**. Wiley, 2006.

[Liu *et al.* 2019] LIU, Z.; XIA, X.; TREUDE, C.; LO, D. ; LI, S.. **Automatic generation of pull request descriptions**. In: 2019 34TH IEEE/ACM

INTERNATIONAL CONFERENCE ON AUTOMATED SOFTWARE ENGINEERING (ASE), p. 176–188. IEEE, 2019.

[MacCormack, Rusnak and Baldwin 2006] MACCORMACK, A.; RUSNAK, J. ; BALDWIN, C.. **Exploring the structure of complex software designs: An empirical study of open source and proprietary code**. Manage. Sci., 52(7):1015–1030, 2006.

[MacQueen 1967] MACQUEEN, J.. **Classification and analysis of multivariate observations**. In: 5TH BERKELEY SYMP. MATH. STATIST. PROBABILITY, p. 281–297, 1967.

[Macia 2013] MACIA, I.. **On the Detection of Architecturally-Relevant Code Anomalies in Software Systems**. PhD thesis, Pontifical Catholic University of Rio de Janeiro, Informatics Department, 2013.

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

[Macia *et al.* 2012a] MACIA, I.; ARCOVERDE, R.; CIRILO, E.; GARCIA, A. ; VON STAA, A.. **Supporting the identification of architecturally-relevant code anomalies**. In: ICSM12, p. 662–665, Sept 2012.

[Macia *et al.* 2012b] MACIA, I.; GARCIA, J.; POPESCU, D.; GARCIA, A.; MEDVIDOVIC, N. ; VON STAA, A.. **Are automatically-detected code anomalies relevant to architectural modularity?: An exploratory analysis of evolving systems**. In: AOSD '12, p. 167–178, New York, NY, USA, 2012. ACM.

[Martin and Martin 2006] MARTIN, R. C.; MARTIN, M.. **Agile Principles, Patterns, and Practices in C# (Robert C. Martin)**. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2006.

[Martins 2020] MARTINS, J.; BEZERRA, C.; UCHÔA, A. ; GARCIA, A.. **Are code smell co-occurrences harmful to internal quality attributes? a mixed-method study**. In: 34TH BRAZILIAN SYMPOSIUM ON SOFTWARE ENGINEERING, SBES '20, p. 52–61, New York, NY, USA, 2020. ACM.

[McHugh 2013] MCHUGH, M.. **The chi-square test of independence**. Biochemia medica, 23(2):143–149, 2013.

[Melo *et al.* 2019] MELO, H.; COELHO, R. ; TREUDE, C.. **Unveiling exception handling guidelines adopted by java developers**. In: 2019 IEEE 26TH INTERNATIONAL CONFERENCE ON SOFTWARE ANALYSIS, EVOLUTION AND REENGINEERING (SANER), p. 128–139. IEEE, 2019.

[Moha, Gueheneuc and Leduc 2006] MOHA, N.; G. GUEHENEUC, Y. ; LEDUC, P.. **Automatic generation of detection algorithms for design defects**. In: 21ST IEEE/ACM INTERNATIONAL CONFERENCE ON AUTOMATED SOFTWARE ENGINEERING (ASE'06), p. 297–300, Sept 2006.

[Moha *et al.* 2010] MOHA, N.; GUEHENEUC, Y.; DUCHIEN, L. ; MEUR, A. L.. **Decor: A method for the specification and detection of code and design smells**. IEEE Transaction on Software Engineering, 36:20–36, 2010.

[Mohammed and Alemneh 2021] MOHAMMED, E.; ALEMNEH, E.. **Identification of architecturally significant non-functional requirement**. In: 2021 INTERNATIONAL CONFERENCE ON INFORMATION AND COMMUNICATION TECHNOLOGY FOR DEVELOPMENT FOR AFRICA (ICT4DA), p. 24–29. IEEE, 2021.

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

[Nakshatri et al. 2016] NAKSHATRI, S.; HEGDE, M. ; THANDRA, S.. **Analysis of exception handling patterns in java projects: An empirical study**. In: 2016 IEEE/ACM 13TH WORKING CONFERENCE ON MINING SOFTWARE REPOSITORIES (MSR), p. 500–503. IEEE, 2016.

[Noll and Liu 2010] NOLL, J.; LIU, W.-M.. **Requirements elicitation in open source software development: a case study**. In: PROCEEDINGS OF THE 3RD INTERNATIONAL WORKSHOP ON EMERGING TRENDS IN FREE/LIBRE/OPEN SOURCE SOFTWARE RESEARCH AND DEVELOPMENT, p. 35–40, 2010.

[Oizumi *et al.* 2015] OIZUMI, W.; GARCIA, A.; COLANZI, T.; STAA, A. ; FERREIRA, M.. **On the relationship of code-anomaly agglomerations**

and architectural problems. Journal of Software Engineering Research and Development, 3(1):1–22, 2015.

[Oizumi *et al.* 2016] OIZUMI, W.; GARCIA, A.; SOUSA, L.; CAFEO, B. ; ZHAO, Y.. **Code anomalies flock together: Exploring code anomaly agglomerations for locating design problems**. In: THE 38TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING; USA, 2016.

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

[Oizumi *et al.* 2019] OIZUMI, W.; SOUSA, L.; OLIVEIRA, A.; CARVALHO, L.; GARCIA, A.; COLANZI, T. ; OLIVEIRA, R.. **On the density and diversity of degradation symptoms in refactored classes: A multi-case study**. In: 2019 IEEE 30TH INTERNATIONAL SYMPOSIUM ON SOFTWARE RELIABILITY ENGINEERING (ISSRE), p. 346–357. IEEE, 2019.

[Oizumi *et al.* 2020] OIZUMI, W.; BIBIANO, A. C.; CEDRIM, D.; OLIVEIRA, A.; SOUSA, L.; GARCIA, A. ; OLIVEIRA, D.. **Recommending composite refactorings for smell removal: Heuristics and evaluation**. In: PROCEEDINGS OF THE XXXIV BRAZILIAN SYMPOSIUM ON SOFTWARE ENGINEERING, p. 72–81, 2020.

[Oliveira, Valente and Terra 2016] SILVA, M. C. O.; 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 '16, p. 248–254, 2016.

[Oliveira *et al.* 2016] OLIVEIRA, J.; CACHO, N.; BORGES, D.; SILVA, T. ; CASTOR, F.. **An exploratory study of exception handling behavior in evolving android and java applications**. In: PROCEEDINGS OF THE XXX BRAZILIAN SYMPOSIUM ON SOFTWARE ENGINEERING, p. 23–32, 2016.

[Oliveira *et al.* 2019] OLIVEIRA, A.; SOUSA, L.; OIZUMI, W. ; GARCIA, A.. **On the prioritization of design-relevant smelly elements: A mixed-method, multi-project study**. In: PROCEEDINGS OF THE

XIII BRAZILIAN SYMPOSIUM ON SOFTWARE COMPONENTS, ARCHITECTURES, AND REUSE, SBCARS '19, p. 83–92, New York, NY, USA, 2019. Association for Computing Machinery.

[Oliveira *et al.* 2022] OLIVEIRA, A.; OIZUMI, W.; SOUSA, L.; ASSUNÇÃO, W. K.; GARCIA, A.; LUCENA, C. ; CEDRIM, D.. **Smell patterns as indicators of design degradation: Do developers agree?** In: PROCEEDINGS OF THE XXXVI BRAZILIAN SYMPOSIUM ON SOFTWARE ENGINEERING, p. 311–320, 2022.

[Oliveira *et al.* 2023] OLIVEIRA, A.; CORREIA, J. L.; SOUSA, L. D. S.; ASSUNÇÃO, W, K.; COUTINHO, D.; GARCIA, A.; OIZUMI, W.; BARBOSA, C.; UCHÔA, A. ; ALVES PEREIRA, J.. **Don't forget the exception! considering robustness changes to identify design problems**. In: PROCEEDINGS OF THE 20TH INTERNATIONAL CONFERENCE ON MINING SOFTWARE REPOSITORIES, 2023.

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

[Parnas 1972] PARNAS, D. L.. **On the criteria to be used in decomposing systems into modules**. Commun. ACM, 15(12):1053–1058, Dec. 1972.

[Parnas 1978] PARNAS, D. L.. **Designing software for ease of extension and contraction**. In: PROCEEDINGS OF THE 3RD INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, ICSE '78, p. 264–277, Piscataway, NJ, USA, 1978. IEEE Press.

[Perry and Wolf 1992] PERRY, D. E.; WOLF, A. L.. **Foundations for the study of software architecture**. SIGSOFT Softw. Eng. Notes, 17(4):40–52, Oct. 1992.

[Rachow 2019] RACHOW, P.. **Refactoring decision support for developers and architects based on architectural impact**. In: 2019 IEEE INTERNATIONAL CONFERENCE ON SOFTWARE ARCHITECTURE COMPANION (ICSA-C), p. 262–266, March 2019.

[Ran *et al.* 2015] MO, R.; CAI, Y.; KAZMAN, R. ; XIAO, L.. **Hotspot patterns: The formal definition and automatic detection of architecture smells**. In: SOFTWARE ARCHITECTURE (WICSA), 2015 12TH WORKING IEEE/IFIP CONFERENCE ON, p. 51–60, May 2015.

[Rastogi and Nagappan 2016] RASTOGI, A.; NAGAPPAN, N.. **On the personality traits of github contributors**. In: 2016 IEEE 27TH INTERNATIONAL SYMPOSIUM ON SOFTWARE RELIABILITY ENGINEERING (ISSRE), p. 77–86. IEEE, 2016.

[Reimer and Harini 2003] REIMER, D.; SRINIVASAN, H.. **Analyzing exception usage in large java applications**. In: PROCEEDINGS OF ECOOP'2003 WORKSHOP ON EXCEPTION HANDLING IN OBJECT-ORIENTED SYSTEMS, p. 10–18, 2003.

[Rizzi *et al.* 2018] RIZZI, L.; FONTANA, F. A. ; ROVEDA, R.. **Support for architectural smell refactoring**. In: PROCEEDINGS OF THE 2ND INTERNATIONAL WORKSHOP ON REFACTORING, IWoR 2018, p. 7–10, New York, NY, USA, 2018. Association for Computing Machinery.

[Robillard 2000] ROBILLARD, M. P.; MURPHY, G. C.. **Designing robust java programs with exceptions**. In: PROCEEDINGS OF THE 8TH ACM SIGSOFT INTERNATIONAL SYMPOSIUM ON FOUNDATIONS OF SOFTWARE ENGINEERING: TWENTY-FIRST CENTURY APPLICATIONS, p. 2–10, 2000.

[Robiolo *et al.* 2019] ROBIOLO, G.; SCOTT, E.; MATALONGA, S. ; FELDERER, M.. **Technical debt and waste in non-functional requirements documentation: An exploratory study**. In: PRODUCT-FOCUSED SOFTWARE PROCESS IMPROVEMENT: 20TH INTERNATIONAL CONFERENCE, PROFES 2019, BARCELONA, SPAIN, NOVEMBER 27–29, 2019, PROCEEDINGS 20, p. 220–235. Springer, 2019.

[Rocha et al. 2018] ROCHA, J.; MELO, H.; COELHO, R. ; SENA, B.. **Towards a catalogue of java exception handling bad smells and refactorings**. In: PROCEEDINGS OF THE 25TH CONFERENCE ON PATTERN LANGUAGES OF PROGRAMS, p. 1–17, 2018.

[Rousseeuw 1987] ROUSSEEUW, P. J.. **Silhouettes: a graphical aid to the interpretation and validation of cluster analysis**. Journal of computational and applied mathematics, 20:53–65, 1987.

[RxJava 2023] **Rxjava/flowableflatmap.java at 2572fa74ab93c4f3ffe0ea51932eecc180873904 · reactivex/rxjava · github**. `https://github.com/ReactiveX/RxJava/blob/2572fa74ab93c4f3ffe0ea51932eecc180873904/src/main/java/io/reactivex/internal/operators/flowable/FlowableFlatMap.java`. (Accessed on 01/11/2023).

[Saadatmand *et al.* 2012] SAADATMAND, M.; CICCHETTI, A. ; SJÖDIN, M.. **Toward model-based trade-off analysis of non-functional requirements**. In: 2012 38TH EUROMICRO CONFERENCE ON SOFTWARE ENGINEERING AND ADVANCED APPLICATIONS, p. 142–149. IEEE, 2012.

[Scacchi 2009] SCACCHI, W.. **Understanding requirements for open source software**. In: DESIGN REQUIREMENTS ENGINEERING: A TEN-YEAR PERSPECTIVE: DESIGN REQUIREMENTS WORKSHOP, CLEVELAND, OH, USA, JUNE 3-6, 2007, REVISED AND INVITED PAPERS, p. 467–494. Springer, 2009.

[Schach *et al.* 2002] SCHACH, S.; JIN, B.; WRIGHT, D.; HELLER, G. ; OFFUTT, A.. **Maintainability of the linux kernel**. Software, IEE Proceedings -, 149(1):18–23, 2002.

[Shadish, Cook and Campbell 2001] SHADISH, W. R.; COOK, T. D. ; CAMPBELL, D. T.. **Experimental and Quasi-Experimental Designs for Generalized Causal Inference**. Houghton Mifflin, 2 edition, 2001.

[Shah *et al.* 2010] SHAH, H.; GORG, C. ; HARROLD, M. J.. **Understanding exception handling: Viewpoints of novices and experts**. IEEE Transactions on Software Engineering, 36(2):150–161, 2010.

[Sharma and Spinellis 2018] SHARMA, T.; SPINELLIS, D.. **A survey on software smells**. J. Syst. Softw., 138:158 – 173, 2018.

[Sharma *et al.* 2020] SHARMA, T.; SINGH, P. ; SPINELLIS, D.. **An empirical investigation on the relationship between design and architecture smells**. Empirical Software Engineering, 25(5):4020–4068, 2020.

[Shaw and Mitchell-Olds 1993] SHAW, R.; MITCHELL-OLDS, T.. **Anova for unbalanced data: an overview**. Ecology, 74(6):1638–1645, 1993.

[Sheskin 2003] SHESKIN, D. J.. **Handbook of parametric and nonparametric statistical procedures**. Chapman and Hall/CRC, 2003.

[Silva *et al.* 2016] SILVA, M. C. O.; VALENTE, M. T. ; TERRA, R.. **Does technical debt lead to the rejection of pull requests?** arXiv preprint arXiv:1604.01450, 2016.

[Slankas and Willians 2013] SLANKAS, J.; WILLIAMS, L.. **Automated extraction of non-functional requirements in available documentation**. In: 2013 1ST INTERNATIONAL WORKSHOP ON NATURAL

LANGUAGE ANALYSIS IN SOFTWARE ENGINEERING (NATURALISE), p. 9–16. IEEE, 2013.

[Soares *et al.* 2015] SOARES, D. M.; DE LIMA JÚNIOR, M. L.; MURTA, L. ; PLASTINO, A.. **Acceptance factors of pull requests in open-source projects**. In: 30TH ANNUAL ACM SYMPOSIUM ON APPLIED COMPUTING, p. 1541–1546, 2015.

[Sousa 2018] SOUSA, L.. **Understanding How Developers Identify Design Problems in Practice**. PhD thesis, Ph. D. dissertation, Informatics Department, PUC-Rio, Brazil, 2018.

[Sousa *et al.* 2017] SOUSA, L.; OLIVEIRA, R.; GARCIA, A.; LEE, J.; CONTE, T.; OIZUMI, W.; DE MELLO, R.; LOPES, A.; VALENTIM, N.; OLIVEIRA, E. ; LUCENA, C.. **How do software developers identify design problems?: A qualitative analysis**. In: PROCEEDINGS OF 31ST BRAZILIAN SYMPOSIUM ON SOFTWARE ENGINEERING, SBES'17, 2017.

[Sousa *et al.* 2018] SOUSA, L.; OLIVEIRA, A.; OIZUMI, W.; BARBOSA, S.; GARCIA, A.; LEE, J.; KALINOWSKI, M.; DE MELLO, R.; FONSECA, B.; OLIVEIRA, R.; LUCENA, C. ; PAES, R.. **Identifying design problems in the source code: A grounded theory**. In: PROCEEDINGS OF THE 40TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, ICSE '18, p. 921–931, New York, NY, USA, 2018. ACM.

[Sousa *et al.* 2020] SOUSA, L.; OIZUMI, W.; GARCIA, A.; OLIVEIRA, A.; CEDRIM, D. ; LUCENA, C.. **When are smells indicators of architectural refactoring opportunities: A study of 50 software projects**. In: PROCEEDINGS OF THE 28TH INTERNATIONAL CONFERENCE ON PROGRAM COMPREHENSION, ICPC '20, p. 354–365, New York, NY, USA, 2020. Association for Computing Machinery.

[Spring, 2022] SPRING. **Spring | home**. `https://spring.io/`, 2022. (Accessed on 10/21/2022).

[Spring Boot 2022] GITHUB. **Optimize cachekey handling in springiterableconfigurationpropertysource by dreis2211 · pull request #16717 · spring-projects/spring-boot · github**. `https://github.com/spring-projects/spring-boot/pull/16717`, 2019. (Accessed on 10/21/2022).

[Spring Community 2022] SPRING. **Spring | community**. `https://spring.io/community`, 2022. (Accessed on 10/21/2022).

[Spring Framework 2022] GITHUB. **Avoid unnecessary sorting overhead by dreis2211 · pull request #24617 · spring-projects/spring-framework · github**. `https://github.com/spring-projects/spring-framework/pull/24617`, 2020. (Accessed on 10/21/2022).

[Spring Team 2022] SPRING. **Spring | team**. `https://spring.io/team`, 2022. (Accessed on 10/21/2022).

[Stal 2014] STAL, M.. **Chapter 3 - refactoring software architectures**. In: Babar, M. A.; Brown, A. W. ; Mistrik, I., editors, AGILE SOFTWARE ARCHITECTURE, p. 63 – 82. Morgan Kaufmann, Boston, 2014.

[Tang *et al.* 2010] TANG, A.; ALETI, A.; BURGE, J. ; VAN VLIET, H.. **What makes software design effective?** Design Studies, 31(6):614 – 640, 2010. Special Issue Studying Professional Software Design.

[Taylor et al. 2009] TAYLOR, R.; MEDVIDOVIC, N. ; DASHOFY, E.. **Software Architecture: Foundations, Theory, and Practice**. Wiley Publishing, 2009.

[Tóth, 2019] TÓTH, L.; VIDÁCS, L.. **Comparative study of the performance of various classifiers in labeling non-functional requirements**. Information Technology and Control, 48(3):432–445, 2019.

[Trifu and Marinescu 2005] TRIFU, A.; MARINESCU, R.. **Diagnosing design problems in object oriented systems**. In: WCRE'05, p. 10 pp., Nov 2005.

[Tsantalis *et al.* 2018] TSANTALIS, N.; MANSOURI, M.; ESHKEVARI, L. M.; MAZINANIAN, D. ; DIG, D.. **Accurate and efficient refactoring detection in commit history**. In: PROCEEDINGS OF THE 40TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, ICSE '18, p. 483–494, New York, NY, USA, 2018. ACM.

[Tufano *et al.* 2015] 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**. In: PROCEEDINGS OF THE 37TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, ICSE '15, New York, NY, USA, 2015. ACM.

[Uchôa et al. 2020] UCHÔA, A.; BARBOSA, C.; OIZUMI, W.; BLENÍLIO, P.; LIMA, R.; GARCIA, A. ; BEZERRA, C.. **How does modern code review impact software design degradation? an in-depth empirical**

study. In: 2020 IEEE INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE AND EVOLUTION (ICSME), p. 511–522. IEEE, 2020.

[Vidal *et al.* 2016] VIDAL, S.; GUIMARAES, E.; OIZUMI, W.; GARCIA, A.; PACE, A. D. ; MARCOS, C.. **Identifying architectural problems through prioritization of code smells**. In: SBCARS16, p. 41–50, Sept 2016.

[Wang *et al.* 2018] WANG, C.; ZHANG, F.; LIANG, P.; DANEVA, M. ; VAN SINDEREN, M.. **Can app changelogs improve requirements classification from app reviews? an exploratory study**. In: PROCEEDINGS OF THE 12TH ACM/IEEE INTERNATIONAL SYMPOSIUM ON EMPIRICAL SOFTWARE ENGINEERING AND MEASUREMENT, p. 1–4, 2018.

[Weimer and Necula 2008] WEIMER, W.; NECULA, G. C.. **Exceptional situations and program reliability**. ACM Transactions on Programming Languages and Systems (TOPLAS), 30(2):1–51, 2008.

[Xiao, Cai and Kazman 2014] XIAO, L.; CAI, Y. ; KAZMAN, R.. **Design rule spaces: A new form of architecture insight**. In: PROCEEDINGS OF THE 36TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, p. 967–977, New York, NY, USA, 2014. ACM.

[Yamashita and Moonen 2012] YAMASHITA, A.; MOONEN, L.. **Do code smells reflect important maintainability aspects?** In: ICSM12, p. 306–315, 2012.

[Yamashita and Moonen 2013] YAMASHITA, A.; MOONEN, L.. **Exploring the impact of inter-smell relations on software maintainability: an empirical study**. In: PROCEEDINGS OF THE 35TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING; SAN FRANCISCO, USA, p. 682–691, 2013.

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

[Yin 2015] YIN, R.. **Qualitative research from start to finish**. Guilford publications, 2015.

[Zanaty *et al.* 2018] ZANATY, F. E.; HIRAO, T.; MCINTOSH, S.; IHARA, A. ; MATSUMOTO, K.. **An empirical study of design discussions in**

code review. In: PROCEEDINGS OF THE 12TH ACM/IEEE INTERNATIONAL SYMPOSIUM ON EMPIRICAL SOFTWARE ENGINEERING AND MEASUREMENT, p. 1–10, 2018.

[Zimmermann 2015] ZIMMERMANN, O.. **Architectural refactoring: A task-centric view on software evolution**. IEEE Software, 32(02):26–29, mar 2015.

[Zou *et al.* 2017] ZOU, J.; XU, L.; YANG, M.; ZHANG, X. ; YANG, D.. **Towards comprehending the non-functional requirements through developers' eyes: An exploration of stack overflow using topic analysis**. Information and Software Technology, 84:19–32, 2017.

[de Lemos and Romanovsky 2001] DE LEMOS, R.; ROMANOVSKY, A.. **Exception handling in the software lifecycle**. International Journal of Computer Systems Science and Engineering, 16(2):167–181, 2001.

[de Mello *et al.* 2023] DE MELLO, R.; OLIVEIRA, R.; UCHÔA, A.; OIZUMI, W.; GARCIA, A.; FONSECA, B. ; DE MELLO, F.. **Recommendations for developers identifying code smells**. IEEE Software, 40(2):90–98, 2023.

[de Souza *et al.* 2005] DE SOUZA, S. C. B.; ANQUETIL, N. ; DE OLIVEIRA, K. M.. **A study of the documentation essential to software maintenance**. In: PROCEEDINGS OF THE 23RD ANNUAL INTERNATIONAL CONFERENCE ON DESIGN OF COMMUNICATION: DOCUMENTING & DESIGNING FOR PERVASIVE INFORMATION, p. 68–75, 2005.

A
Study on the Identification of Design Problems Using Maintainability Smells Patterns

In this appendix, we present details of our first study (see Chapter 3). The following subsections describe the study *(i)* refactorings considered, *(ii)* hypothesis and variables, *(iii)* developers' characterization, *(iv)* summary of developers' answers, *(v)* automated tool details, and *(vi)* forms applied.

A.1
Common Refactorings Applied for Specific Maintainability Smells

The full list of common refactorings for each maintainability smell explored in Chapter 3 is detailed as follows. In Table A.1, we present the description of each refactoring. In Table A.2, we present the maintainability smell and the respective refactorings that could be applied to remove them.

A.2
Hypothesis and Variables

Table A.1: Refactorings and their Descriptions

Refactoring	Description
Collapse Hierarchy	A class hierarchy, in which the subclass is almost identical to its superclass. We merge bloth classes.
Extract Class	One class has the responsibilities of two. We create a new class responsible for the relevant functionalities, making both classes have single responsibilities.
Extract Field	Expression is hard to understand. We place the result of the expression is in separate fields.
Extract Method	A code fragment can be grouped together. We move this code to a new method and replace the old code to a call to the method.
Move Field	A field is used more in another class rather than its own class. We create a field in a new class and put the old fields in this new class.
Move Method	A method is used more in another class rather than its own class. We move this method to the right class and move the code from the original method to the new one.
Inline Class	A class that does not have any responsibility, performing no functionalities. We move all fields and methods from this class to a new one.
Rename Method	The current method name do not explain what the method does. We change the method name to fit its actions.
Remove Parameter	A parameter is not used in the method body. We remove this parameter.

Table A.2: Common Refactorings Applied for Each Maintainability Smell

Maintainability Smell	**Common Refactoring**
Complex Class	Extract Method, Move Method, Extract Class
Dispersed Coupling	Extract Method
Feature Envy	Move Method, Move Field, Extract Field
God Class	Extract Class, Move Method, Move Field
Intensive Coupling	Move Method, Extract Method
Lazy Class	Inline Class, Collapse Hierarchy
Lazy Method	Extract Method
Shotgun Surgery	Move Method, Move Field, Inline Class
Speculative Generality	Collapse Hierarchy, Inline Class, Remove Parameter, Rename Method

A.2.1 Hypothesis RQ. 1

For our first research question (***RQ. 1****: Are Smell Patterns Indicators of Degradation?*), we formulated the following hypotheses:

Null Hypothesis:

– $\mathcal{H}_0$: "Smell patterns do not indicate degradation."

Alternative Hypotheses:

– $\mathcal{H}_1$: "Single Smell Patterns are indicators of degradation."

– $\mathcal{H}_2$: "Multiple Smell Patterns are indicators of degradation."

A.2.2 Hypothesis RQ. 2

For our second research question (***RQ. 2****: Can Smell-Patterns Indicate Refactoring Opportunities?*), we formulated the following hypotheses:

Null Hypothesis:

– $\mathcal{H}_0$: "Smell patterns can not indicate refactoring opportunities."

Alternative Hypotheses:

– $\mathcal{H}_1$: "Single Smell Patterns can indicate refactoring opportunities."

– $\mathcal{H}_2$: "Multiple Smell Patterns can indicate refactoring opportunities."

A.2.3 Independent Variables

The *independent variables* in our study are:

- The characteristics of the software systems under investigation.
- The level of familiarity with these software systems among the developers.
- The use of single smell patterns (SSP) in the analysis.
- The use of multiple smell patterns (MSP) in the analysis.
- The use of multiple combinations of smell patterns.

A.2.4 Dependent Variables

The *dependent variables* in our research are:

- Presence of Degradation (***RQ. 1***).
- Acceptance of Refactoring Opportunities (***RQ. 2*** and ***RQ. 3***).

A.3 Developers' Characterization

We recruited the developers of this quasi-experiment through our network of contacts in the industry and other research groups. We also looked for potential developers in our professional social media (Twitter and LinkedIn). We defined the following criteria to select the final list of developers:

1. Intermediary knowledge of Java programming language.
2. Intermediary knowledge about software architecture.
3. Basic knowledge about code smells and refactoring.
4. More than one year of experience with software development.

The developers themselves informed such characteristics through a characterization form. We did not conduct any tests to verify the experience and level of knowledge on each topic. Table A.3 contains information on the selected developers for this study.

Table A.3: Characteristics of Developers from Study One

ID	Programming Experience	Java Knowledge	Architecture Knowledge	Code Smells Knowledge	Refactoring Knowledge
1	6 to 10 years	Specialist	Experienced	Experienced	Experienced
2	1 to 5 years	Intermediate	Intermediate	Experienced	Experienced
3	6 to 10 years	Intermediate	Intermediate	Experienced	Specialist
4	6 to 10 years	Intermediate	Intermediate	Intermediate	Intermediate
5	6 to 10 years	Intermediate	Intermediate	Basic	Basic
6	6 to 10 years	Experienced	Intermediate	Intermediate	Intermediate
7	6 to 10 years	Intermediate	Intermediate	Basic	Intermediate
8	1 to 5 years	Intermediate	Intermediate	Experienced	Intermediate
9	6 to 10 years	Experienced	Experienced	Experienced	Experienced
10	1 to 5 years	Intermediate	Intermediate	Experienced	Experienced
11	11 to 15 years	Specialist	Experienced	Experienced	Experienced
12	1 to 5 years	Experienced	Specialist	Experienced	Experienced
13	6 to 10 years	Experienced	Intermediate	Experienced	Experienced

Table A.4: Summary of Developers' Answers

Participant ID	Agreed With the Presence of a Problem	Is Implementation Level	Is Architectural Level	Agreed With the Refactorings	Partially Agreed With the Refactorings
1	6	5	1	6	0
2	4	3	1	0	4
3	1	0	1	1	0
4	6	3	3	1	4
5	4	0	4	4	0
6	5	4	1	3	2
7	4	2	2	3	1
8	5	1	4	2	3
9	2	2	0	0	0
10	3	2	1	0	1
11	3	3	0	0	2
12	4	0	4	2	2
13	2	2	0	2	0

A.4 Summary of Developers' Answers

Table A.4 presents a summary of developers' answers. 1st column shows the developers' ID. 2nd column presents how many times (from the 6 cases) the developer agreed with the presence of a degradation problem. 3rd and 4th columns show how often the developer considered the problem to be on an implementation or architectural level. 5th and 6th columns show how often developers agreed or partially agreed with the proposed refactoring.

A.5 Automated Tool

We developed a tool to be used by the developers to identify design problems within their systems. In this tool, the developers use as input their project source code. The tool analyzes and automatically collects the

maintainability smells and metrics. In Figure A.1, we show a screenshot of the tool with numbered parts.

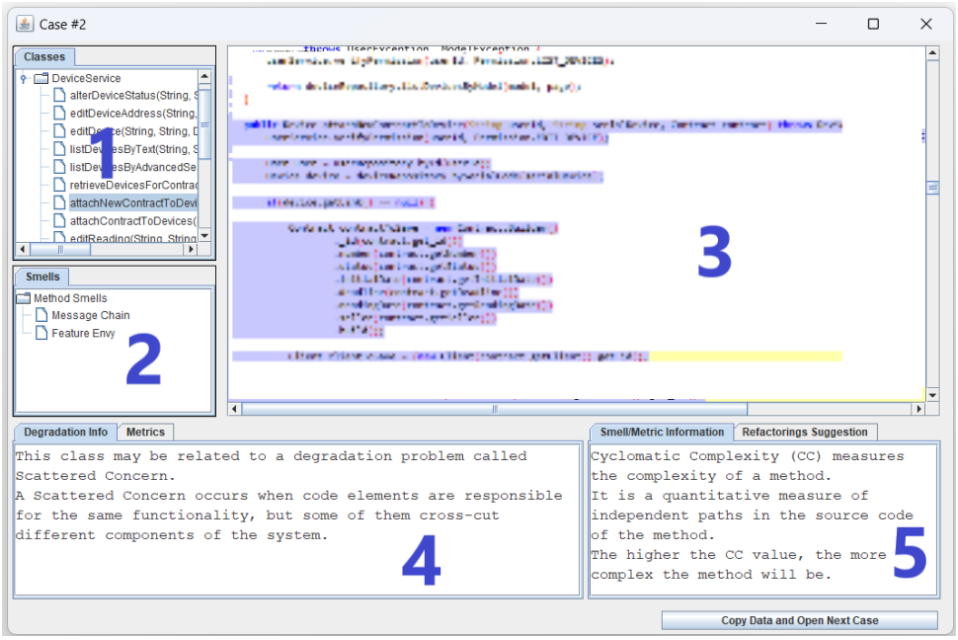

Figure A.1: Screenshot of the Tool Used by the Developers to Identify Design Problems

Following, we describe each part.

- **Part 1:** Presents the components' source code tree. The developer can navigate through the files of this tree and evaluate each source code.
- **Part 2:** Presents the maintainability smells affecting the current file analyzed by the developer.
- **Part 3:** Presents the source code of the current file analyzed.
- **Part 4:** It has two sections. The 1st section (Degradation Info) provides a description of the design problem that may be affecting the code analyzed. The 2nd section (Metrics) provides a list of metrics of the source code analyzed (*e.g.* lines of code in the method and coupling intensity)
- **Part 5:** It has two sections. The 1st section (Smell/Metric Information), presents the description of the maintainability smell or metric selected in the tool. The 2nd section (Refactoring Suggestions) presents the refactoring suggestions to remove the design problem.

A.6 Forms

A.6.1 Characterization Form

Following, we present the form filled by the developer prior to their task of identifying design problems.

Characterization and Consent Form

The purpose of this form is to characterize and collect the consent of people participating in the study on structural software degradation carried out by researchers from PUC-Rio and Carnegie Mellon University.
The identity of the participants is confidential and, therefore, will not be stored or disclosed in any way.
This study causes little to no discomfort or stress in the participants. Although the forms have mandatory fields, the participant can choose to fill any field with the following text or option: "I prefer not to inform".
In this study, we will use a software for static analysis of source code. This software does not make any changes to the analyzed code, nor does it send any type of information to us.
The participating person will have full control over the information that will be collected during the study, as all of it will be filled out and sent through a web form.
At the end of the study, we will send a link to access the source code of the static analysis software used in this experiment.
Participation in this study is completely voluntary, with no compensation for the participants.

Researchers in charge of this study:
Willian Oizumi (PUC-Rio)
Leonardo Sousa (CMU)
Anderson Oliveira (PUC-Rio)
Alessandro Garcia (PUC-Rio)
Diego Cedrim (PUC-Rio)
Carlos Lucena (PUC-Rio)

Contact: oizumi.willian@gmail.com

*Obrigatório

1. Do you agree to voluntarily participate in this study? *

Marque todas que se aplicam.

☐ I agree to voluntarily participate in this study

2. E-mail *

3. Where do you live (City/Country)? *

4. What is your age group? *

   Marcar apenas uma oval.

   - ◯ Between 18 and 24 years old
   - ◯ Between 25 and 34 years old
   - ◯ Between 35 and 44 years old
   - ◯ Between 45 and 54 years old
   - ◯ Between 55 and 64 years old
   - ◯ Between 65 and 74 years old
   - ◯ Over 74 years old
   - ◯ I prefer not to inform

5. Where do you identify yourself in the gender spectrum? *

   Examples: Binary (Female or Male), Non-Binary (Transfeminine, Trans-male, Polygender, etc)

6. What is your current position? *

   Examples: Software Developer, Software Architect

7. What is your experience with software development? *

Marcar apenas uma oval.

- ◯ Less than 1 year
- ◯ Between 1 and 5 years
- ◯ Between 6 and 10 years
- ◯ Between 11 and 15 years
- ◯ Between 16 and 20 years
- ◯ More than 20 years
- ◯ I prefer not to inform

8. How do you rate your knowledge of the Java programming language? *

Marcar apenas uma oval por linha.

	I prefer not to inform	None	Beginner	Intermediary	Experienced	Specialist
Level	◯	◯	◯	◯	◯	◯

9. How do you rate your knowledge of architectural decisions in software systems? *

Marcar apenas uma oval por linha.

	I prefer not to inform	None	Beginner	Intermediary	Experienced	Specialist
Level	◯	◯	◯	◯	◯	◯

10. How do you rate your knowledge about refactoring? *

Marcar apenas uma oval por linha.

	I prefer not to inform	None	Beginner	Intermediary	Experienced	Specialist
Level	◯	◯	◯	◯	◯	◯

11. How do you rate your knowledge of code smells? *

Marcar apenas uma oval por linha.

	I prefer not to inform	None	Beginner	Intermediary	Experienced	Specialist
Level	◯	◯	◯	◯	◯	◯

12. What type of project will you perform the study on? *

Marcar apenas uma oval.

- ◯ Open Source Software
- ◯ Closed Source Software

13. Please provide us with information, which is not confidential, about the project to be analyzed in this study. *

We are interested in information, such as: architectural style, domain, frameworks and size of the development team. If the project is open source, please provide us with the link to the repository.

14. How do you rate your familiarity with the project chosen to analyze? *

Marcar apenas uma oval por linha.

	I prefer not to inform	None	Small	Medium	High	Very High
Level	◯	◯	◯	◯	◯	◯

15. How long have you been working with the project chosen to analyze? Set to zero if you have never made changes to the source code of the chosen project. *

A.6.2
Form Used During the Quasi-Experiment

Following, we present the form filled by the developers during the task of identifying design problems.

Case Analysis

*Obrigatório

1. E-mail *

2. Start Time *

   Please indicate the time when you started to analyze this case. Take the time you think is necessary to analyze each case.

   Exemplo: 08h30

3. In your opinion, is there a problem of structural degradation in this case? *

   Marcar apenas uma oval.

   - Yes *Pular para a pergunta 4*
   - No *Pular para a pergunta 11*

Structural Degradation Identified

4. Description and justification of the degradation problem found *

5. What is the granularity of the identified degradation problem? *

Marcar apenas uma oval.

- ◯ Implementation Level
- ◯ Architecture and Design Level

6. How do you rate the severity of the identified degradation problem? *

The closer to 5, the greater the perceived severity.

Marcar apenas uma oval por linha.

	1	2	3	4	5
Severity	◯	◯	◯	◯	◯

7. How useful was the information provided by the static analysis tool to reach this conclusion? *

The higher the rating, the greater the perceived usefulness.

Marcar apenas uma oval por linha.

	1	2	3	4	5
Usefulness	◯	◯	◯	◯	◯

8. Describe what specific information was helpful in identifying the degradation problem. Such information may or may not have been provided by the tool. *

9. Do you believe that the refactorings suggested by the tool would be sufficient to remove the identified degradation problem? (for each smell check the "Refactorings Suggestion" tab) *

Marcar apenas uma oval.

- ◯ Yes
- ◯ No
- ◯ Partially

10. Additional comments

Additional comments on this case or on your responses.

Pular para a pergunta 14

No Structural Degradation

11. How useful was the information provided by the static analysis tool to reach this conclusion? *

The higher the rating, the greater the perceived usefulness.

Marcar apenas uma oval por linha.

	1	2	3	4	5
Usefulness	◯	◯	◯	◯	◯

12. Describe what information, provided or not by the tool, was most useful to rule out the existence of a degradation problem. *

13. Additional comments

Additional comments on this case or on your responses.

Case Analysis Completed

14. End Time *

Please indicate the time when you finished analyzing this case.

Exemplo: 08h30

15. Case Description *

In the static analysis software, click the "Copy Data and Open Next Case" button. The case description will be copied to the clipboard. Paste here the case description provided by the experiment tool. If, for any reason, you lose data from the clipboard, the description can also be copied from a file ("case_ [case number]") that will be saved in the same folder as the static analysis software.

A.6.3 Form Used as a Post-study Interview

Following, we present the form filled by the developers after performing the task of identifying design problems.

Post-Study Interview

*Obrigatório

1. E-mail *

2. Did any external factors hinder the execution of the study's tasks? For example, you cannot concentrate because there was an interruption during the study execution. *

3. Regarding the duration of the study, do you think it was appropriate, too long or would you need more time? What led you to that conclusion? *

4. What features in the tool would you like to include in your IDE or in the plug-ins you already use? *

5. What features does the tool need to become a plug-in that you would use in your IDE? *

6. Would you like to receive reports with information about the results of this study? *

   We plan to have a first report in January 2021.

   Marcar apenas uma oval.

   - ◯ Yes
   - ◯ No

7. General comments and feedback on your participation in this study.

B
Study on the Use of Robustness Smells Combined With Maintainability Smells

In this appendix, we present details of our second study (see Chapter 4). The following subsections describe the *(i)* code smells collected, and *(ii)* detailed statistics of maintainability smells density for each system.

B.1
Code Smells Collected

Table B.1 presents the maintainability smells considered in this study and their respective descriptions.

Table B.1: Maintainability Smells and its Descriptions

Maintainability Smell	Description
Brain Method	Long and complex method that centralizes the intelligence of a class
Dispersed Coupling	A method that accesses many elements, and the accessed code elements are dispersed among many classes
Feature Envy	A method that is more interested in a class other than the one it actually is in
Intensive Coupling	A method that has tight coupling with other methods, and these coupled methods are defined in the context of few classes
Long Method	A method that is unduly long in terms of lines of code
Message Chain	A long chain of method invocations is performed to implement a class functionality
Long Parameter List	A method having a long list of parameters, some of which are avoidable
Shotgun Surgery	When a single change performed on the method demands multiple other changes on other classes

Table B.2: Robustness Smells and its Descriptions

Robustness Smell	PMD Name	Description
Catch Generic Exception	AvoidCatchingGenericException	A method that catches generic exceptions such as NullPointerException, RuntimeException, and Exception in a try-catch block
Method Throws Exception	SignatureDeclareThrowsException	A method that explicitly throws java.lang.Exception
Empty Catch Block	EmptyCatchBlock	An exception is caught, but nothing is done
Catch NPE	AvoidCatchingNPE	Code throwing NullPointerExceptions
Rethrows Exception	AvoidRethrowingException	Catch blocks that rethrow a caught exception
Throw New Instance of Same Exception	AvoidThrowingNew InstanceOfSameException	Catch blocks that rethrow a caught exception wrapped inside a new instance of the same type
Throw Exception in Finally	DoNotThrowExceptionInFinally	Method throwing exceptions within a 'finally'
Exception as Flow Control	ExceptionAsFlowControl	Catches the use of exception statements as a flow control device
Throw NPE	AvoidThrowingNullPointerException	A method that throws a NullPointerException

Table B.2 presents the robustness smells considered in this study. 2nd column presents the name as recognized by the PMD static analysis tool used to identify the smell. 3rd column presents the description of the robustness smell.

B.2 Density of Maintainability Smells in Classes with and without Robustness Changes

Following, we present, for each system, the detailed statistics for the analysis that we performed. Each table presents the density of maintainability smells when a class had robustness changes and when it did not have this change.

Table B.3: Density of Maintainability Smells for the *elasticsearch-hadoop* System

Statistics for elasticsearch-hadoop	**Density With Exception**	**Density Without Exception**
Minimum	0	0
Maximum	32	38
Mean	2.9	2.6
Median	1	0
Mode	0	0
Std. Dev.	5.2	4.5

Table B.4: Density of Maintainability Smells for the *apm-agent-java* System

Statistics for apm-agent-java	**Density With Exception**	**Density Without Exception**
Minimum	0	0
Maximum	73	48
Mean	3.2	2.3
Median	1	1
Mode	0	0
Std. Dev.	7.5	6.7

Table B.5: Density of Maintainability Smells for the *okhttp* System

Statistics for okhttp	**Density With Exception**	**Density Without Exception**
Minimum	0	0
Maximum	197	181
Mean	30.8	4.3
Median	8	0
Mode	0	0
Std. Dev.	37.8	22.3

Table B.6: Density of Maintainability Smells for the *dubbo* System

Statistics for dubbo	**Density With Exception**	**Density Without Exception**
Minimum	0	0
Maximum	38	17
Mean	4.7	0.8
Median	3	0
Mode	0	0
Std. Dev.	6.6	3.2

Table B.7: Density of Maintainability Smells for the *fresco* System

Statistics fresco	**Density With Exception**	**Density Without Exception**
Minimum	0	0
Maximum	27	30
Mean	3.29	1.57
Median	1.00	1.00
Mode	0.00	0.00
Std. Dev.	5.25	2.91

Table B.8: Density of Maintainability Smells for the *netty* System

Statistics netty	**Density With Exception**	**Density Without Exception**
Minimum	0	0
Maximum	113	157
Mean	8.72	3.74
Median	4.00	1.00
Mode	0.00	0.00
Std. Dev.	16.39	8.19

Table B.9: Density of Maintainability Smells for the *rxjava* System

Statistics rxjava	**Density With Exception**	**Density Without Exception**
Minimum	0	0
Maximum	86	244
Mean	3.53	5.58
Median	1.00	1.00
Mode	0.00	0.00
Std. Dev.	12.52	13.23

Table B.10: Density of Maintainability Smells for the *spring-security* System

Statistics spring-security	**Density With Exception**	**Density Without Exception**
Minimum	0	0
Maximum	27	71
Mean	2.27	2.34
Median	1.00	1.00
Mode	0.00	0.00
Std. Dev.	4.28	4.96

Table B.11: Density of Maintainability Smells for the *spring-boot* System

Statistics spring-boot	**Density With Exception**	**Density Without Exception**
Minimum	0	0
Maximum	45	56
Mean	3.39	2.63
Median	1.00	1.00
Mode	0.00	0.00
Std. Dev.	7.78	5.97

Table B.12: Density of Maintainability Smells for the *spring-framework* System

Statistics spring-framework	**Density With Exception**	**Density Without Exception**
Minimum	0	0
Maximum	23	132
Mean	3.41	3.89
Median	2.00	1.00
Mode	0.00	0.00
Std. Dev.	4.08	9.25

C
Study on How Developers Discuss and Address NFRs

In this appendix, we present details of our third study (see Chapter 5). The following subsections describe the *(i)* dataset, *(ii)* keywords identified in the dataset, *(iii)* developers' metrics,*(iv)* developers metrics related to NFRs, *(v)* NFR sub-categories, and *(vi)* survey applied.

C.1
Dataset

We provide a dataset with 1,583 PR discussions classified in terms of NFR presence. After manual classification, we built a dataset composed of PR discussions, each one classified in terms of *(i)* the presence of the NFRs type addressed, *(ii)* the location in the PR where the discussions are triggered, *(iii)* keywords mentioned in the discussion, and *(iv)* discussion content addressing the NFR. This classification allowed us to characterize the PR discussions and identify the developers discussing NFRs.

Our dataset is structured with the following columns:

- **NUMBER_PR**: The number from the PR analyzed.
- **SYSTEM**: The system analyzed.
- **URL**: The URL from the PR analyzed.
- **NFR_TYPE**: The NFR type identified in the PR discussion.
- **PHRASE**: The phrase that led to the NFR classification.
- **KEYWORDS**: The keywords related to the NFR present in the discussion.
- **LOCATION**: The location where the mention of the NFR was present (e.g., title, description).
- **OBS**: Observations regarding the discussion.

In our dataset, we have:

- 609 PR discussions related to **Maintainability**.
- 393 PR discussions related to **Security**.
- 336 PR discussions related to **Robustness**.
- 195 PR discussions related to **Performance**.

C.2 Keywords

Through our dataset, we identified a set of keywords related to the NFRs. We divided these keywords into two groups: (i) single keywords and (ii) composed sentences keywords. The second group has combinations of keywords that can reinforce the presence of the NFR due to its context.

For each NFR, we also ranked the keywords according to the number of times that the keywords appeared on our manual classification. For instance, the "cleanup" keyword appeared 15 times when we were classifying *maintainability* PRs.

The "*" represents that this keyword can have multiple variations. For instance, "doc*" can be related to "documentation", "document", "documenting", and so on.

Following, we present the full list of keywords identified.

Maintainability

Single Keywords

"doc*: 49,
"javadoc": 15,
"cleanup": 15,
"polish*": 12,
"typo": 8,
"simpli*": 7,
"read*": 6,
"improve*": 5,
"fix": 5,
"duplicat*": 5,
"refact*": 4
"update": 4,
"customi*": 3,
"nam*": 2,
"dependenc*": 2,
"deprecated": 2,
"build": 2,
"maintain*": 2,
"clean*": 2,
"replace*: 2,
"readme": 2,
"unecessary": 1,
"syntax": 1,
"misleading": 1,
"clarify": 1,
"comprehensible: 1,
"outdated": 1,
"renaming": 1,
"migrate": 1,
"maintainability": 1,
"guidelines": 1

Composed Sentences Keywords

"documentation + improvement": 1,
"fix + documentation": 1,
"improve + docs": 1,
"clarify + docs + for + usage": 1,
"improve + extension": 1,
"duplicating + logic": 2,
"not + required: 2,
"configuration + properties": 1,
"duplicated code": 1,
"remove + incorrect": 1,
"facilitate + creation": 1,

"long method": 1,
"less code: 1,
"unused + dependency": 1,
"easier + understand": 1,
"correction + doc": 1,
"naming + convention": 1,
"to + make + the + code + more": 1,
"difficult + to + read: 1,
"easy + integration": 1,
"single + responsibility: 1,
"outdated + javadoc: 1,
"whitespace format": 1,
"code + style: 1

Security

Single Keywords
"auth*": 39,
"oauth*: 17,
"security": 15,
"JWT": 11,
"token": 9,
"saml*": 8,
"password": 5,
"vulnera*": 4,
"exception": 4,
"session: 4,
"ssl": 3,
"exposed": 3,
"sha*": 3,
"jwks": 3,
"bearer": 3,
"attack*": 3
"crypto*": 3,
"sanitize": 2,
"denied": 2,
"leaking": 2,
"null": 2
"login": 2,
"CSRF": 2,
"protected": 1,
"cookie": 1,
"ldap: 1,
"breach": 1,
"anonymous: 1,
"decrypt": 1,

"access": 1,
"secure": 1,
"key": 1,
"secret": 1,
"threat": 1,
"victim": 1,
"thread": 1,
"safe": 1,
"decrypter": 1

Composed Sentences Keywords
"prefer + the + https + protocol": 1,
"memory + leak": 1
"public + key": 1,
"grant + access": 1
"token + Authentication": 1
"access + tokens": 1,
"exposed + as + a + bean": 1,
"timing + attacks": 1,
"token + hashing": 1

Robustness

Single Keywords
"exception*": 30,
"*throw*": 19,
"error*": 16,
"*Exception": 10,
"fail*: 8,
"null": 6,
"*handl*": 5,
"catch*": 4,
"NPE": 4,
"stacktrace": 1,
"logging": 1,
"caught": 1,
"robustness": 1,
"crash": 1,
"Ensure": 1,

Composed Sentences Keywords
"null + check*"": 2,
"error + handl*": 2,
"should + fail": 1,
"shuts + down": 1,

"error + code": 1,
"message + exception": 1,
"throw + exception": 1,
"chain + exception": 1,
"propagate + root + cause": 1,
"returning + validation + errors": 1,
"null + pointer + exception": 1,
"exception + handling": 1,
"handling + exception": 1
"does + not + handle": 1,
"avoid *Exception: 1

Performance

Single Keywords
"performance": 14,
"optimiz*": 11,
"*sych*": 4,
"benchmark": 3,
"memory": 2,
"lock": 2,
"parallel": 2,
lazy*: 2
"cpu": 2,
"inneficien*": 2,
"threadsafe": 1,
"sockets": 1,
"overhead": 1,
"stream: 1,
"loop": 1
"timeout": 1,
"fast": 1,
"hotspots": 1,
"expensive": 1,
"caching": 1,
"speed up": 1,
"buffer": 1,
"payload": 1,
"byte": 1,
"encoder": 1,
"costly": 1,
"cache": 1,
"pipeline": 1,
"session": 1,

"allocation": 1,
"memorysaving": 1
"slower": 1

Composed Sentences Keywords
"multiple + beans": 1,
"time + improvement": 1,
"early + initialization": 1,
"interface + cache": 1,
"prevent + excessive + object + reallocations": 1,
"lazily + inititalize": 1,
"multiple + invocations": 1,
"called + just + once": 1,
"memorysaving": 1
"unnecessary + fetching": 1,
"reduce + allocations": 1,
"lazy + exceptions": 1,
"storing + unnecessary": 1,
"cpu + cycles": 1

C.3 Developers' Metrics

Aiming to investigate the developers' characteristics, we computed metrics describing their repository activities and code quality. We computed the metrics by gathering raw data through the GitHub API and performing aggregations to compound more complex metrics. The complete list of metrics is presented as follows.

- Number of PRs merged
- Number of PRs opened
- Number of PRs closed
- Mean time between merged PRs
- Number of Commits
- Average size of commits (Size in terms of source file)
- Number of commits with .XML files
- Number of commits with .Java files
- Number of reviews
- Number of lines revised
- Number of modules revised
- Number of refactorings
- Number of comments
- Mean time between developer comments
- Mean discussion duration (when the developer participates)
- Mean number of words from the developer's messages on PR discussions
- Total number of words from the developer's messages on PR discussions
- Experience in days (From the first contribution to the last)

C.4 Developers' Non-Functional Requirements Metrics

We identified 63 developers based on their participation in particular tasks, allowing us to explore their individual characteristics.

participates_NFR: Number of times that the developer participates in a discussion related to the NFR (e.g., *participates_security: 10* means that the developer participated in 10 discussions related to security).

opened_discussion_NFR: Number of times the developer opened discussions related to the NFR.

commented_NFR: Number of times the developer commented on discussions related to the NFR.

reviews_NFR: Number of times the developer reviewed source code on discussions related to the NFR.

commited_NFR: Number of times the developer had committed to discussions related to the NFR.

None indicates the number of times the developer did not participate in any task related to the NFRs. All NFRs are the total sum for the tasks regarding the four NFRs

We computed the quartiles for each metric and indicated whether the developer was part of the high, low, or medium quartile. The never tag is marked as True when the developer never performs the task for the correspondent NFR. To exemplify, let us consider the following metrics:

```
"reviewed_robustness_high": true,
"reviewed_robustness_low": false,
"reviewed_robustness_medium": false,
"reviewed_robustness_never": false
```

In this case, the developer had a high rate of reviews related to Robustness compared to other developers in the same system.

C.5 Non-Functional Requirements Sub-Categories

For each NFR used in this study, we subdivided them according to the type of change that was mentioned in our dataset. That allowed us to understand how many topics the pull requests title and descriptions could address. Following we describe each sub-category.

Robustness

1. Error Representability: whether errors are properly represented/specified/thrown.
2. Error Propagation: whether the error is properly propagated by either remapping to another error type or directly propagating to upper levels.
3. Error Recoverability: whether proper exception handling and clean-up actions are executed after an exception is raised, and the program's normal behavior returns to a consistent/safe state after those actions are executed.

Performance

1. Memory Usage: whether the change increased the memory usage on the application.
2. Concurrency (Thread Sync): whether there was a change in the concurrency of the system by using threads.
3. Concurrency (Scheduling): whether there was a change in the concurrency of the system by using scheduling.
4. Response Time: whether the change increased (or decreased) the response time (e.g., page loading).
5. Complexity: whether the change increased (or decreased) the computational complexity.

Security

1. Customize Parameters: whether the change allows configuring security parameters.
2. Information Protection: whether the change avoids accessing encrypted addresses.

3. Support for New Feature: whether the change supports security features (e.g., authentication mechanisms).

4. Inconsistent Behavior: whether the change fixes an inconsistent behavior related to security.

Maintainability

1. Feature Enhancement: whether the change is related to improving the system maintainability.

2. Code Simplification: whether the changes aim to simplify the code, improving its readability.

3. Move Component: whether the change focuses on moving components that could be highly coupled, hampering the maintainability.

4. Documentation: whether the change improves the software documentation.

5. Readability: whether the change improves the code readability.

6. Extensibility: whether the change improves the software extensibility.

7. Encapsulation: whether the change improves the modules' encapsulation.

C.6 Survey

To answer *How do developers perceive and address NFRs in their daily work?*, we conducted an opinion survey with 44 developers working with multiple closed-source systems. The survey is composed of questions ranging from the early stages of the software system to its continuous maintenance. The survey questions are presented as follows:

Exploring Developers' Approach for Dealing with Non-Functional Requirements

Researchers: Anderson Oliveira, Alessandro Garcia, Caio Barbosa, Daniel Coutinho, João Correia, Juliana Alves Pereira, Paulo Vítor Libório, Rafael Maiani, Wesley KG Assunção

Objective: The primary goal of this survey is to gain an understanding of your experiences and strategies when dealing with Non-Functional Requirements (NFRs) and engaging in discussions related to this kind of requirement during software development. By NFR, we refer to aspects of a software system that go beyond its primary functionalities. These requirements define how the system should behave in terms of performance, security, robustness, and other qualities essential for overall effectiveness and to ensure the system's longevity. By sharing your insights, you will contribute to enhancing the awareness and practices surrounding NFRs, ultimately aiding the development of tools and best practices that benefit the software development community. Your participation will help shed light on the significance of NFR discussions and their impact on project outcomes.

Consent to Participation

1. I declare that I am over 18 years old and agree to participate in non-invasive studies conducted by researcher Anderson José Silva de Oliveira under the coordination of Prof Dr. Alessandro Garcia and Prof. Dr. Juliana Alves Pereira as part of the research carried out by the OPUS Research Group, belonging to PUC-Rio. This study aims to understand how developers deal with NFR discussions on their systems.

2. I understand that all responses are personal and based on my experience.

3. All information collected in this study is confidential. I understand that my name will not be identified at any time. There will be no attempt to identify who you are, and your details will be reported in a non-identifying form.

4. You may withdraw from participating in the research and withdraw your consent at any time.

For questions, doubts or complaints about the study, you can contact **Anderson Oliveira: anderson.jose.so@gmail.com**

* Indica uma pergunta obrigatória

1. Thanks in advance! Do you want to participate in this survey? *

   Marcar apenas uma oval.

   - ◯ Yes *Pular para a pergunta 2*
   - ◯ No

Characterization

2. Age *

   _____

3. Gender *

   Marcar apenas uma oval.

   - ◯ Male
   - ◯ Female
   - ◯ Other
   - ◯ Prefer not to say

4. Higher academic degree *

   Marcar apenas uma oval.

   - ◯ High School
   - ◯ Bachelor
   - ◯ MBA
   - ◯ MSc.
   - ◯ PhD/DSc.
   - ◯ Outro: _____

5. Years of experience in software development *

   _____

6. Number of software development projects participated *

_____

7. In your opinion, what is your experience level with software development?

Marcar apenas uma oval.

- ◯ Very Low
- ◯ Low
- ◯ High
- ◯ Very High

8. Current role in your company: *

Marcar apenas uma oval.

- ◯ Software Developer
- ◯ Software Engineer
- ◯ Software Architect
- ◯ Software Testing
- ◯ Project Management
- ◯ Requirement Analysis
- ◯ Interface Design
- ◯ Outro: _____

9. Programming languages that you are proficient in *

Marque todas que se aplicam.

- [ ] Java
- [ ] Python
- [ ] C
- [ ] C#
- [ ] C++
- [ ] JavaScript
- [ ] Outro: _____

NFR Practices

10. 2.1 Please describe occasions in which you were involved in discussions about Non-Functional Requirements (NFRs) in software projects *

_____

_____

_____

_____

_____

11. 2.2 What triggers you to actively collaborate with NFR-related discussions in your projects? (Select all that apply)

Marque todas que se aplicam.

- [ ] Relevance to your area of expertise
- [ ] Impact on software quality that you use
- [ ] Alignment with project goals
- [ ] Potential risks of NFRs are not addressed
- [ ] Learning opportunity
- [ ] Outro: _____

12. 2.3 Which NFR types do you typically discuss during software development *

Marque todas que se aplicam.

- [ ] Performance: Speed, responsiveness, and efficiency of the software.
- [ ] Scalability: Ability to handle increased workload or user traffic.
- [ ] Robustness: Ability of the software to recover in various conditions.
- [ ] Availability: System uptime and accessibility for users.
- [ ] Security: Measures to protect data and prevent unauthorized access.
- [ ] Usability: User-friendliness and ease of interaction.
- [ ] Maintainability: Ease of making updates and changes to the software.
- [ ] Outro: _____

13. 2.4 What is your level of contribution to the NFRs elicitation and specification?

Marcar apenas uma oval.

- ◯ Very Low
- ◯ Low
- ◯ High
- ◯ Very High

14. 2.5 When dealing with problems related to NFRs, how do you access the information necessary? (Select all that apply) *

Marque todas que se aplicam.

- [ ] Requirements documentation
- [ ] Chat platforms with the team (e.g., Teams, Slack, Discord)
- [ ] Code comments
- [ ] Discussions on version control platforms (e.g., issues and pull requests)
- [ ] Meetings
- [ ] With other (more experienced) colleagues
- [ ] E-mail
- [ ] Outro: _____