Pontifícia Universidade Católica
do Rio de Janeiro

Daniel José Barbosa Coutinho

Revealing Interacting Factors in Decay of Software Design

Dissertação de Mestrado

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

Advisor: Prof. Alessandro Fabricio Garcia

Rio de Janeiro
September 2021

Pontifícia Universidade Católica
do Rio de Janeiro

Daniel José Barbosa Coutinho

Revealing Interacting Factors in Decay of Software Design

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

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

Prof. Marcos Kalinowski
Departamento de Informática – PUC-Rio

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

Rio de Janeiro, September 22nd, 2021

Daniel José Barbosa Coutinho

A MSc student in Computer Science at Pontifical Catholic University of Rio de Janeiro (PUC-Rio, Brazil). I have a bachelor's degree in Computer Science from Rio de Janeiro State University (UERJ). During his undergraduate course, was a part of research projects focusing on topics such as sofware engineering, data visualization and data science. Currently, my research focuses on software engineering topics such as: software maintenence and evolution, refactoring, and software repository mining. As a result of my research, I co-authored paper that were accepted in relevant conferences, such as: International Working Conference on Source Code Analysis and Manipulation (SCAM), Brazilian Symposium on Software Engineering (SBES), International Conference on Program Comprehension (ICPC), International Conference in Mining Software Repositories (MSR), and International Conference on Software Maintenance and Evolution (ICSME).

Bibliographic data

Barbosa Coutinho, Daniel José

Revealing Interacting Factors in Decay of Software Design / Daniel José Barbosa Coutinho; advisor: Alessandro Fabricio Garcia. – 2021.

84 f: il. color. ; 30 cm

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

Inclui bibliografia

1. Informática – Teses. 2. Degradação do Design. 3. Qualidade de Software. 4. Métricas de Software. I. Fabricio Garcia, Alessandro. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

Acknowledgments

I would like to first thank my family for always believing in me and for the support they have given me throughout my life, as without them I would not be where I am today. I would also like to thank my advisor, Prof. Alessandro Garcia, for his guidance and faith in my work, and for providing me the means and opportunities to mature as a researcher. To my undergraduate advisor, Prof. Marcelo Schots, for introducing me to research at a time when I was lost, always believing in me and my work, and for helping me improve not only as a professional but also as a human being. I would also like to thank all my friends and research colleagues, who have always been helpful when I needed guidance and support. My sincere thanks to Prof. Marcos Kalinowski and Prof. Alberto Raposo, for participating in my dissertation proposal committee, and for the valuable feedback which shaped this dissertation. I would also like to thank them both again for agreeing to participate in this dissertation committee and providing their feedback. I am also grateful to Capes, CNPq, FAPERJ, and PUC-Rio for the financial support that made this research possible. Moreover, my sincere thanks to the administrative staff of DI at PUC-Rio. This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001.

Abstract

Barbosa Coutinho, Daniel José; Fabricio Garcia, Alessandro (Advisor). **Revealing Interacting Factors in Decay of Software Design**. Rio de Janeiro, 2021. 84p. Dissertação de Mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Developers constantly perform code changes throughout the lifetime of a project. These changes may induce the introduction of design decay over time. Design decay may be reduced or accelerated by interacting factors that underlie each change. These factors may come from specific actions of change or repair – e.g., refactorings – to how developers contribute and discuss the changes. However, existing studies do not explain how these factors interact and influence design decay. They solely tend to focus on a few types of factors, and often consider them in isolation. Interactions between factors may cause different outcomes than those previously studied. Studying factors in isolation may not properly explain what are the most relevant set of interacting factors that influence design decay. This may indicate that existing approaches to avoid or mitigate design decay are misleading since they do not consider potentially relevant interactions between various factors. Thus, this dissertation reports an investigation that aims to increase the understanding of how a wide range of interacting factors can influence design decay in order to facilitate the investigation of which practices can be used to avoid or mitigate design decay. To this end, we performed an in-depth analysis to fill knowledge gaps on two types of factors: process-related (i.e., related to changes and their produced outcomes) and developer-related (i.e., related to the developer working on the changes) factors. We focused on analyzing the effects of potential interactions between the aforementioned factors and 12 sub-factors with regards to how they affected modules with different levels of decay. We observed diverging decay patterns in these modules. Our results indicate that both types of factors can be used to distinguish between different decay levels in classes. We have also observed that: (1) individually, the developer-related subfactor that represented first-time contributors, as well as the process-related one that represented size of the changes, did not exert negative effects on the changed classes. However, when analyzing specific factor interactions, we saw that changes where both of these factors interacted tended to have a negative effect and led to decay. Interestingly, this behaviour did not

alter even when the change was introduced via pull request (which usually triggers a code review process); (2) surprisingly, extraction-type refactorings often do not have a positive effect on code quality, while, by contrast, move refactorings were mostly positive. We also discuss how these findings in this dissertation can aid developers and researchers in improving their guidelines for the avoidance and monitoring of design decay.

Keywords

Design Decay; Software Quality; Software Metrics.

Resumo

Barbosa Coutinho, Daniel José; Fabricio Garcia, Alessandro. **Relevando Fatores Interativos na Degradação do Design de Software**. Rio de Janeiro, 2021. 84p. Dissertação de Mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Desenvolvedores realizam mudanças de código constantemente durante a vida de um projeto de software. Essas mudanças podem induzir a degradação progressiva do design. A degradação do design pode ser reduzida ou acelerada por fatores que interagem em cada mudança. Esses fatores podem variar desde uma mudança ou ação de reparo específica – e.g., refatorações – até a maneira como os desenvolvedores contribuem e discutem mudanças. Entretanto, estudos anteriores não exploram como esses fatores interagem e influenciam na degradação do design. Eles apenas focam em alguns fatores e tendem a os investigar em isolamento. Estudar os fatores em isolamento pode não explicar adequadamente qual é o conjunto mais relevante de interações entre fatores e qual sua influência na degradação do design. Isso pode indicar que abordagens existentes para evitar ou mitigar a degradação do design são incompletas, já que elas não consideram interações entre fatores que podem ser relevantes. Portanto, essa dissertação relata uma investigação que almeja aumentar a compreensão sobre como uma ampla gama de interações entre fatores pode afetar a degradação do design, para que consequentemente possam ser investigadas práticas efetivas para evitar ou mitigar esse fenômeno. Para tal fim, nós realizamos uma análise aprofundada buscando preencher lacunas no conhecimento existente sobre dois tipos de fatores: fatores relacionados ao processo (*i.e.* relacionados às mudanças e seus resultados produzidos) e fatores relacionados ao desenvolvedor (*i.e.* relacionados ao desenvolvedor trabalhando nas mudanças). Nós focamos em analisar os efeitos de possíveis interações entre os fatores previamente mencionados e uma série de sub-fatores, no que diz respeito como essas interações afetam módulos que sofreram diferentes níveis de degradação. Por exemplo, nós observamos que: (1) individualmente, tanto o sub-fator relacionado ao desenvolvedor que representa um desenvolvedor novato (que está contribuindo pela primeira vez), quanto o sub-fator relacionado ao processo que representa tamanho de uma mudança, não se mostraram relacionados a efeitos negativos na qualidade de código das classes alteradas. Porém, analisando interações entre fatores, nós observamos que mudanças em

que esses dois fatores interagem tendem a ter um efeito negativo no código, causando degradação. Interessantemente, esse comportamento não se alterou mesmo quando mudança foi introduzida através de uma pull request (o que frequentemente inicia um processo de revisão de código), (2) surpreendentemente, refatorações de extração frequentemente não tem um efeito positivo na qualidade do código, enquanto, em contrapartida, as refatorações de movimentação foram predominantemente positivas. Nós também discutimos como esses achados apresentados na dissertação podem ajudar desenvolvedores e pesquisadores na melhoria de suas diretrizes sobre como evitar e monitorar a degradação do design.

Palavras-chave

Degradação do Design; Qualidade de Software; Métricas de Software.

Table of contents

List of figures

List of tables

List of Abreviations

IQA – Internal Quality Attributes
RQ – Research Question
GQM – Goal Question Metric
CBO – Coupling Between Objects
LCOM – Lack of Cohesion in Methods
TCC – Tight Class Cohesion
CC – Cyclomatic Complexity
MAxNEST – Maximum Nesting
WMC – Weighted Methods per Class
NM – Number of Methods
NOC – Number of Children
DIT – Depth of Inheritance Tree
LOC – Lines of Code

1
Introduction

Changes are constantly being made to software systems that are in active development. In modern software development, developers usually submit their changes to version control repositories. Often, those repositories are hosted on social coding platforms, which are either open or private to an organization. This strategy allows developers to review and discuss the contents of each change made to the software.

Over time, especially without preventive measures, these changes increase the complexity of the system and reduce its overall quality [2]. One key facet of degradation of software quality is called design decay, in which design decisions made by developers in the aforementioned changes introduce problems [3], e.g., negative impacts on program maintainability and extensibility [4, 5]. Design decay can affect distinct modules differently, given that its evolution is asymmetrical and different modules can be introduced to the system with a variable set of preexisting design problems.

Developers and researchers tend to rely on different types of software symptoms to identify design decay [5]. Previous studies frequently use code smells for this purpose [6, 7]. Code smells are code structures, occurring in isolation or together, which may suggest the presence of design problems [8]. In this work, we will be utilizing another key type of symptom: those symptoms affecting the internal quality attributes of a software. We will directly utilize alterations to four IQAs to identify design decay: *cohesion*, *coupling*, *complexity* and *inheritance*. Another IQA, *size*, while often not used to identify design decay, may also be relevant as a complementary information.

Design decay can be affected by a myriad of factors, which may act singularly or simultaneously to affect (either to avoid, slow down or accelerate) the decay process [6, 9, 7, 10]. Due to the large number of ways in which those factors may be encountered, and the possibility of the presence of confounding factors, it can be challenging to pinpoint whether and to which extent each factor influences decay.

In this work, we are going to consider two of these types of factors: the *process-related factors* and the *developer-related factors*. These factors were chosen because there are known ways to observe and measure them during the evolution history of a project, allowing the development of a preventive approach to avoid design decay. One example where such an approach (based on factors' monitoring) could be developed would be in a code

review environment. Given the ever-growing popularity of collaborative version control platforms such as GitHub, the usage of the pull-request development model, where developers review and discuss code changes, is increasingly more common. In such an environment, it could be valuable to developers if a tool could be able to monitor certain interactions between influential factors, in order to assist developers in prioritizing which changes should be reviewed more carefully. These two aforementioned factors are explained as follows.

The *process-related factors* are factors directly related to activities governing the program changes and their produced outcomes. Previous studies have shown that some traditional metrics included in this category are good indicators of changes that impact design decay [6]. This group of metrics will be combined in this work as measures of the factor called here as "*change outcomes*", given that they directly represent the outcome of changes made by developers [11, 12, 13].

Another important aspect to consider is that repair actions may be performed during development, attempting to reduce the effects of design problems (or remove them altogether) present in the software system [14, 15, 16]. This strategy discourages an approach that considers factors in isolation, given the influence that these repair actions have on design decay might not be considered. In this work, we chose refactorings (e.g., actions aiming to improve the design of existing code through structural changes [8], further discussed in Section 2.2.1.2) to represent these repair actions. Thus, they will be considered a process-related factor.

The characteristics of each change may also be affected by the developer working on it. Thus, a second type of factor, the *developer-related factors*, will also be considered. This group can contain many types of different factors, e.g., the *discussion activity factors* and the *contribution factors*. The former relates to how developers communicate during the development of a change [7, 17], while the latter focuses on the developer's ownership of the source code affected by a change made by him/her [18, 19].

Given that scenario where multiple factors can simultaneously affect design quality, it becomes essential to study the process of design decay considering various factors altogether. Otherwise, we have only a very partial knowledge of how design decay occurs. Unfortunately, previous works have not studied design decay with this multi-dimensional perspective.

The current section motivates and contextualizes this work. The remaining sections of this chapter are organized as follows: Section 1.1 addresses the problem statement; Section 1.2 introduces the research questions addressed in this work; Section 1.3 discusses the limitations of related works; Section 1.4

provides a brief overview of the methodology of this work and presents its main contributions; and 1.5 provides an outline for how the remaining chapters of this dissertation are structured.

1.1 Problem Statement

There is a large number of non-trivial ways that different types of factors may influence software development, and more specifically, design decay. Previous research and, consequently, existing support resources tend to only consider factors individually. This simplistic explanation may indicate that those existing resources [1] are often not adequate as they do not consider potentially relevant influences that can arise from interactions between factors. For instance, in our results, when investigating changes that were performed by first-time contributors and large changes individually, we could not associate these factors with negative effects in terms of decay. However, when we investigated changes where those two factors interacted, we observed that those changes tended to have a negative effect on the code, leading to design decay. The fact that previous studies do not consider those interactions makes it so that information about how they affect design decay are scarce.

These problems create a situation where the elaboration of helpful resources to avoid or mitigate design decay is challenging, consequently making the job of practitioners trying to maintain design quality harder. This difficulty in creating new resources, inadequacy of existing resources, and lack of awareness on how developers perceive this process contribute to an ineffective combat and monitoring of decay. In summary, our general problem is described as follows.

> **General Problem.** There is a lack of resources to support software developers on avoiding or mitigating design decay.

We derived 2 specific problems from our general problem, which will be tackled in the research presented in this Master's Dissertation. First, while some of the above-named factors have been previously studied, those studies tend to focus on each factor individually (further discussed on Section 1.3). In software development, multiple factors can be influential at a given point in time. This might lead to different results than when those factors are studies individually, which are not addressed by those previous studies. In this work,

[1] In the scope of this work, we consider resources as information that be used to help developers, tool builds, and researchers to be more aware of the influences that different interactions between factors can have on design quality. Other types of resources, such as automated techniques and tools, are discussed in Section 4.2.

we call these cases where two (or more) factors co-occur in a code change that was performed to a module of an *interaction between factors* or a *co-occurrence of factors*. Second, some factors such as those regarding social aspects have only recently started being studied in this context. They were not previously linked to design decay, and when they were, this link was predominantly studied only in the context of code review.

Both these observations reinforce that there is a lack of understanding about how those factors actually affect design decay. In summary, our first problem is defined as follows.

Problem 1: There is a lack of understanding of how the co-occurrence of different factors can affect design decay.

Another important aspect often overlooked is that design decay is not a static process. Its progression may change significantly over the course of the software development. It is also an asymmetrical process, affecting distinct modules differently, since not all changes affect all classes and neither do they degrade design quality equally. This asymmetry com also be affected by variables such as the fact that when modules are first added to a system, they can already have varying amounts of preexisting design problems. Another factor that can cause this asymmetry are the repair actions, especially since developers usually need multiple changes to fully remove a design problem [20].

Given that scenario, we do not currently have a concrete understanding of how design decay structurally evolves in a software system. This also, consequently, limits our understanding on the interacting factors that affect the design structure's decay process, and whether different strategies to avoid or mitigate decay are necessary depending on how decayed a module already currently is. Those different strategies may be relevant as, at any given time, a developer may be introducing a module to the system that has varying amounts of preexisting design problems or performing changes to a module that may have already suffered different degrees of decay. In summary, our second problem is defined as follows.

Problem 2: There is a lack of understanding about how the evolution of the IQAs on modules differs depending on their levels of decay.

1.2 Research Questions

Given the scenario presented in the previous section, we can introduce our research questions, as follows. Each of the following questions is aligned with the problems stated in Section 1.1.

> **RQ**$_1$: Which factors, be they process-related or developer-related, may be indicators of changes that affect design decay?

This research question aims at investigating if it is possible to use the factors that were analyzed in this work, individually, to identify design decay. Only a minor part of our analysis focus on issues addressed in previous studies. Out first goal is to make sure our list of analyzed factors have (or not) a relationship with evolving design decay. Thus, we can investigate which associations of such factors can be relevant to understand design decay. This is addressed by our second research question as follows.

> **RQ**$_2$: What associations involving co-occurrences of certain factors may be inferred from code changes?

These associations allow us to better understand the design decay process, and identify if there are factors that stand out as being closely related to the introduction or avoidance of design decay. It also allow us to differentiate the behaviour of certain factors when comparing modules that did not decay much with classes that were considerably decayed.

> **RQ**$_3$: Do the associations between the co-occurrence of factors behave differently depending on the level of decay of a class?

As aforementioned, design decay is an asymmetrical process. Therefore, throughout the evolution of a system, those associations may interact with modules that suffered different levels of decay, which may affect the decay process. By investigating these differences, we may be able to infer whether and how different strategies can be employed when performing changes to modules with varying levels of decay. In turn, these strategies can be used to more effectively avoid or mitigate design decay.

1.3 Limitations of Related Work

There is a plethora of previous studies that investigate whether specific factors can affect design decay [18, 21, 22, 9, 23, 24, 25]. Those studies often focus on investigating those factors, be they process-related [23, 6, 24] or developer-related [7, 18, 25], in isolation. Therefore, they do not tackle the fact that modules can be affected, simultaneously, by different combinations of factors throughout their lifetime, which can affect the progression of how these modules are affected by design decay. As previously mentioned in Section 1.1, while we were investigating changes that were performed by first-time contributors and large changes individually, we could not associate them with negative effects of design quality. However, when we investigated interactions between those factors, we observed that those changes often had a negative effect on design quality.

Some factors, especially more socially-oriented ones, such as the developer-related factors (e.g., discussion activity, previous contributions), also only started being investigated in the context of design decay recently [7]. Previously, studies were limited to the context of code review, e.g. Uchôa et al. [6]. The presence of repair actions, such as refactorings, is also often not investigated as something that could interfere in how other factors affect design decay. Even though repair actions aim at improving code structure, they may often exert a negative effect.

Some factors (and/or their interactions) may influence design decay at different stages along software maintenance and evolution. The intensity of this influence along the the software lifetime is asymmetric between modules, and therefore, modules should be differentiated by the degree in which they have decayed, be it temporally or in intensity. Most studies do not make this distinction. For example, studies that only use code smells as indicators of design decay, instead of changes to the internal quality attributes, often use smells' density and smells' diversity to measure decay (e.g., Barbosa et al. [7], Uchôa et al. [6]). However, they do not consider when each smell was introduced or how much the software quality measures exceeded the thresholds used for the detection of a given code smell. As a consequence, this strategy can only reveal the influence of (interacting) factors only when a more advanced stage of decay (i.e., a smell is detected) manifests. The other disadvantage is that the selection of thresholds for smell detection is always subjective [26].

Due to the lack of evidence on how multiple relevant factors interact to influence design decay over time, researchers and developers cannot pinpoint which factors (or combinations of them) should be monitored in order to avoid

changes that increase design decay. On the opposite, they tend to focus in addressing one or another factor only. This restrict knowledge may misguide developer actions to combat design decay.

1.4 Main Research Contributions

Overview: In this work, we aimed to find associations (i.e., factor interactions) determining whether and how a subset of process-related and developer-related factors interact and affect design decay. To find these associations, we employed association rule mining, a data mining technique that aims to discover associations among a large set of items [27]. This technique does that by detecting patterns of values that frequently occur together in a given dataset [28]. In our case, the patterns to be discovered are the interactions between factors described earlier in this chapter.

To utilize the rule mining technique, we considered the presence of each of those aforementioned factors and the occurrence of changes to each of the internal quality attributes as events that can occur simultaneously in a code change. This technique allows us to identify not only whether factors interacted. It also reveals how this interaction occurred. To this end, we also considered, when necessary, different ways in which a factor could manifest, and also whether a change to an IQA was positive or negative. This will be further discussed in Step 8 of Section 3.3.2. This research was executed utilizing data that was mined from the version control repositories of seven different software systems, spanning approximately 45K commits. We also mined data about twelve sub-factors (Section 2.2), from which ten were found to be relevant to the decay process. Chapter 3 presents an extended version of a paper that was submitted to SANER'2022 at the time this dissertation was submitted. The paper provides a detailed description of the methodology and results derived from this work. A summary of our main findings can be observed by reading the statements surrounded by a rectangle in Section 3.4. A submission for this extended version is also planned, to be executed quite soon.

Main Contributions: In this work, we present two main contributions: (i) the design of a methodology that allows researchers to associate patterns of co-occurrence of factors affecting design decay; and (ii) a set of findings and insights obtained from the execution of this methodology. Our findings and insights inform researchers and developers on how design decay structurally evolves. By leveraging them, developers can progressively monitor the interactions between factors that accelerate decay. After a more in-depth investigation, those findings and insights can also lead to practical guidelines.

Those can guide developers and researchers understand how design decay is preserved, reduced or increased and how to avoid or mitigate the worrisome status of a design structure. For example, we recommended that developers should avoid or closely monitor extraction refactorings that are complex or executed alongside other changes. This is due to our observation that, unlike for other types of refactorings, negative effects were more frequent in extraction refactorings when large or complex changes were detected.

We also consider relevant three secondary contributions, described as follows: (i) the development of several scripts to extract historic data from version control repositories (specifically git); (ii) the detailed design and implementation of several metrics, quantifying various decay-related factors, which were produced by and used throughout the Master's research (described in Table 3.4); and (iii) a dataset containing not only the raw data that was used in our methodology, but also the processed data generated by each step of our studies.

Publications: As aforementioned, the contributions of this Master's Dissertation are reported in the publication presented in Chapter 3. Nevertheless, this research also contributed to the Master's research of other students (e.g., Soares et al.[29]) and to three Doctorate studies (e.g., Barbosa et al.[7], Bibiano et al. [30, 31], and Uchôa et al.[9]). Table 1.1 lists the related publications worked on during the execution of this research. Each of its columns present, respectively, the title, the venue it was submitted to, its status and whether it was directly derived from the author's dissertation or just involved his partial contributions. Regarding the first five papers presented, all of them are already published or were accepted in top international and national events. These are leading conferences in Software Engineering subareas and all of them have QUALIS scores varying from A1 to A3.

Table 1.1: Publications worked on during this research

Title	**Conference**	**Status**	**Relation to Master's Research**
How Does Incomplete Composite Refactoring Affect Internal Quality Attributes	ICPC 2020 (QUALIS A3)	Published	Related
On the Relation between Complexity, Explicitness, Effectiveness of Refactorings and Non-Functional Concerns	SBES 2020 (QUALIS A3)	Published	Related
Revealing the Social Aspects of Design Decay	SBES 2020 (QUALIS A3)	Published	Related
Predicting Design Impactful Changes in Modern Code Review: A Large-Scale Empirical Study	MSR 2021 (QUALIS A1)	Published	Related
Look Ahead! Revealing Complete Composite Refactorings and their Smelliness Effects.	ICSME 2021 (QUALIS A2)	Accepted	Related
On the Influential Interactive Factors on Degrees of Design Decay: A Multi-Project Study	SANER 2022 (QUALIS A2)	Submitted	Directly Derived
On the Influential Interactive Factors on Degrees of Design Decay: A Multi-Project Study (Extended Version)	IST Journal (QUALIS A1)	To Be Submitted	Directly Derived

1.5 Dissertation Outline

The remaining chapters of this dissertation are organized as follows. Chapter 2 presents background information and introduces: (1) design decay and how it can be observed via the internal quality attributes; (2) influential factors, their sub-factors, and how they relate to software development; and (3) association rule mining and the algorithm used in this work.

Chapter 3 presents a paper which presents a detailed description of the methodology and results derived from this work. A shortened version of this work was submitted to the *29th IEEE International Conference on Software Analysis, Evolution, and Reengineering* (SANER'2022). Finally, Chapter 4 summarizes the work, its contributions, and also describes the future work that is planned to further refine and strengthen this research.

2
Background

This chapter introduces three concepts widely used in this work. Section 2.1 introduces design decay, its symptoms, and how they are related to the internal quality attributes. Section 2.2 discusses process- and developer-related factors that can influence software development, and how they can be divided into sub-factors. Finally, Section 3.2.3 discusses association rule mining, a technique utilized in this work to detect co-occurrences of the aforementioned factors.

2.1
Design Decay and Internal Quality Attributes

Software design results from a series of decisions made during software development [32, 33]. However, throughout the development of a software project, a software's design may decay due to the progressive introduction of poor structures into the system, often called decay symptoms [8, 5]. Therefore, design decay is a phenomenon in which developers progressively introduce design problems in a system [3]. For example, a class being overloaded with multiple unrelated concerns – making its usage difficult and potentially causing ripple effects on other classes. Given the potential harmfulness of design decay, developers often need to perform repair actions such as refactoring to impacted source code locations [8]. These problems negatively impact quality attributes such as maintainability and extensibility [4, 5].

A previous study [5] has proposed a grounded theory that explains that developers tend to rely on five categories of symptoms to identify design decay. Examples of those categories are the code smells [8], principle violations [34], and internal quality attributes [8]. While previous studies frequently use code smells to identify instances of design decay [6, 7], in this work we selected metrics[1] that quantify proprieties of five different internal quality attributes (i.e., cohesion, coupling, complexity, inheritance, and size) to indicate symptoms of design decay. By using this approach, we can observe the decay directly, without facing the fragility of setting specific thresholds. This is the typical problem of strategies to detect smells. The simpler the model, the more widely it can be apllied.

[1]The metrics used in this work were chosen due to their presence on well-known catalogs and usage in previous works to quantify software quality [35, 21].

Additionally, AlOmar et al. [35] states that developers usually present intentions that can be mapped to internal quality attributes – some even being able to be mapped to specific metrics. Those five internal quality attributes and the metrics associated with them are described as follows.

Coupling represents the degree of interdependency between classes [36]. Thus, high *coupling* negatively affects the maintainability of a set of classes, by making even a small change in a method in one highly coupled class have a large, and possibly unpredictable, effect the change of many other interdependent classes. In this work, we utilize three different metrics to represent this internal quality attribute: Coupling Between Objects (CBO) [36], FANIN [37], and FANOUT [37]. We selected this set of metrics since they represent different dimensions of this attribute, i.e., CBO represents type-level undirected dependencies, while FANIN and FANOUT represent directed incoming and outgoing dependencies, i.e., inputs and outputs of call-level couplings. Table 2.1 provides a description for each of these metrics and how to interpret changes to their values, i.e., whether a change is value is considered an improvement when it increases or decreases.

Table 2.1: List of coupling metrics used in this work

Metric	**Description**	**Improves When**
Coupling Between Object (CBO) [36]	The number of classes coupled to the analyzed class.	Decreases
FANIN [37]	The number of external objects that invoke methods from the analyzed class.	Decreases
FANOUT [37]	The number of external method invocations made by the analyzed class to objects of other classes.	Decreases

Cohesion represents the degree to which the internal elements of a module are related to each other. A low *cohesion* within a class might lead to developers having difficulty on understanding the responsibilities of that class, thus potentially leading that class to become complex and bug-prone. In this work, we utilize three different metrics to represent this internal quality attribute: Lack of Cohesion in Methods 2 (LCOM2) [36], Lack of Cohesion in Methods 3 (LCOM3) [38], and Tight Class Cohesion (TCC) [39]. Again, we selected this set of metrics since they represent different dimensions of this attribute, i.e., LCOM2 and LCOM3[2] use different approaches to represent cohesion by considering methods that are unconnected in a class, while TCC considers connections between public methods. Table 2.2 provides a description for each of these metrics and how to interpret changes to their values.

[2]Two versions of the LCOM metric are considered as their different computation methods can result in different values.

Table 2.2: List of cohesion metrics used in this work

Metric	Description	Improves When
Lack of Cohesion in Methods 2 (LCOM2) [36]	Number of method pairs that do not share attributes, minus the number of method pairs that share attributes.	Decreases
Lack of Cohesion in Methods 3 (LCOM3) [38]	Treats each method pair as an individual entity, and determines the difference between the amount of similar and different pairs.	Decreases
Tight Class Cohesion (TCC) [39]	The number of directly connected visible methods in a class divided by the number of maximal possible connections between the visible methods of a class.	Increases

Complexity represents the level of overload of responsibilities in a single module or class. Such a complexity causes difficulties on the reading and understanding of classes by maintainers, who end up making mistakes and possibly introducing design problems or even bugs. In this work, we utilize five different metrics to represent this internal quality attribute: Cyclomatic Complexity (CC) [40], Essential Complexity (ev(G)) [40], Maximum Nesting (MAxNEST) [41], Weighted Methods per Class (WMC) [36], and Number of Methods (NM) [42]. Those metrics are used to represent different facets of complexity, i.e., NM considers the number of functionalities provided by a class as a measure of complexity, while the remaining metrics consider the structure of how these functionalities are implemented. Three of the remaining metrics also differ, i.e., CC considers the control flow of a method. In this work, we represent the CC of a class by the mean of the CC of its methods. Ev(G) connects complexity to whether that control flow can be simplified, while MAxNEST connects it to block nesting. WMC represents an alternative way of considering CC, as it represents a class by the sum of its methods. Table 2.3 provides a description for each of these metrics and how to interpret changes to their values.

Table 2.3: List of complexity metrics used in this work

Metric	Description	Improves When
Cyclomatic Complexity (CC) [40]	Measure of the complexity of a module's decision structure.	Decreases
Essential Complexity (ev(G)) [40]	Measure of the degree to which a module contains unstructured constructs.	Decreases
Maximum Nesting (MAxNEST) [41]	Maximum nesting level of control constructs.	Decreases
Weighted Methods per Class (WMC) [36]	The sum of the cyclomatic complexity of the methods of a class.	Decreases
Number of Methods (NM) [42]	The number of non-inherited methods declared in the analyzed class.	Decreases

Inheritance represents the complexity of a class's inheritance tree due to the number of superclasses and subclasses in the hierarchy. A class with a complex *inheritance* tree may cause additional difficulty during the maintenance process. For instance, it can cause developers to lose track of which class

in the inheritance tree has the implementation of which method, and which method is used by a specific class. In this work, we utilize three different metrics to represent this internal quality attribute: Base Classes (IFANIN) [43], Number of Children (NOC) [36], and Depth of Inheritance Tree (DIT) [36]. These metrics were selected as they represent different aspects of inheritance, i.e., IFANIN from whom a class has inherited, while conversely, NOC and DIT consider those who inherit from that class, directly (NOC) or indirectly (DIT). Table 2.2 provides a description for each of these metrics and how to interpret its value.

Table 2.4: List of inheritance metrics used in this work

Metric	**Description**	**Improves When**
Base Classes (IFANIN) [43]	The number of immediate base classes and interfaces.	Decreases
Number of Children (NOC) [36]	The number of immediate subclasses to the analyzed class.	Decreases
Depth of Inheritance Tree (DIT) [36]	The number of nodes between the root of the inheritance tree and the analyzed class.	Decreases

Finally, *size* represents the physical size of a code structure (e.g., a method or a class), often measured by the number of lines of code or by the number of statements. Even though many previous studies [44, 30, 45] associate the *size* attribute with code quality, due to its connection to bad code structures, such as long methods and long classes, the values of the metrics related to size vary wildly throughout a project's evolution – thus, care should be taken when considering size changes as being a positive or negative influence to the code's quality. Previous studies have also concluded that, in isolation, size metrics may not reflect what developers consider as quality in practice [35]. In this work, we utilize 24 different metrics to represent this internal quality attribute. This large number of metrics was chosen as an attempt to mitigate some of the aforementioned risks of using size metrics to quantify design quality, in order to increase the probability of observing meaningful findings. Table 2.5 provides a description for each of these metrics and how to interpret changes to their values. Given the aforementioned caveats when considering size as a measure of code quality, in this work, these interpretations for the values of the metrics representing this attribute were only used as complimentary information and were not used to directly measure code quality or to identify design decay.

Table 2.5: List of size metrics used in this work

Metric	**Description**	**Improves When**
Physical Lines (NL) [42]	The number of physical lines in the class.	Decreases
Blank Lines of Code (BLOC) [42]	The number of blank lines of code in the class.	Decreases
Source Lines of Code (LOC) [42]	The number of lines containing source code in the class.	Decreases
Declarative Lines of Code [42]	The number of lines containing declarative source code in the class.	Decreases
Executable Lines of Code [42]	The number of lines containing executable source code in the class.	Decreases
Lines with Comments (CLOC) [42]	The number of lines containing omments in a class.	Increases
Semicolons [42]	The number of semicolons in a class.	Decreases
Statements [42]	The number of statements in a class.	Decreases
Declarative Statements [42]	The number of declarative statements in a class.	Decreases
Executable Statements [42]	The number of executable statements in a class.	Decreases
Comment to Code Ratio [42]	The ratio of comment lines to code lines in a class.	Increases
Instance Variables (NIV) [42]	The number of instance variables in a class.	Decreases
Instance Methods (NIM) [42]	The number of instance methods in a class.	Decreases
Response For Class (RFC) [42]	The total number of methods, including inherited ones, in a class.	Decreases
Local Default Visibility Methods [42]	The number of local methods with default visibility in a class.	Decreases
Average Number of Lines [42]	The average number of physical lines between methods of a class.	Decreases
Average Number of Blank Lines [42]	The average number of blank lines of code between methods of a class.	Decreases
Average Number of Lines of Code [42]	The average number of lines of source code between methods of a class.	Decreases
Average Number of Lines with Comments [42]	The average number of lines containing comments between methods of a class.	Increases
Class Methods [42]	The number of class methods in a class.	Decreases
Class Variables [42]	The number of class variables in a class.	Decreases
Private Methods (NPM) [42]	The number of local private methods in a class.	Decreases
Protected Methods [42]	The number of local protected methods in a class.	Decreases
Public Methods (NPRM) [42]	The number of local public methods in a class.	Decreases

2.2 Influential Factors Along Software Development

Software development is constantly affected by a myriad of different factors. For convenience, these factors can be grouped into a number of key categories, which can be exemplified as follows.

- *Technical factors* relate to characteristics of the code or the software project itself, e.g., the internal quality attributes or non-functional characteristics such as performance.
- *Process-related factors* reflect the changes themselves and their outcomes, e.g. characteristics of the change such as size, churn, number of segments, number of files modified, as so forth. Other examples are actions performed in that change, such as refactorings.
- *Developer-related factors* comprise aspects pertaining to characteristics of the developers and their collaboration among developers, and discussions [46, 47, 48].
- *Collaboration network factors* emerge from the social networks formed by collaborations between developers [49].
- *Organization-related factors* from the practices of a community or organization, changing the developer's behaviour, e.g. community smells, which are organizational practices which might lead to problems in development [50].

As shown above, each of these larger groups of factors can include factors ranging from change actions, e.g., refactorings (i.e., actions aiming to improve the design of existing code), to more qualitative concepts such as how developers contribute and discuss the changes.

Previous studies [6, 9, 7, 10] consistently reported that different influential factors may contribute to avoiding, reducing, or accelerating design decay. Thus, understanding the influential factors on design decay is greatly important. In this work, we focus specifically on two of the aforementioned categories of factors: *Process-related factors* and *Developer-related factors*. These factors were chosen because there are known ways to observe and measure them during the evolution history of a project. Some of the previously mentioned factors were also not considered due to them requiring observations that are external to the version control repository (e.g., contribution factors and organization-related factors) or that can only be observed after the changes are integrated in the software system (e.g., technical factors). These aforementioned observations and measurements allow us to relate those factors with changes made by

the developers, which also enables the development of preventive approaches to avoid design decay. These two factors and their respective sub-factors are explained as follows.

2.2.1 Process-related Factors

In this work, we considered that *process-related factors* represent the universe of files, code segments, and repairing actions including refactorings (e.g., the number of move refactoring actions). We also considered that process-related factors are grouped into two factors: *change outcomes*, and *refactoring actions*.

2.2.1.1 Change Outcomes

Change outcomes represent the result of changes performed by developers during development [11, 12, 13]. We further divided this factor into three sub-factors that can be directly mapped to widely used process metrics that are known to represent different properties of changes [51] : *Change Set*, *Hunks Count*, and *Code Churn*. Previous studies that investigated those metrics have concluded that they are very effective at indicating changes that cause design decay [9].

Change Set relates to the number of classes that are modified in a code change alongside the file that is currently being observed. The rationale for using this sub-factor is that previous studies have shown that large changes are more likely to go through a process of code review, since developers perceive that large changes tend to introduce design decay [11].

Hunks Count refers to the number of distinct code segments (i.e., contiguous blocks of code) of a class that are modified in a code change. The rationale for looking at this sub-factor is that a high number of code segments modified is an indicator that a change might be complex [12].

Code Churn describes the sum of the number of lines added and deleted from a file in a code change. The rationale for investigating this sub-factor is that previous studies have observed that classes that are more change-prone are more likely to contain design problems [13].

2.2.1.2 Refactoring Actions

Refactoring actions represent code transformations made by developers to avoid or mitigate design decay [15, 16]. The presence of these actions in a

change might indicate that developers have identified a design problem on the target file that needs to be removed or minimized. For this factor, we chose to separate refactoring actions into four groups that represent different natures of that action: *Extraction*, *Move (or Movement)*, *Hierarchical*, and *Rename*. These groups of refactorings were chosen as they are able to represent the most frequent types of refactorings observed by previous studies [52, 30, 31, 53] (some of which analyze software projects in common with this work). These refactoring types are also the canonical types used to improve non-functional requirements associated with design decay: maintainability, comprehensibility, and reusability [8]. Less frequent refactoring types were not considered as they could result in findings that are not meaningful in the software projects analyzed (one case of this can be seen at the end of Section 3.4.3.3).

Extraction refactorings datache pieces of code from an expression, segment or construct into a new code element. The motivation for this type of refactoring can be due to a myriad of reasons, such as to improve readability, generalization, encapsulation, to avoid code duplication, among other reasons. This category includes the following refactoring types: *Extract Method*, *Extract Superclass*, *Extract Interface*, *Extract Attribute*, *Extract Class*, *Extract Subclass*, and *Extract Variable*.

Move refactorings move pieces of code between code elements that can contain it (e.g., a method can be moved between classes), without creating new code elements. Developers might execute this type of refactoring to improve encapsulation, cohesion, reduce coupling, and so forth. The following refactorings were included in this category: *Move Method*, *Move Class*, and *Move Attribute*.

Hierarchical refactorings are actions that move a code element or construct up or down in the class hierarchy. This type of refactoring often occurs when developers identify a generalization problem, where code constructs or elements on the same level of hierarchy perform similar work or have similar purposes, and can be generalized. We have included the following refactoring types in this category: *Pull Up Method*, *Pull Up Attribute*, *Push Down Method*, and *Push Down Attribute*.

Rename refactorings are actions that change the names given to code elements. This type of refactoring is mostly done to improve readability when developer has identified that the name of that element does not represent its current purpose. For this category, we have included the following refactoring types: *Rename Method*, and *Rename Class*.

2.2.2 Developer-related Factors

Developer-related factors were considered to represent the human-centered actions of code contributions, e.g., the number of words in a discussion. Developer-related factors were grouped into *discussion activity* and *contribution*. These groups and their sub-factors were selected as they represent a meaningful sample of social aspects that have been analyzed in previous studies [7, 9], and can also be measured without needing observations external to the version control repository or its hosting platform (in the case of this work, GitHub).

2.2.2.1 Discussion activity

Discussion activity represents the developers interaction during the exchange of messages and the contents of messages [7, 17]. We further divided this factor into three sub-factors: *Associated Pull Requests*, *Discussion Length* and *Comment Length*.

Associated Pull Requests refer to GitHub pull requests that refer to or introduce the changes to the file that is currently being analyzed. Measuring this allows us to track whether a change was introduced via a pull request or not (changes introduced via pull request often go through a code review process) and whether multiple pull requests refer to the same change, which might indicate that a change is complex or important, and therefore more discussed by developers.

Discussion Length and **Comment Length** represent two different facets (in terms of granularity) of the size of a discussion. The former refers to the entirety of a discussion, and is measured by summing the number of words inside all comments in a pull request. The latter refers to only pieces of it (i.e., each individual comment), and is measured as the mean of the number of words per comment. Previous studies have observed that changes introduced via pull requests with large discussions are more likely to include complex changes [17, 7], which consequently might be more likely to introduce design decay [11].

2.2.2.2 Contribution

Contribution focuses on the developers which have contributed to a file, based on who are the developers that have contributed to a certain software project, how many times each of these developers modified each file in the

system and their level of contribution to each class [18, 19]. We further divided this factor into two sub-factors, which were chosen since they can be observed at change time: **Contributors** and **Code Ownership**.

Contributors refers to the list of developers that have made changes to a certain class. Tracking this sub-factor can allow us to discern whether the author of a specific change is a newcomer (i.e., it is his first change) or not. Previous studies have observed that classes modified by new developers are more prone to decay [19, 18].

Code Ownership represent how responsible a developer is for the authorship of a specific file. It is measured as the percentage of lines of code in a file that is authored by a specific developer. We consider this sub-factor since previous studies have showed that files that were changed by minor contributors tend to be decayed than files only modified by major contributors [19, 18].

2.3 Association Rule Mining

In this work, we used Apriori, a widely used association rule algorithm [54], to investigate whether and how process- and developer-related factors interact to influence varying levels of design decay. Those interactions can be observed through the associations resulting from this technique, which is explained as follows.

Association rule mining is a data mining technique that aims to discover associations among a large set of data items [27]. This technique is used to detect patterns of values that occur together in a given dataset [28, 27]. This technique has been widely applied to support decision-making in various fields, such as business [55], intrusion detection [56], and even software engineering [57, 58, 59].

To illustrate the concept of association rule mining, consider a set of data items $I = \{i_1, i_2, ..., i_n\}$. Let D be a set of transactions (i.e., dataset), in which each transaction T is a subset of I, i.e., $T \subseteq I$. An association rule is an implication expressed as:

$$A \Rightarrow B \; where \; A \subset I, \; B \subset I, \; and \; A \cap B = \emptyset \tag{2-1}$$

In other words, when A occurs, B tends to occur (the opposite is not necessarily true). More specifically, A and B are disjointed sets of data items, in which A is called the *antecedent* and B is called the *consequent*. There

are three key measures commonly used to filter the relevant association rules: *support*, *confidence* and *lift* [28, 60].

Support determines how frequent the rule is applicable in the transaction set D. It is expressed as $Support(A \Rightarrow B)$, and represents the percentage of transactions that contain both A and B. It can be represented as follows.

$$Support(A \Rightarrow B) \ := \ Frequency(A \ AND \ B) \tag{2-2}$$

Confidence, on the other hand, measures the strength of the rules. It is expressed as $Confidence(A \Rightarrow B)$, and represents how frequent B appears in transactions that contain A. It can be represented as follows.

$$Support(A \Rightarrow B) \ := \ \frac{Support(A \Rightarrow B)}{Frequency(A)} \tag{2-3}$$

Finally, Lift is expressed as $Lift(A \Rightarrow B)$, and represents how strongly a rule influences a potentially random occurrence – if a rule's lift is equal to 1, it means that the consequent of the rule is independent from the antecedent, thus being a random result. Having a lift value higher than 1 means that the antecedent being fulfilled likely causes the consequent to appear, while lift values below 1 mean that the condition being fulfilled likely causes the inverse, i.e., the consequent to not appear. It can be represented in two different ways, as follows.

$$Lift(A \Rightarrow B) \ := \ \frac{Support(A \Rightarrow B)}{Frequency(A) \times Frequency(B)} \tag{2-4}$$

$$Lift(A \Rightarrow B) \ := \ \frac{P(A \cap B)}{P(A) \times P(B)} \tag{2-5}$$

The algorithm used in this work, *apriori*, is one of the widely used algorithms in association rule discovery [54]. The algorithm uses Apriori principle, which states that if an itemset is not frequent, all its subsets are not frequent as well. In this sense, it first scans the transactions and generates frequent item sets based on filtering criteria set by users. Then, a list of association rules is generated from frequent item sets. We used this algorithm to quantify the degree of association between different factors that can influence design decay. Since those associations happen inside each of the transactions, which in the case of this work are the commits, those associations can be seen as representing the interactions between the different factors analyzed.

2.4 Summary

This chapter introduced three key concepts to support the understanding of this dissertation. First, it introduced design decay and discussed how its symptoms can be identified using internal quality attributes. Second, described how different influential factors can affect software development, which of these factors were the focus of this work and why they were chosen, and also how we divided them into sub-factors. Third, it also introduced association rule mining and the apriori algorithm, which were used in this work to mine influential interactions between the aforementioned factors. The next chapter presents the study we conducted to address the problems listed in Section 1.1.

3
On the Influential Interactive Factors on Degrees of Design Decay: A Multi-Project Study

As we discussed in Chapter 1, developers regularly make design changes throughout the lifetime of a software project. These changes may introduce design problems, consequently causing decay. Design decay may be influenced interacting factors that underlie each change. These factors may vary from specific change actions – e.g., refactorings – to actions from developers, e.g., a comment in a code review process. Without being aware of interacting factors influencing design decay, developers cannot be warned of harmful changes to design.

However, while existing studies consider each factor in isolation, they lack of evidence about how these factors interact and influence design decay. In this chapter, we present the paper "On the Influential Interactive Factors on Degrees of Design Decay: A Multi-Project Study". A shortened version of this paper was submitted to the *29th IEEE International Conference on Software Analysis, Evolution, and Reengineering* (SANER 2022). As described in Section 1.5, this paper reports a study where we aimed to find associations between how a subset of process-related and developer-related factors (Section 2.2) interact and affect design decay (Section 2.1).

To find these associations, we employed association rule mining (Section 2.3), a data mining technique that aims to discover associations among a large set of items [27]. The associations are found by detecting patterns of values that frequently occur together in a given dataset [28]. To utilize this technique, we considered the presence of each of those aforementioned factors and the occurrence of changes to each of the internal quality attributes as items that can occur simultaneously in a code change. For example, an association rule mined using this technique could identify that, frequently, changes that modify a large number of files and contain move refactorings tend to improve cohesion and reduce coupling. This research was executed utilizing data that was mined from the version control repositories of seven different software systems, spanning approximately 45K commits.

In this study, we first observed that the factors that were considered can indeed be good indicators of changes that affect design decay, addressing the first research question of this Master's Dissertation, which is analogous to the first research question of the paper. After that, tackling the two remaining research questions of the dissertation, which also addresses the second research

question of the paper, we observed several patterns in how these factors interacted and affected design decay. We also compared how the design decay process differed between classes that had a few internal quality attributes affected by decay and classes where most of the internal quality attributes had suffered decay. As an additional viewpoint, we also compared these results with the patterns found when looking at all classes in the data set. In case the reader has read Chapters 1 and 2, you may consider skipping Section 3.1, which briefly describes the context and motivation of this work, and also provides a list of the main contributions and Section 3.2, which contains a shortened description of concepts such as design decay, developer- and process- related factors, and association rule mining. Some of the information in Section 3.3.2 is also redundant, specifically Tables 3.2, 3.3, and 3.4, which briefly introduce the metrics and sub-factors already described in detail in Chapter 2.

On the Influential Interactive Factors on Degrees of Design Decay: A Multi-Project Study

Daniel Coutinho*, Anderson Uchôa*, Caio Barbosa*, Vinícius Soares*,
Alessandro Garcia*, Marcelo Schots†, Juliana Pereira*, Wesley K. G. Assunção*
*Informatics Department – Pontifical Catholic University of Rio de Janeiro (PUC-Rio), Rio de Janeiro, Brazil
† Informatics and Computer Science Department – Rio de Janeiro State University (UERJ), Rio de Janeiro, Brazil

3.1 Introduction

Developers constantly change software systems, submitting their contributions to version control repositories, and allowing other developers to review [61] and discuss [62] their contents. Often, these changes may induce design decay over time [2, 3], whose symptoms manifest when the software modules become increasingly complex, large, coupled, and incohesive [63]. Decay levels can also vary among different modules of a system, since they evolve asymmetrically. Developers might also apply certain repair actions (e.g., refactorings) in an attempt to potentially revert design decay [14], which can further contribute to this asymmetry.

Design decay is often studied as a cut-and-dried issue [5]. Still, a myriad of factors can influence – either alone or simultaneously – how decay occurs, and to which extent it can be slowed down or accelerated. These influence factors can vary in nature and can also affect, to a greater or lesser extent, the effectiveness of repair actions. As such, the quality of a change might

be influenced by the developer(s) working on it and the change process itself. Process-related factors include actions and outcomes associated with the resulting software changes [11, 12, 13]. Conversely, developer-related factors include ways in which they discuss and contribute to a system [7].

Previous studies investigate whether these factors, in isolation, affect design decay [23]. Some focus on analyzing either developer- [7, 18] or process-related [23, 6] factors. They often disregard that modules are subjected to varying levels of decay and to evolving combinations of influential factors. They do not differentiate preliminary or advanced stages of design decay [64]. Thus, investigations on potential associations between those factors, as well as to which levels they can affect design decay, are currently missing in the literature. No study investigates if and how these complementary factors simultaneously and progressively influence decay. Thus, one cannot effectively explore which factors (or associations between them) should be monitored to avoid decay.

In this paper, we address these gaps through a multi-project study that reveals influential factors in varying levels of design decay. For this reason, we divided modules[1] in terms of how many of their internal quality attributes (IQAs) were affected by decay. We consider a module is slightly-decayed whenever only a few of its IQAs were affected; largely-decayed modules are those in which various IQAs were affected (further clarified in Section 3.3.2). To analyze how developer- and process-related factors can relate to these groups, we use association rule mining [65] to infer how factors may have differently influenced the decay of modules in these groups. We mined associations from nearly 45k commits made to seven systems. We analyzed 38 software metrics to characterize the design decay of modules with respect to five IQAs. To support our observations, we also analyzed 12 (sub-)factors, quantified by metrics, regarding process- and developer-related factors.

Our main contributions include an investigation about whether specific process- and developer-related factors differ between classes with varying levels of decay, and how certain factor associations can influence decay. We also provide a novel data set, allowing researchers to further investigate how these factors influence the IQAs over time. Our main findings are summarized as follows:

i) Process- and developer-related factors can be used to distinguish between slightly- and largely-decayed classes. Specifically, change actions (e.g., refactorings) and their outcomes consistently indicate alterations in decay level. Some developer-related factors – e.g., contribution volume and discussion density – also stood out.

[1]In this study, we considered each individual class as a module.

ii) Surprisingly, extraction-type refactorings often do not have a positive effect on IQAs. Even when interacting with factors that represent developers' experience and the class' level of decay, the result continues to be non-positive. By contrast, move-type refactorings have the opposite effect, consistently improving the IQAs.

iii) Specifically in slightly-decayed classes, when the factors that represented first-time contributors and significant changes in a class's code interacted, they tended to cause negative effects on the IQAs. Even when interacting with factors that show that the change was introduced via a pull request process (which could imply code review), they continued to maintain their negative effects.

3.2 Background

This section describes several concepts widely used throughout this work. Section 3.2.1 presents a overview of design decay, its symptoms and how they relate to the internal quality attributes. Section 3.2.2 discusses process- and developer-related factors that can influence software development. Finally, Section 3.2.3 discusses association rule mining.

3.2.1 Design Decay and Internal Quality Attributes

Design decay is a phenomenon in which developers progressively introduce design problems in a system [3]. It is caused by design decisions that negatively impact quality attributes such as maintainability and extensibility [4, 5]. An example is a class being overloaded with multiple unrelated responsibilities – making its usage difficult and potentially causing ripple effects on other classes. Given the potential harmfulness of design decay, developers often need to identify and refactor impacted source code locations [8].

As evidenced by an existing grounded theory [5], a key indicator used by developers to reduce design decay is the improvement of internal quality attributes (IQAs), such as *coupling*, *cohesion*, and *complexity* [66]. For instance, *coupling* represents the degree of interdependency between classes [36]. In this sense, a high *coupling* negatively affects the maintainability of a set of classes, by making even a small change in a method in one highly coupled class have a large, and possibly hard to predict, effect on many other classes.

In the case of *cohesion*, it represents the degree to which the internal elements of a module are related to each other. A low *cohesion* within a class might lead to developers having difficulty understanding the responsibilities of

that class, thus potentially leading that class to become hard to change and bug-prone. Finally, *complexity* represents the level of overload of responsibilities and decisions of a module or class. It causes difficulties in the reading and understanding of classes by developers, who end up making mistakes.

Other two IQAs are also adopted by the literature and considered by developers are: *size* and *inheritance* [36, 35, 5]. In the case of *inheritance*, it represents the complexity of a class's inheritance tree – due to the number of superclasses and subclasses in the hierarchy. A class with a complex *inheritance* tree may cause additional difficulty during the maintenance process. It can cause developers to lose track of which class in the inheritance tree has the implementation of which method, and which method is used by a specific class. As for the *size* attribute, it represents the physical size of a code structure (e.g. method, class), often measured by the number of lines of code. It is often considered as a measure of code quality, due to its connection to bad code structures, such as long methods and classes. However, the semantics of high and low values of the metrics related to this size vary widely throughout a software project's evolution and across different projects. Thus, care should be taken when considering size changes as being a positive or negative influence to the code's quality. Their meanings are hard to interpret in each context. Due to the importance of IQAs for detecting design decay, in this work, we selected a set of 38 metrics from the literature. They quantify a variety of properties of each IQA. These metrics are explained in more detail in Section 3.3.2.

3.2.2 Influential Factors along Software Development

Software development is constantly affected by a myriad of different factors. For convenience, these factors can be grouped into a number of key categories. In this work, we focus on two of these categories of factors, namely, *process-related factors*, that reflect the changes themselves and their outcome, and *developer-related factors*, which emerge from collaboration among developers [48]. These factors may vary from change actions, e.g., refactorings, to how developers contribute and discuss the changes, e.g., comments.

More specifically, *process-related factors* can be seen as representing the universe of files, code segments, code changes and repair actions (e.g. refactorings). On the other hand, *developer-related factors* represent the human-centered actions in code contributions, e.g., code review discussions. Previous studies [6, 9, 7, 10], consistently report that those groups of influential factors may contribute to avoiding, reducing, or accelerating design decay. Thus, understanding the influential factors on design decay is greatly important. We

specifically chose these factors since there are known ways to observe and measure them. The observation of such factors can be made before a change is introduced in a system, allowing developers to use a preventive approach to avoid design decay. Other types of factors, e.g. runtime information – do not enable this type of approach. In other words, there are many factors that can be only computed after the change has already made into the system.

Such factors are usually further grouped into more specific groups of factors that can be captured by the computation of metrics. In this work, process-related factors are grouped into two factors: *Change outcomes* and *Refactoring actions*. *Change outcomes* represent the properties of each change performed by developers during development [11, 12, 13]. *Refactoring actions* represent code transformations usually made by developers to avoid or mitigate design decay [15]. Developer-related factors are grouped into *Discussion activity* and *Contribution*. *Discussion activity* represents the developers interaction during the exchange of messages and the contents of messages [7, 17]. *Contribution* focuses on the developer's code ownership, based on the number of their previous commits to each file the software project and their level of contribution to a specific class [18, 19].

We also defined a set of 12 (sub-)factors, which can be found in Section 3.3.2. In this work, we rely on meaningful interactions between these sub-factors to understand how changes relate to decay. These interactions can happen between sub-factors grouped in the same factor. For example, a change where a new developer (i.e., it is his first change to the software project.) changes a large portion of a single class. This change would be identified as a interaction between *contribution* sub-factors, as that new developer would gain ownership of that code. However, they can also happen between sub-factors grouped in different factors. For example, that same aforementioned change could also, at the same time, interact with a *change outcomes* factor, since it could be a fairly large change. Another example in which this could happen would be a change that contained an extraction refactoring, was extensively discussed by developers, and was done by a new developer. This change would be identified as an interaction between sub-factors grouped in three different factors: *refactoring actions*, *discussion activity*, and *contribution*. By observing how there interactions are related to changes to the internal quality attributes, we are able to discern the influence these interactions exert over design decay. In order to identify these interactions and their influence, we utilize a technique called Associated Rule Mining, which is introduced in the following section.

3.2.3 Association Rule Mining

In this work, we used Apriori, a widely used association rule algorithm [54], to investigate whether and how process- and developer- related factors interact to influence varying levels of design decay. Those interactions can be observed through the associations resulting from this technique, which is explained as follows.

Association rule mining is a data mining technique that aims to discover associations among a large set of data items [27]. This technique is used to detect patterns of values that occur together in a given dataset [28, 27]. To illustrate the concept of association rule mining, consider a set of data items $I = \{i_1, i_2, ..., i_n\}$. Let D be a set of transactions (i.e., dataset), in which each transaction T is a subset of I, i.e., $T \subseteq I$. An association rule is an implication expressed by $A \Rightarrow B$ where $A \subset I$, $B \subset I$ and $A \cap B = \emptyset$. In other words, when A occurs, B tends to occur (the opposite is not necessarily true). More specifically, A and B are disjointed sets of data items, in which A is called the *antecedent* and B is called the *consequent*.

There are three key measures commonly used to filter the relevant association rules: *support*, *confidence* and *lift* [28]. Support determines how frequent the rule is applicable in the transaction set D. It is expressed as $Support(A \Rightarrow B)$, and represents the percentage of transactions that contain both A and B. Confidence, on the other hand, measures the strength of the rules. It is expressed as $Confidence(A \Rightarrow B)$, and represents how frequent B appears in transactions that contain A. Finally, lift is expressed as $Lift(A \Rightarrow B)$, and represents how strongly a rule influences a potentially random occurrence – if a rule's lift is equal to 1, it means that the consequent of the rule is independent from the antecedent, thus being a random result. Having a lift value higher than 1 means that the antecedent being fulfilled likely causes the consequent to appear, while lift values below 1 mean that the condition being fulfilled likely causes the inverse, i.e., the consequent to not appear.

We used this technique to quantify the degree of association between different factors that can influence design decay. Since those associations happen inside each of the transactions, which in the case of this work are the commits, those associations can be seen as representing the interactions between the different factors analyzed.

3.3 Study Settings

Section 3.3.1 presents both the goal and the associated research questions. Finally, Section 3.3.2 describes the study steps and procedures.

3.3.1 Goal and Research Questions

Our study can be described using the GQM approach [67], as follows: **Characterize** interactions between process- and developer-related factors **for the purpose of** identifying whether and how they simultaneously influence design decay, **with respect to** four IQAs, **from the viewpoint of** developers, tool builders, and researchers, **in the context of** software evolution.

To this end, we analyze the change history of seven software projects in terms of a number of different metrics (which are listed in Sections 3.3.2.2 and 3.3.2.4, to tackle the following research questions (RQs).

RQ_1: *How intensively, if at all, are process- and developer-related metrics good indicators of decay levels?* – **RQ_1** aims to infer if it is possible to distinguish between classes with varying levels of decay by looking at process- and developer-related metrics, and how strongly they indicate those differences. We analyze decay at the class-level as classes represent the main abstractions of project's object-oriented designs. Those results will be compared with previous studies that have investigated some of these factors.

RQ_2: *What associations between factors can be inferred from classes that suffered distinct levels of design decay?* – **RQ_2** aims to identify associations between either process- and developer-related factors, and changes to the IQAs. Those associations will provide us with information on how the decay process occurred. We also aim to identify if there are significant differences in the associations found for the analyzed levels of decay, and how often those associations can be found. Thus, by answering **RQ_2**, we can better understand how design decay happens by establishing how responses to different interactions between factors can affect decay.

3.3.2 Study Steps and Procedures

Figure 3.1 shows the study procedures, described as follows.

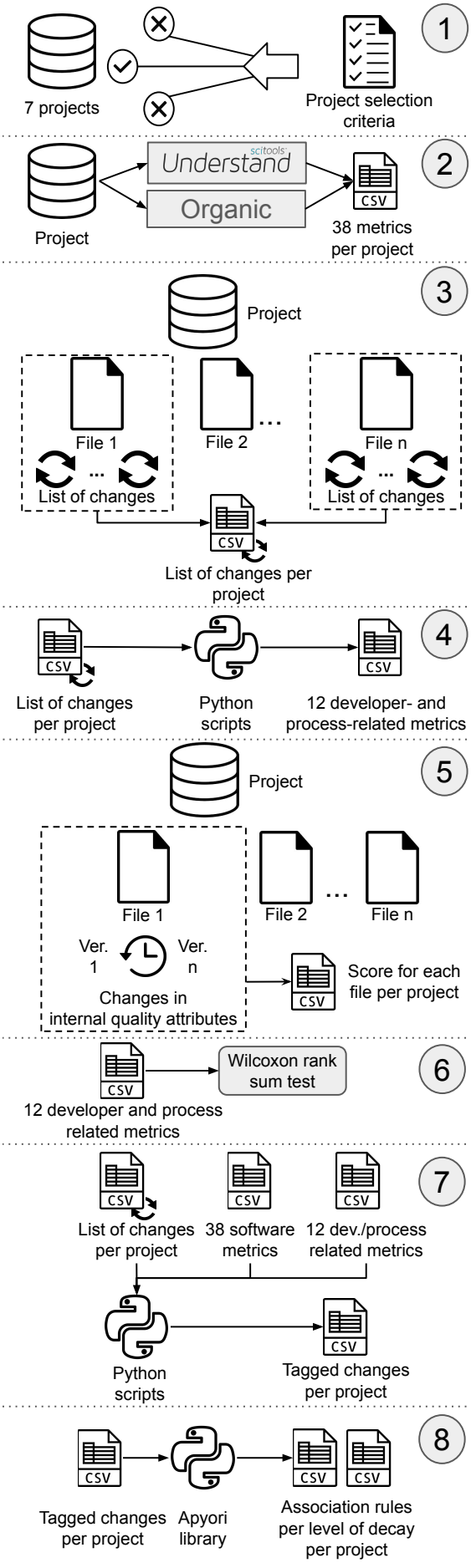

Figure 3.1: Overview of Study Steps

3.3.2.1 Step 1: Project Selection

Selecting systems for analysis: To select the open-source systems, we used the following criteria based on related studies [68]:

(i) the system must be mostly written in the Java programming language, due to the availability of available robust tools for software measurement and the fact that previous studies focusing on factors in isolation are conducted in Java projects (see Step 2);

(ii) the system must be at least approximately 5 years old, have at least 1k commits, and 250 pull requests [2], in order to provide us with enough data to effectively observe and measure our selected sub-factors (see Step 4); and

(iii) the system must be active as we would like to capture current development practices in our findings.

Table 3.1 provides details about each of the seven selected systems, including the name[3], the domain, number of commits, number of pull-requests, and time span we analyzed. We selected seven Java systems: *Fresco*, *Netty*, *OkHttp*, *RxJava*, *Presto*, *Dubbo*, and *S1*. The last system is a closed-source system developed by our industrial partners. We chose to include it as it may hint at possible differences between how to design decay happens in open and closed-source projects.

Table 3.1: General data of the target software systems

System	Domain	# Commits	# Pull Requests	Time span
Fresco	Library	2291	361	4.8 years
Netty	Framework	9322	5086	10 years
OkHttp	HTTP Client	3655	2709	7.8 years
RxJava	Library	5723	3407	7.7 years
Presto	Query Engine	16895	10039	7.5 years
Dubbo	Framework	4105	2462	8 years
S1	Web Application	2548	-	1,5 years

3.3.2.2 Step 2: Software metrics collection

Collecting software metrics of internal quality attributes: We used a non-commercial license of the Understand[4] tool and the freely available

[2]These thresholds were selected subjectively, and were based in criteria used in a previous work [7].

[3]We omitted the name of S1 due to intellectual property constraints.

[4]<https://www.scitools.com/>

Organic[5] [69] tool to collect a total of 38 metrics representing different properties of each IQA. For example, LCOM2 and TCC measure the lack of cohesion from different viewpoints and are each only computed by one of these tools. Moreover the selection of these tools was driven by the fact that they implement already validated metrics for design decay in previous studies [21, 30, 45, 53, 52].

Table 3.2 overviews the 14 used to measure cohesion, coupling, complexity, and inheritance. Each column lists: the IQAs, the metrics related to each IQA and descriptions for each metric, respectively. Table 3.3 display the remaining metrics 24 metrics used in this work, which are related to *size* attribute. It utilizes the sable column structure as Table 3.2. Additionally, we emphasize that the five IQAs used in this work have been chosen based in a variety of studies that use these metrics to characterize and quantify software quality [35, 21].

Table 3.2: Subset of software quality metrics used in this study

Attribute	Software Metric	Description
Cohesion	Lack of Cohesion in Methods 2 (LCOM2) [36]	Number of method pairs that do not share attributes, minus the number of method pairs that share attributes.
	Lack of Cohesion in Methods 3 (LCOM3) [38]	Treats each method pair as an individual entity, and determines the difference between the amount of similar and different pairs.
	Tight Class Cohesion (TCC) [39]	The number of directly connected visible methods in a class divided by the number of maximal possible connections between the visible methods of a class.
Coupling	Coupling Between Objects (CBO) [36]	The number of classes coupled to the analyzed class.
	FANIN [37]	The number of external classes that invoke methods from the analyzed class.
	FANOUT [37]	The number of external method invocations made by the analyzed class.
Inheritance	Base Classes (IFANIN) [43]	The number of immediate base classes and interfaces.
	Number of Children (NOC) [36]	The number of immediate subclasses to the analyzed class.
	Depth of Inheritance Tree (DIT) [36]	The number of nodes between the root of the inheritance tree and the analyzed class.
Complexity	Cyclomatic Complexity (CC) [40]	Measure of the complexity of a module's decision structure.
	Essential Complexity (ev(G)) [40]	Measure of the degree to which a module contains unstructured constructs.
	Maximum Nesting (MAxNEST) [41]	Maximum nesting level of control constructs.
	Weighted Methods per Class (WMC) [36]	The sum of the cyclomatic complexity of the methods of a class.
	Local Methods	The number of non-inherited methods declared in the analyzed class.

[5]While this tool is typically used as a code smell detection tool, it can also be used to collect software metrics. It is available at <https://github.com/opus-research/organic>.

Table 3.3: List of size metrics used in this work

Software Metric	**Description**
Physical Lines (NL) [42]	The number of physical lines in the class.
Blank Lines of Code (BLOC) [42]	The number of blank lines of code in the class.
Source Lines of Code (LOC) [42]	The number of lines containing source code in the class.
Declarative Lines of Code [42]	The number of lines containing declarative source code in the class.
Executable Lines of Code [42]	The number of lines containing executable source code in the class.
Lines with Comments (CLOC) [42]	The number of lines containing comments in a class.
Semicolons [42]	The number of semicolons in a class.
Statements [42]	The number of statements in a class.
Declarative Statements [42]	The number of declarative statements in a class.
Executable Statements [42]	The number of executable statements in a class.
Comment to Code Ratio [42]	The ratio of comment lines to code lines in a class.
Instance Variables (NIV) [42]	The number of instance variables in a class.
Instance Methods (NIM) [42]	The number of instance methods in a class.
Methods [42]	The total number of methods, including inherited ones, in a class.
Local Default Visibility Methods [42]	The number of local methods with default visibility in a class.
Average Number of Lines [42]	The average number of physical lines between methods of a class.
Average Number of Blank Lines [42]	The average number of blank lines of code between methods of a class.
Average Number of Lines of Code [42]	The average number of lines of source code between methods of a class.
Average Number of Lines with Comments [42]	The average number of lines containing comments between methods of a class.
Class Methods [42]	The number of class methods in a class.
Class Variables [42]	The number of class variables in a class.
Private Methods (NPM) [42]	The number of local private methods in a class.
Protected Methods [42]	The number of local protected methods in a class.
Public Methods (NPRM) [42]	The number of local public methods in a class.

3.3.2.3 Step 3: File history collection

Tracking system file evolution: As we aim to analyze factors related to decay at a class-level, we collected the evolution history for each file present in a specific version of the system. We considered the most current versions at the time of data collection, which is listed for each project in the replication package [70]. To mine the git history, we used the Pydriller library [71] and built a list of modifications for each file, from its introduction in the system to the most recent commit analyzed.

3.3.2.4 Step 4: Sub-factor metric collection

Collecting process- and developer-related metrics for tracked files: In this step, we implemented and collected the metrics representing the process- and developer-related sub-factors, introduced in Section 3.2.2, for each of the file changes previously collected.

Table 3.4 presents the 12 metrics collected in order to represent our sub-factors. The table columns present, respectively, the type of factor that contains each metric, each of the metrics collected (divided by the aforementioned factors), a description, and the rationale behind choosing each metric. These metrics were chosen as they represent different dimensions of each factor, and therefore complement each other. Specifically for refactorings, i.e. the implementation of metrics in the *Refactoring action* factor, we have utilized Refactoring Miner 2.0[6] [72] tool, due of its high precision and recall levels (98% and 87%, respectively).

3.3.2.5 Step 5: Design Decay Score

Differentiating classes based on how they were affected by design decay: In order to differentiate the files based on how to design decay affected their IQAs, we combined the metrics collected in Step 2 with the file history built in Step 3. First, we observed the variation of the metrics pertaining to four IQAs – cohesion, coupling, complexity, and inheritance – taking into account the difference between the creation of the file and its most recent version.

For instance, an increase in the cyclomatic complexity represents a deterioration (i.e., an observable event of decay), whereas a decrease represents an improvement. As seen, this interpretation can vary depending on the

[6] <https://github.com/tsantalis/RefactoringMiner>

Table 3.4: Process and Developer Metrics used in this study

Factor Type	Metric	Description	Rationale
Process-related	**Change outcome**		
	Change Set	Number of files modified alongside a file in a commit.	Large change sets are more likely to be reviewed by developers, due to having high chances of causing decay [11].
	Hunks Count	Number of distinct code segments modified in a file.	A higher number of code segments being modified in a class might indicate complex changes [12].
	Code Churn	Sum of the number of lines added and deleted in a file.	Classes having design problems are more change-prone.[13]
	Refactoring action		
	Move Refatoring Count	Sum of the number of refactoring operations related to motions, i.e, Move Method, Move Class and Move Attribute.	The amount of refactoring actions made on classes may indicate that developers are combating or minimizing design decay [15].
	Extraction Refactoring Count	Sum of the number of refactoring operations related to extractions, i.e, Extract Method, Extract Superclass, Extract Interface, Extract Attribute, Extract Class, Extract Subclass, and Extract Variable.	
	Hierarchical Refactoring Count	Sum of the number of refactoring operations related to hierarchies, i.e, Pull Up Method, Pull Up Attribute, Push Down Method, and Push Down Attribute.	
	Rename Refactoring Count	Sum of the number of refactoring operations related to renames, i.e, Rename Method, and Rename Class.	
Developer-related	**Discussion activity**		
	Number of Associated Pull Requests	Number of pull requests associated with a code change.	Changes with more pull requests associated are more complex, that may lead to design decay.
	Number of Words in discussion	Sum of the all words of each comment inside a Pull Request. Here we applied the preprocessing in the text removing contractions, stop words, punctuation, and replacing numbers.	Discussions with a high number of words are related to more complex changes, that may lead to design decay [17, 7].
	Number of Words per comment	Sum of the all words of each comment inside a Pull Request weighted by the number of comments. Here we applied the preprocessing in the text removing contractions, stop words, punctuation, and replacing numbers.	Discussions with a high weighted number of words are related to more complex changes, that may lead to design decay [17, 7]
	Contribution		
	Contributor Count	Number of distinct developers that have modified a file.	Classes touched by multiple developers are more prone to being degraded [19, 18].
	File Contribution Percentage	Percentage of the lines of a file that is by the same author as the current commit.	Minor contributors tend to degrade more the source code than code changed only by major contributors [19, 18].

semantics of each metric. Where needed, we used normalized variations of some metrics (e.g. LCOM2). We also did not consider size-related metrics in this step, since they, in isolation, may not reflect what developers consider as quality [35].

Thus, we ranked the classes in terms of how many[7] metrics indicated a deterioration in an IQA. To this end, we calculated the percentage of metrics from an IQA that have deteriorated, as shown on Equation 3-1, and computed the mean of this value for those four attributes, resulting in a score with range between 0 and 1 (shown on Equation 3-2).

$$Attribute\% = \frac{Number\ of\ deteriorated\ metrics}{Total\ number\ of\ metrics\ in\ this\ attribute} \tag{3-1}$$

$$Score = \frac{Cohesion\% + Coupling\% + Complexity\% + Inheritance\%}{4} \tag{3-2}$$

In general, code smells' detection strategies also consider multiple attributes, albeit not necessarily all at the same time or in the same smell [73]. Due to that, we've considered changes to all four attributes equally. It has recently been found that smells often co-occur in the same class, thus affecting somehow the four IQAs [74]. In cases where it did not affect all four attributes, it would at least impact three of them, with inheritance being the exception. We've also executed an additional analysis to ascertain the frequency of changes in a subset of changes in our dataset. This additional analysis is described in Section 3.4.2.

For each target system, this score was used to determine quartiles that classify the classes based on how their IQAs were affected by decay. Thus, if a class was below the 25th percentile, it was considered a *slightly-decayed class*. Conversely, if it was above the 75th percentile, it was considered a *largely-decayed class*. This can be seen in Equations 3-3 and 3-4. The output of this step, for each system, is two sets of classes (namely, slightly- and largely-decayed classes).

$$Bottom\ 25\% = Slightly\text{-}Decayed\ Classes \tag{3-3}$$

$$Top\ 25\% = Largely\text{-}Decayed\ Classes \tag{3-4}$$

[7]Previous studies have associated this diversity in the deterioration of IQAs with design decay [21].

3.3.2.6 Step 6: Individual Sub-factors and Decay

Assessing the relationship between individual process and developer-related sub-factors and design decay: To determine if (and which) process- and developer-related metrics are able to distinguish between slightly- and largely decayed classes, we used the *Wilcoxon Rank Sum Test* [75], since our metrics were not normally distributed [76].

Our test used the customary .05 significance level. We needed to adjust the *p-values* of the *Wilcoxon Tests*, since we performed multiple comparisons, to take into account the increased chance of rejecting the *null hypothesis* simply due to random chance. For this adjustment, we used the *Bonferroni correction* [76], which controls the *familywise* error rate. The group we use to represent the family is the *project*, i.e., the correction in the *p-values* apply to project-level metrics.

We also used the *Cliff's Delta (d)* measure [77] to investigate the magnitude of the difference between the two groups of classes. To interpret the Cliff's Delta (d) effect size, we employed a well-known classification [78] that defines four categories of magnitude: *negligible*, *small*, *medium*, and *large*. This classification can be seen as representing the probability that a random sample from each group will have different values [79]. Since, in this work, we expect to be mostly dealing with influences that happen infrequently, all effect sizes besides *negligible* can be impactful.

3.3.2.7 Step 7: Tagging commits

Tagging commits to track influential factors on design decay: To identify which factors interacted in each single commit and how this affected the IQAs, we applied a set of tags to each individual change based on related process- and developer-related metrics and on how the IQAs (and their respective metrics) were affected by the change.

Table 3.5 lists these tags and the conditions under which they are applied. The different states (i.e., +/- or High/Low) of certain tags are mutually exclusive. These tags allow us to identify interacting developer activities that happen prior or along the commit process, as well as how the change affected the IQAs. Thus, an antecedent-consequent relationship spanning the history of the project can be established through association rules.

Table 3.5: Commit Tags used in this study

Tag	Condition
(+/-) Coupling (+/-) Cohesion (+/-) Complexity (+/-) Size (+/-) Inheritance	Generated based on the variation of the related metrics. Positive changes lead to a positive score, and negative changes to a negative score. The resulting tag is based on whether or not the score sum is above or below zero.
+NumPullRequests +WordsInDiscussion (+/-) WordsPerComment	The tag is applied if there was a variation in the metric's value between two commits. Since the NunPullRequests and WordsInDiscussion metrics are cumulative, however, they cannot have negative values.
(High/Low) ChangeSet (High/Low) HunksCount (High/Low) CodeChurn	These tags were applied using a quartile-based strategy, calculated per project. If a change is in the top 25% (considering all classes) in a metric, it is tagged as High. If it is at the bottom 25%, it is tagged as Low.
+Contributor	This tag indicates the presence of a brand-new contributor in the class.
MajorContributor	This tag indicates that the contributor is responsible for over 5% of the class's code.
MinorContributor	This tag indicates that the contributor is responsible for less than 5% of the class's code.
MoveRefactoring ExtractionRefactoring HierarchicalRefactoring RenameRefactoring	This tag indicates that a refactoring from the specified category was detected in the class's code. This detection is done by RMiner.

3.3.2.8 Step 8: Mining association rules

Mining association rules from code changes: Using the changes collected per project in Step 3, plus their associated tags (collected in Step 7) as input, we executed the Apriori algorithm. For this step, we used an open-source library called Apyori [8], that implements this algorithm. As we were only interested in identifying meaningful association rules, we chose three criteria as the thresholds for rule creation:

(i) only the tags related to the process and developer sub-factors should appear as antecedents, since we intend to look specifically at interactions between their sub-factors as influential. Otherwise, the model would try to associate tags relating to the IQAs to themselves, creating a situation where they would be dependent and independent variables at the same time;

(ii) the consequent must only contain one or more tags related to IQA changes; thus, the rules inferred in this step do not consider cases in which there are no modifications to the IQAs; and

(iii) minimum support = 1% and minimum confidence = 30%.

Those thresholds were iteratively defined, initially starting from the default values of the used implementation. We arrived at these final values

[8]https://github.com/ymoch/apyori

from the subjective representativeness of the observations made on the results.[9] These observations made sure that those thresholds were adequate to create rules that, while maybe infrequent, are present in multiple of our software projects.

Using this procedure, we mined three sets of rules: rules found in slightly-decayed classes, rules found in largely-decayed classes and rules found in all classes. The former 2 groups of classes were defined in Step 5. In order to verify if there are associations common to all projects, we also repeated the same aforementioned steps considering changes in classes from all software projects. We refer to this set of association rules as the *All projects* dataset.

3.4 Results and Discussion

This section presents and discusses the results obtained in this study. Section 3.4.1 provides general observations about the raw data. Section 3.4.2 describes the results from the study steps that aim to answer our first research question. Section 3.4.3 describes the association rules mined in this work and discussed how they can be used to answer our second research question.

3.4.1 Slight vs. Large Levels of Design Decay

Assessing the decay score distribution. After executing the steps described in Section 3.3.2 we were able to observe how the decay score proposed by this work behaved in the seven software systems of our study. Figure 3.2 provides a violin plot that illustrates the distribution of decay scores per system. The violin plot used in the Figure displays data using a simplified box plot, with the thin lines in the edges of the plot representing the lower and upper quartiles, and the thick line representing the middle quartiles, separated by a white dot which is the median. They also visually display the distribution (or density) of elements by relying on the thickness of the colored shape that surrounds the box plot.

As we can observe, there is a significant difference between how slightly- and largely-decayed classes behave in terms of score. The group of slightly-decayed classes (i.e., classes in the lower quartile of decay score, presented in the graph) was densely grouped, which means they share some similarities in the amount of IQAs that have deteriorated. On the other hand, largely-decayed

[9]The possibility that some relationships were not detected due to the thresholds used is an inherent threat to the validity of studies of this nature.

classes (i.e., classes in the the upper quartile) were more sparse, showing that the degree as to which their IQAs decayed was more more varied.

We can also observe by looking at the distribution of the classes in each of the projects, that all projects behave quite differently, and only a few subsets of projects that behave similarly can be inferred, such as dubbo vs. RxJava, netty vs. okhttp, and fresco vs. presto. This observation implies that our project selection was able to successfully capture projects with different types of behaviour.

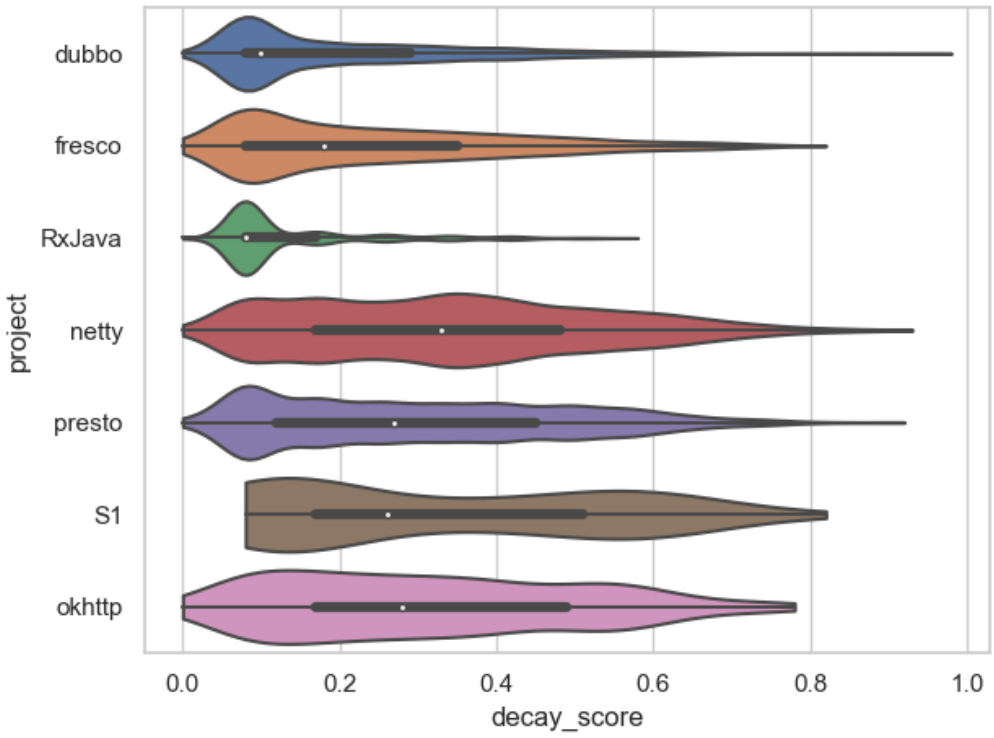

Figure 3.2: Violin plot showing the distribution of the decay score per system

Decay score adequation in a maintainability context. As as additional analysis, we observed whether the method used to calculate the decay score was adequate for our context, which is to improve maintainability. As such, we investigate instances of refactoring that might represent an attempt by the developer to remove or mitigate design problems, thus improving design quality, and consequently, maintainability. We computed the frequency in which each type of refactoring considered in this work (see Table 3.4) modified the four IQAs (see Table 3.2) considered when calculating decay score.

Table 3.6 shows the results. Each row represents a refactoring type. The columns presents, respectively, the total number of refactorings considered and the number of refactorings that modified each of the four IQAs: cohesion, coupling, complexity, and inheritance. The table displays the results aggregated

from all projects, as most of the projects showed similar characteristics. The results from each individual system are available in the replication package [70].

Table 3.6: Frequency of changes to IQAs in refactoring instances.

Refactoring Type	Total	Cohesion	Coupling	Complexity	Inheritance
Extraction	30125	24%	41%	32%	3%
Move	37424	25%	41%	30%	3%
Rename	11582	23%	40%	33%	3%
Hierarchical	960	23%	40%	33%	4%

Overall, all refactoring types consistently changed the four IQAs, albeit at different frequencies. While changes to cohesion, coupling, and complexity were more common (23% to 41% of instances), inheritance stood out as it was only changes in 3% to 4% of instances. Since each attribute contributes to 25% of the decay score computation, this result might be a possible explanation for why classes with decay score bigger than 0.8 are very infrequent (this distribution is shown on Figure 3.2). This result is in line with the rationale behind the decay score (see Section 3.3.2.5) and with previous results that investigated the relationship between IQAs and co-occurrences of code smells [74]. This result implies that, albeit not necessarily in the same change, developers trying to remove design problems tend to consistently modify the four internal quality attributes.

3.4.2 The Relationship Between Sub-Factors and Decay

To answer $\mathbf{RQ}_1$, we needed to first assess the relationship between each individual process- and developer-related sub-factor and decay, according to Step 6 of our methodology (see Section 3.3.2.6). Table 3.7 shows the results. Each column lists, respectively: the type of factors, the sub-factors (named via the metrics that represent them) split by those types, an classification of the Wilcoxon Test and d results for each system, and a summarization of the results considering the data from all systems. We employ a well-known classification [78] for representing the results, as shown in the table's legend. Cells containing N/A represent cases without enough data to execute the test (further discussed in Section 3.6). We discuss the metrics associated with factors that have been shown to differentiate between classes with different levels of design decay as follows.

Process-related factors and design decay. From Table 3.7, we observed that process-related metrics were the most statistically significant, i.e., able to differentiate between levels of design decay. Moreover, we observed

Table 3.7: Results of the Statistical Significance (p-value) of the Wilcoxon Rank Sum Test and the Cliff's Delta (d).

Factor	Metric	dubbo	fresco	netty	okhttp	presto	RxJava	S1	All
	Changes outcome								
	Change Set	(−)**	(−)**	(−)*	(−)*	(−)*	(−)*	(−)***	(−)**
	Hunks Count	(+)**	(+)**	(+)*	(+)*	(+)*	(+)*	(+)***	(+)*
	Code Churn	(−)**	(−)**	(−)*	(−)*	(−)*	(−)*	(−)**	(−)**
Process-related	**Refactoring action**								
	Move Refactoring Count	(+)	(+)	(+)*	(+)*	(+)*	(−)	(+)*	(+)*
	Extraction Refactoring Count	(+)*	(+)**	(+)**	(+)**	(+)***	(+)	(+)	(+)**
	Hierarchical Refactoring Count	(+)	(+)	(+)	(+)	(+)	(+)	(−)	(+)
	Rename Refactoring Count	(+)*	(+)*	(+)**	(+)***	(+)**	(+)	(+)***	(+)**
	Discussion activity								
	# of Associated Pull Requests	(+)**	(+)	(+)	(−)	(+)*	N/A	N/A	(+)*
	# of Words in Discussion	(+)**	(+)*	(+)**	(+)***	(+)***	N/A	N/A	(+)**
Developer-related	# of Words per Comment	(+)*	(+)*	(+)*	(+)	(+)*	N/A	N/A	(+)*
	Contribution								
	# of Contributors	(+)***	(+)**	(+)**	(+)***	(+)***	(+)*	(+)***	(+)***
	File Contribution Percentage	(+)	(+)	(−)	(−)*	(−)	(+)	(−)*	(−)

[a] Gray cells represent statistically significant differences, with ($p\text{-}value < 0.05$) between the two levels of design decay.

[b] The *Cliff's Delta (d)* effect size is shown as four categories of magnitude: *negligible* (no symbol), *small* (*), *medium* (**), and *large* (***).

[c] The positive d magnitudes are represented by the (+) symbol, and negative ones are represented by the (−) symbol.

that metrics related to change outcomes were good indicators, having at least a small magnitude in all cases and medium magnitude on 11 of 24 cases. These results are in line with the findings of previous studies [80] that also investigated process metrics as indicators of design decay. Regarding refactoring actions, while they had worse overall results than change outcome metrics, two sub-factors stood out: *Extract Refactoring Count* and *Rename Refactoring Count*. Both presented magnitudes higher than medium on 10 out of 16 cases and reached a large magnitude on presto, okhttp, and S1. Those results support the first finding of out study, as follows.

> **Finding 1**: Sub-factors that represent change outcomes are strong indicators to distinguish the level of design decay. Moreover, some sub-factors related to refactoring actions also stand out as good indicators of design-level change.

Developer-related factors and design decay. Back to Table 3.7, we observed that in general, the metrics presented statistical significance on almost all projects. However, the magnitudes were medium and large in only 13 of 34 cases. More specifically, when we are looking at metrics that consider discussion factors, we observed that the *Number of Words in Discussion by Class* metric stood out in comparison to the others, which presented a small magnitude in only one system (fresco) and large in two systems (okhttp and presto). This result might indicate that classes that contain large discussions are related to more complex changes, which are frequent in largely-decayed classes (as will be discussed on Section 3.4.3.1).

At the same time, the *contribution* factor is mostly represented by the *Contributor Count* sub-factor, which presented a large magnitude on five of the eight data samples, while the *File Contribution Percentage* only presented negligible and small magnitudes. In other words, the volume of contributions affecting the classes is a decisive factor in the characterization of largely-decayed classes. Those results lead us to this next finding.

> **Finding 2**: For developer-related factors, the density (i.e., number of words) of discussions and the number of contributors stood out as decisive sub-factors to differentiate between decay levels.

By leveraging these results, we are able to better study the influential associations discussed in the following section. Since there factors showed that they are individually related to design decay, they could be used as early indicators that a change should be carefully monitored, before certain

interactions that depend on factors that can only be identified later can happen. Further implications of these results will be discussed in Section 3.5.

3.4.3 Influential Associations and Design Decay

As the result of Step 8 of our methodology (described in Section 3.3.2.8) we collected a large number of association rules regarding three groups of code changes: changes made to all classes, changes made to slightly-decayed classes, and changes made to largely-decayed classes. Those groups were them composed of eight subsets, seven pertaining to each of the projects analyzed in this work and an eight that contain an aggregate of all projects. In total, those groups amount to 9131 rules, namely 3245 for all classes, 5096 for slightly-decayed classes, and 790 for largely-decayed classes. Due to the large number of rules, those rules are available in the replication package. However, findings presented in this subsection will provide examples of rules that are related to them.

To answer $\mathbf{RQ}_2$, we organized those association rules and manually analyzed the results (see Section 3.3.2.8). To find relevant associations between specific antecedents and consequents, we utilized a variety of visual aids, such as tables, visualizations provided by the arulesvis [81] R package, and an internally-developed interactive visualization. Figures 3.3, 3.4, and 3.5 present three examples of those visualizations generated by arulesviz, which visually represent the rules that aggregate data from all projects. The first (Figure 3.3) refers to the rules from all classes, the second (Figure 3.4) refers to the rules from slightly-decayed classes and the third (Figure 3.5) refers to the rules from largely-decayed classes. In all of these visualizations, the left-hand side (LHS) refers to the antecedents and the right-hand side (RHS) refers to the consequents. Moreover, rules in the figures are ordered by lift (for more information about association rules, see Section 3.2.3). Given the large amount of variation, especially in the antecedents, this visualization groups rules by similarity.

Due to the large number of association rules, we performed two different analyses. First, we obtained an overview of the entire ruleset by looking at the rules generated with the aggregated data from all projects. Second, we looked at the rules specific to each project. In cases where a large number of rules were present (over 100 rules), we analyzed the top 100, ordered by their *lift* value (see Section 3.2.3). We used the *lift* measure since it favors rules that had a bigger influence on the existence of the consequences. Such strategy also was used in previous works [29, 81].

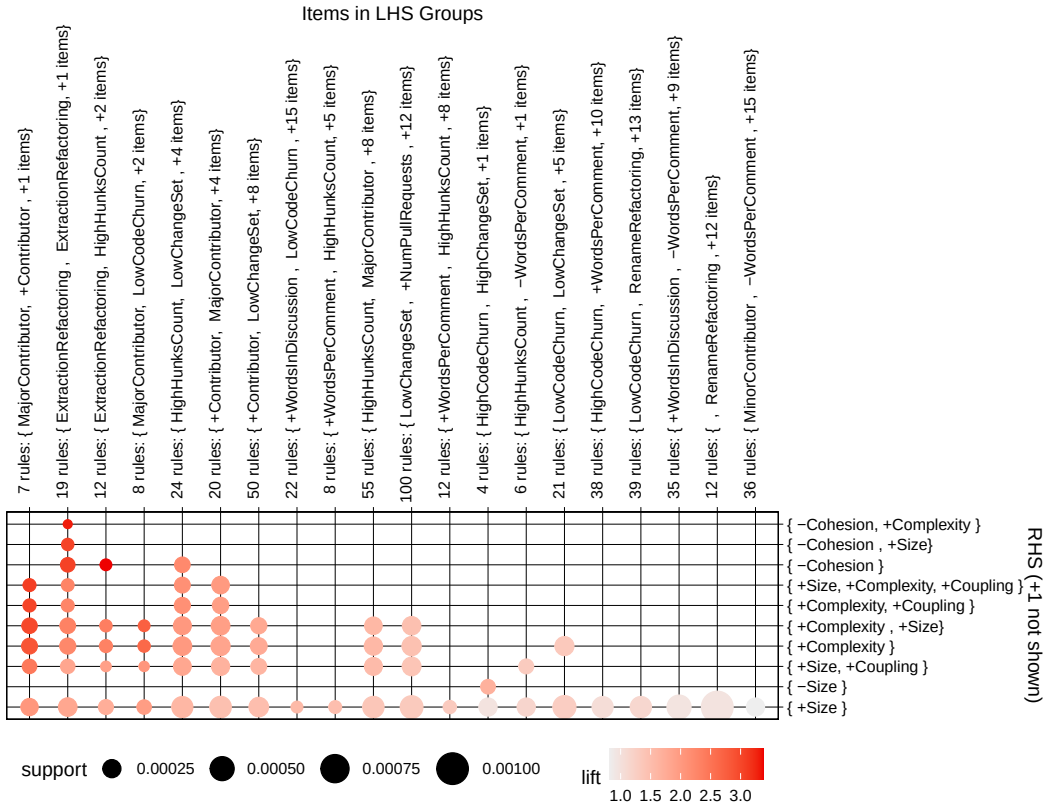

Figure 3.3: Grouped Matrix Chart [1] summarizing rules mined for all classes from all projects

We then separated the results between *general associations* – i.e., present in most, if not all projects – and *specific associations* – i.e., only appeared in one or two projects. By this division, we were able to better understand which associations could be caused by general practices, and which could be caused by project-specific practices or contexts. Finally, we have also performed a manual validation on 96 commits with tags related to some of the results presented in this section, aiming to understand how our findings correlate to the intent specified by the developers in commit messages and issue discussions.

3.4.3.1 General Associations Across Projects

We analyzed the set of association rules found in the *All projects* dataset (see Section 3.3.2.8). We emphasize that the following findings only apply to changes where at least one IQA was affected.[10] We observed that in largely-decayed classes only a small amount of rules (39) reached our support and confidence thresholds when compared to slightly-decayed classes. Moreover, most of those rules only altered the size IQA. The only other attribute modified

[10] 31% of the changes mined did not affect any on the IQAs and another 18% only affected the size attribute.

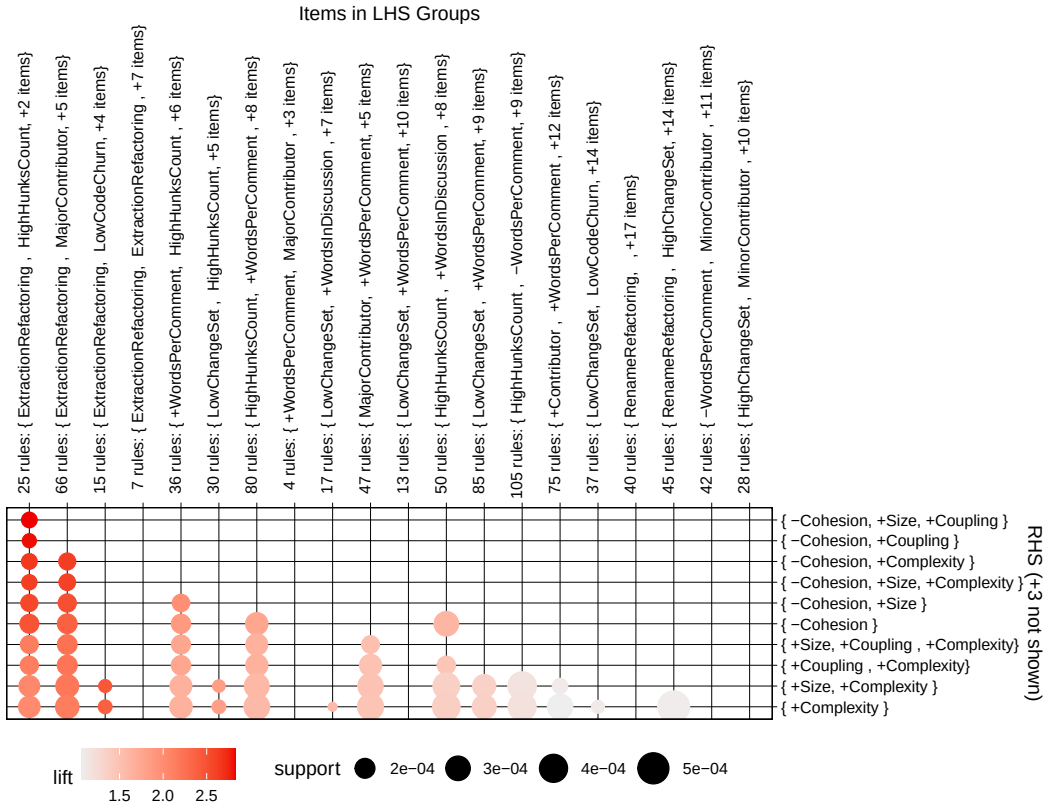

Figure 3.4: Grouped Matrix Chart [1] summarizing rules mined for slightly-decayed classes from all projects

was coupling. We conjecture that this low amount of rules is due to how varied the characteristics of code changes made to this group of classes were, making it harder for rules to reach our support and confidence thresholds. This variance can be seen in Figure 3.2, where we can observe that largely-decayed classes varied greatly. Nonetheless, for slightly-decayed classes, a large number of rules was generated (847 rules), with a great variety of consequents. Regarding the rules pertaining to all classes, all of them had consequents pointing towards negative effects.

In the top 100, we observed that most associations skewed towards negative effects, with 68 of the top 100 rules causing an increase in code complexity. Additionally, in this subset, the antecedents had little variety, with major contributors (i.e., contributors responsible for over 5% of the changed class's code) being present in 75 of the top 100 rules. While not as predominantly as in the slightly-decayed classes, the same behavior was also observed in the group of rules mined from all classes. It was also observed in the manual validation.

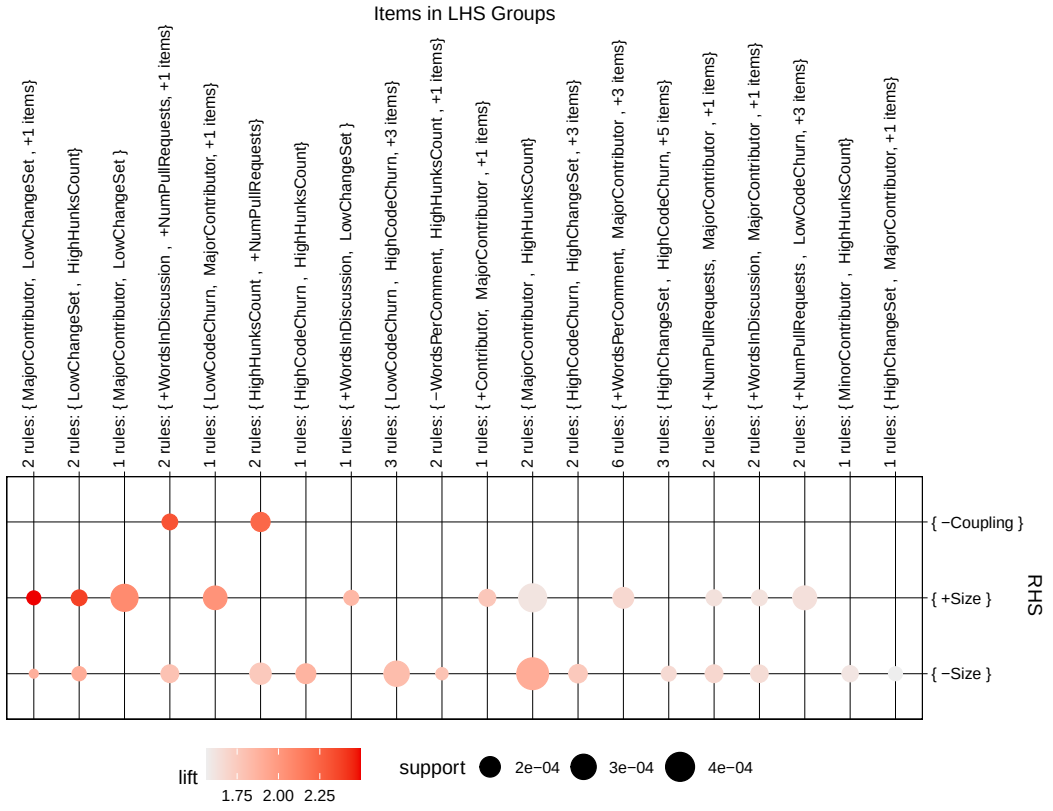

Figure 3.5: Grouped Matrix Chart [1] summarizing rules mined for largely-decayed classes from all projects

Finding 3: Slightly-decayed classes were frequently negatively affected by changes, even if marginally. However, changes to largely-decayed classes had no specific patterns, with their effects being mostly mixed. In the group containing all classes, all of the rules mined pointed towards negative effects.

One possible reasoning for the finding above, is that changes that have positive effects on the IQAs are affected more variedly by the sub-factors, and therefore those changes tend to not reach our detection thresholds. Negative changes, however, tend to occur in more uniform patterns. The same could not be observed in the case of largely-decayed classes because changes in the group are so varied that neither positive or negative changes reached our thresholds.

We believe this finding should be considered in the development of strategies to mitigate design decay. Since largely-decayed classes tend to be more varied, perhaps more targeted approaches could be more adequate when trying to perform repair actions in these classes.

Change outcomes and largely-decayed classes. Regarding the change outcomes, we observed that the presence of complex or large changes, i.e., high amount of hunks or high code churn, happened often in both largely

and slightly-decayed classes, and mostly lead only to changes in the size of the affected code. In fact, in largely-decayed classes, we noticed that around 50% of the rules caused a reduction of code size – which were always preceded by complex changes. Furthermore, when these large changes underwent a pull request process, which might imply a code review, they sometimes improved other IQAs instead of size, mainly coupling.

In terms of factors related to the change outcomes, we could see that the presence of large changes (with a high change set, a high amount of hunks, and high code churn) were very present in both slightly and largely-decayed classes, and almost always reduced the size of the changed code. In fact, in largely-decayed classes, we saw that around 50% of the rules caused a reduction of code size – and showed an interaction with the sub-factor representing complex changes (high number of code hunks). When these large changes underwent a pull request process, which might imply a code review, they also sometimes improved other internal quality attributes, such as reducing code coupling. However, when these large changes (high amount of hunks) are done in a small amount of classes (low change set), they instead tended to increase the size of the changed code.

Table 3.8 provides a list with the top 4 rules (ordered by lift) from largely-decayed classes in the All Projects dataset. Its columns present, respectively, the group in which the rule is present (all classes, slightly-decayed, or largely-decayed), its antecedents and consequents, and its support, confidence, and lift. This finding will be further discussed in Section 3.5. The two middle rules in this table are the only two rules in this group (largely-decayed classes from all projects) that modified IQAs other from size.

Table 3.8: Top 4 rules from largely-decayed classes from all systems.

Group	Antecedents	Consequents	Support	Confidence	Lift
Largely-Decayed	HighHunksCount, LowChangeSet	+Size	1.35%	43.99%	2.38
Largely-Decayed	+NumPullRequests, HighHunksCount, +WordsInDiscussion	-Coupling	1.33%	31.42%	2.32
Largely-Decayed	+NumPullRequests, HighHunksCount	-Coupling	1.77%	30.24%	2.23
Largely-Decayed	MajorContributor, LowChangeSet	+Size	3.44%	38.32%	2.07

3.4.3.2 The Most Common Associations Across Projects

We also performed a cross-project analysis to understand which associations commonly appear in multiple projects. Overall, even when looking at a project-level scope, the results described in Finding 3 continue to appear. Even with the larger sample size of rules, the effect of changes to largely-decayed

classes remained mixed, with changes having either a positive or negative effect on code quality based on the project and the performed actions. Slightly-decayed classes also maintain the negative effect observed in Finding 3.

Here, refactoring actions stood out, and we observed that, in general, refactorings were one of the two main motivators of changes to IQAs (and rules without refactorings tended to only affect size). Surprisingly, extraction refactorings tended to have a non-positive effect on the code, either only changing the code's size (both positively and negatively), or sometimes even worsening the other IQAs. These negative effects, while not very common in general happen more in situations where certain (sub-)factors interact, such as when these refactorings are performed with complex changes (high amount of hunks), or in a small number of classes (low change set). On the other hand, move refactorings usually had a positive effect on the IQAs. Moreover, we also observed that the interaction between movement and extraction refactorings in the same change is generally more positive than extraction refactorings alone. This pattern was observed in the entire dataset, albeit not present in largely-decayed classes (as previously stated, this could be due to our thresholds). It also presented itself more strongly in the data set containing rules from all classes than in the slightly-decayed classes. Regarding the intent, we observed in our manual validation that a majority of the changes in our data set had non-refactoring goals, with refactorings only used as a step in a larger process. However, we noticed that when move refactorings are performed with an explicit refactoring goal, they are more likely to have positive effects.

Finding 4: Extraction refactorings often had a non-positive effect on the IQAs. Conversely, move refactorings had a mostly positive effect. Specific interactions between (sub-)factors enables one to better understand those effects. For instance, extraction refactorings being more likely to be negative when applied in complex changes, and being less likely when applied alongside a move refactoring.

Table 3.9 provides examples of rules that were used to formulate this finding. Its columns present, respectively, the group in which the rule is present (all classes, slightly-decayed, or largely-decayed), its antecedents and consequents, and its support, confidence, and lift. This finding will be further discussed in Section 3.5.

Previous contribution did not reduce the likelihood of extraction refactorings being negative. Another interesting interaction observed, was that even when the developers were experienced and major contributors to that class, their extraction refactorings (often in combination with complex

Table 3.9: Examples of rules used to formulate Finding 4

Group	Antecedents	Consequents	Support	Confidence	Lift
Slightly-Decayed	HighHunksCount, ExtractionRefactoring	-Cohesion, +Coupling	1.04%	32.24%	2.80
Slightly-Decayed	HighHunksCount, MajorContributor, ExtractionRefactoring	-Cohesion, +Complexity	1.08%	37.43%	2.73
All Classes	ExtractionRefactoring, HighHunksCount, MajorContributor	-Cohesion	1.10%	48.80%	3.42
All Classes	ExtractionRefactoring, MajorContributor	-Cohesion, +Complexity	1.01%	31.32%	3.30

changes, a third interacting factor) also frequently caused negative effects on the code. It is important to note that these negative effects may be mostly minor in slightly-decayed classes, given that they are in the group that decayed the least. However, these effects might still mean a gradual decay of the affected classes, and are still potential causes for future concern. In the manual validation, changes with both refactoring and non-refactoring goals, respectively, were common with major contributors. The latter mostly had negative effects, while the former was usually positive, yet coupled with a side-effect that could be seen as negative (i.e., an increase of inheritance tree depth through the usage of *Extract Superclass* refactorings).

Level of contribution and slightly-decayed classes. We observed that, for developer-related factors, interactions between certain (sub-)factors frequently appeared. There were often cases in slightly-decayed classes in which new contributors changed over 5% of the code. Those changes usually interacted with additional factors as well, and modified only a small number of classes.[11] While less frequent, this pattern also presented itself in the rules mined from all classes. Those interactions showed a variety of negative effects, and specifically in slightly-decayed classes, they continued to occur regardless of whether the changes interacted with the associated pull requests (sub-)factor, which may imply that a code review process did not influence this. In the manual validation, the most common changes by new contributors were small functional changes, followed by refactorings. Rarely were large functional changes performed. Functional changes usually had negative effects on code quality, while refactorings had a more mixed effect.

Table 3.10 provides a list of examples of rules that were used to formulate the following finding. Its columns present, respectively, the group in which the rule is present (all classes, slightly-decayed, or largely-decayed), its antecedents and consequents, and its support, confidence, and lift. This finding will also be further discussed in Section 3.5.

[11] Cases where those changes affected a large number of classes were also observed, but were less frequent.

Table 3.10: Examples of rules used to formulate Finding 5

Group	Antecedents	Consequents	Support	Confidence	Lift
Slightly-Decayed	+Contributor, MajorContributor, LowChangeSet	-Cohesion, +Size, +Complexity	1.01%	38.28%	2.86
Slightly-Decayed	+Contributor, HighHunksCount, MajorContributor	-Cohesion, +Size, +Coupling	1.39%	30.94%	2.83
All Classes	LowChangeSet, MajorContributor, +Contributor	+Size, +Complexity, +Coupling	1.29%	47.40%	3.08
All Classes	LowCodeChurn, LowChangeSet, MajorContributor, +Contributor	+Size, +Complexity	1.06%	61.12%	2.85

Finding 5: In slightly-decayed classes, an specific interaction, first time contributors that changed a significant (over 5%) portion of a class' code, tended to cause decay – even if these changes might have gone through a pull request process.

3.4.3.3 Specific Associations per Project

For this set of associations, we analyzed each project individually to investigate more context-specific associations. Thus, we found that certain associations behaved differently than the general associations, or even those that were found in other projects. One such case of this was in Fresco's slightly-decayed classes, where changes showed frequent improvements to code quality – when other projects, in general, had mostly negative effects. Another example is related to OkHttp's slightly-decayed classes, which, differently from other projects, showed large changes mostly having negative effects, increasing the code's complexity.

There were also interesting observations due to project-specific contexts, providing potential insight into how certain development practices can lead to certain results. For instance, in Dubbo's low decay classes, new contributors often brought negative consequents – if the number of changes was low, the complexity increased; if it was high, the cohesion and coupling worsened. Project S1's largely-decayed classes were the only case in which inheritance metrics were changed by rules with a considerable lift, with these changes being negative, and caused alongside extraction refactorings.

For Netty's largely-decayed classes, we observed that when there was an increase in the number of pull requests related to a specific class, the size of that class tended to increase. In Presto's largely-decayed classes, changes associated with short comments, with a small number of words (i.e., comments that reduced the mean number of words per comment), always had positive effects, reducing both coupling and size. Finally, in Presto's slightly-decayed

classes, even though major contributors to such classes were very present, changes made by them had mostly negative effects on the code.

Closed Source vs. Open Source: As we observed in Figure 3.2, software project S1's decay score class distribution was the only that was not similar to any other software project, as it was the only one who did not have any classes approaching zero decay score. These differences also manifest when looking at the association rules mined for S1.

First, they did not follow the previously reported findings about extraction refactorings and new developers. In fact, for the slightly-decayed classes group and the all classes group, the only meaningful interactions between factors were regarding large and complex changes that modify few files. Those interactions had mostly negative effects.

Second, all of the top interactions for its largely-decayed classes that had effects on an attribute other than size, involved extraction refactoring interacting with different sub-factors from the change outcomes factor. But, contrary to the previous findings, all of these interactions had negative effects on only one attribute, which was *inheritance*. Since the confidence of some of these rules was abnormally high, with some of them reaching 100%, a manual investigation revealed that these results were due to the low amount of extraction refactorings mined for this project. This implies that new studies, with access to additional and larger projects, are needed to effectively investigate the differences between closed- and open-source projects.

3.4.3.4 Manual Validation

For the manual validation, we selected a set of 96 commits, split by the seven different projects we analyzed, and then further split into five groups: two groups of changes that contain tags that match with the rules used to formulate our findings 4 and 5, one group exploring other interactions between refactorings and other factors, one for a random sample of slightly-decayed classes, plus one with a random sample of largely-decayed classes. This led to an average of 15 commits per project (as some projects did not have commits that fit in a specific subcategory/finding). To execute the validation the three participants (one PhD and two graduate students) were each given a single code change per project per subcategory and had access to the following information: the hash of the commit containing the change, its contents, the specific file in which the rules were found, and the tags for that change. From that information, they had to describe, where possible, the intent of the developer when performing those changes, as well as if (and how) they

relate to one of the findings in the paper.

In summary, from the results, we were able to see that – for Finding 4 – 12 out of 21 commits were directly in line with the finding, while most of the other nine commits were "tangentially" related (a refactoring occurred together with another non-refactoring change that caused decay). For finding 5, 12 out of 15 commits were directly in line with the finding, with the other three commits being unrelated to the finding (but not contradicting it). Finally, from the refactoring group, two non-related commits were found due to false positives in the refactoring detection. Some additional results have also been observed, and were previously mentioned in earlier parts of this section. The full data for the validation is available in the replication package [70].

3.5 Study Implications

Our study provides findings that lead to implications for researchers, tool builders and practitioners. They are discussed as follows.

3.5.1 Sub-factors as Indicators of Design Decay

Researchers can use a variety of interacting (sub-)factors for differentiating design decay: Findings 1 and 2 shows that process-related factors and some types of developer-related factors are strong indicators for distinguishing design decay. First, these observations confirm the findings of previous studies [9, 11, 80] on the use of process-related metrics as indicators of complex changes leading to design decay. Second, they confirm findings of recent studies [7, 9] reporting that the number of words in developer discussions could be used to distinguish different levels of design decay.

Our study also advances the state of knowledge by revealing that the number of associated pull requests, either high or low, is unable to distinguish different levels of decay. Initially, one could suspect that as the number of pull requests grows – with more features, bug fixes, and refactorings being requested – design would progressively decay, which would be in line with Lehman's software evolution law [2]. However, that was not what was observed. This is an important finding, given that the introduction of a change via a pull requests could imply a code review process occurred. Moreover, our study indicates that the high number of refactoring actions grouped by transformation nature – i.e., extraction and move – performed on a class can be used as indicators of design decay. Finally, we showed how several key interactions between factors could affect design decay. These aforementioned results could be used by developer

to build preventive approaches to avoid design decay. For example, they could be monitored as early indicators that a change or class should be carefully watched.

Researchers might also be able to explore other types of factors (and their interactions) that may help to further differentiate design decay. It is also important to investigate the actual main root-causes for design decay to occur – these causes certainly go beyond the characteristics of both the changes themselves, as well as the developers that perform them. Our study hints at some of these causes, confirming that certain kinds of changes require more attention from developers as they are highly related to design decay.

3.5.2 Refactorings' Side Effects

Developers should be more conscious about refactorings' side effects: Finding 4 (see Section 3.4) shows that, while developers are performing a variety of refactoring actions, some of them (in most cases, extraction refactorings) are often having a non-positive effect on the code. This happens even when this refactoring interacts with other factors such as it applied by developers with high ownership of the original code.

Our observations contradict previous studies on refactorings. However, recent works have started to link extraction refactorings to negative effects [30]. A previous study states extraction refactorings are related to positive changes to size as well as not usually worsening other IQAs [21]. One interesting example found in our manual validation shows an extraction refactorings worsening a class's IQAs.[12] In this change, the developers performed a variety of complex extraction refactorings, which did slightly improve a few of the affected classes, but caused bloat on another class, by adding quite a lot of new (extracted) methods to its method list.

Conversely, move refactorings had a positive effect on the code quality; almost all the cases change IQAs positively. We have also observed that whenever there is spatial interaction (i.e., same class and same commit) or temporal interaction (i.e., same class and different commits) between extraction refactorings and move refactorings (two process-related sub-factors), that change is less likely to cause decay. Extraction-only refactorings often impacts the target class's interface and how it communicates with its clients [82]. However, this sub-factor alone may not be a consistent indicator of design decay (as we observed here), as previous studies have reported [30].

[12]Located in commit 957537774b319bb0109819258a11af78a98bcb97 from the OkHttp project. Available in <https://github.com/square/okhttp/commit/957537774b319bb0109819258a11af78a98bcb97>.

Thus, developers might need to consider the implications of an extraction refactoring's application, to prevent it from potentially bloating the target class or method – especially in cases of specific interactions, such as the changes being applying alongside other, already complex, changes. More attention should also be given by researchers and tool builders to better support this type of situation. For instance, tools should be adapted to track problems in larger, more complex changes, which are the refactorings that tend to have the most negative effects. Refactoring recommendation systems should also consider there side effects when suggesting changes.

3.5.3 Changes by Major or First-time Contributors

Special attention should be taken when reviewing code by first-time contributors: In Finding 5, we also highlighted another frequent interaction between factors, that when first-time contributors make large changes to a class's code, that change tends to lead to decay, regardless of code review. This might mean that developers should be more cautious while reviewing changes performed by contributors who have not performed previous changes to the target class, especially if this new change affects a large section of the class or a large variety of classes. Where possible, the task distribution to first-time contributors should also be carefully considered. For instance, granting them smaller tasks first, while providing constructive feedback in reviews, so they gain a better understanding of the code in question, to then later grant them tasks that require larger changes to the code.

Reviewers should pay attention even when reviewing code by experienced contributors: We've also observed that one of the biggest motivators for changes to the IQAs are changes made by major contributors (are authors of at least 5% of a class' code). We conjecture this could happen due to the developer already knowing the class and being more confident to perform changes to the design of a class. The most frequent interactions we encountered involving major contributors were negative. This happened especially if this sub-factor interacted with the sub-factor representing extraction refactorings, which was the strongest interaction observed in terms of lift. Thus, we believe that reviewers should have the same care they would have when reviewing code by a newcomer, even if the author of the changes is already knowledgeable about a class.

3.5.4 The effects of large changes

Large changes are not necessarily detrimental: While previous work associate large changes with design problems [11, 13], our results show that large changes are very present in all of our analyzed projects and do not necessarily cause negative effects – most of them having a neutral, and, when undergoing a pull request process, sometimes even positive effect. Thus, this might imply that large changes might not have such a majoritarian negative effect on the quality of the changed code. In fact, we have found that, overall, they have neutral or mixed effects. This finding also reinforces the importance of code review in the development process, as the large changes tended to be more positive when introduced through a pull request.

3.6 Threats to Validity

We discuss threats to the study validity [67] as follows.

3.6.1 Construct and Internal Validity

We analyze design decay in terms of five IQAs. Thus, our findings might be biased by them, even though they are commonly investigated in other works [21, 30, 22]. The metrics chosen to capture properties of the IQAs may not be appropriate. In fact, metrics alone might not be enough to capture external factors that might influence decay – e.g., developer intentions and design decisions. However, we were particularly interested in the quantified version of decay, represented by these metrics. To mitigate this, we chose a non-random set of metrics that assess different properties of each IQA based on well-known catalogs [40, 37, 36, 41]. Regarding process- and developer-related metrics, some of them are based on heuristics, e.g., we have assumed that the number of major contributors to a class is the number of developers that contributed at least 5% LOC to this class. Although this is a limitation of measuring such factors, we rely on known heuristics to recover this information [19, 6, 9].

As mentioned in Section 3.3, while we have meticulously chosen our thresholds, it is possible they might cause some relationships to not be found by the Apriori algorithm. We also evaluated the Pearson correlation coefficient to measure the correlation between metrics for each factor, and found that some metrics were correlated. However, we tested our models without their presence and found that their absence weakens the models, and thus, maintained these

metrics. Concerning the lack of discussion activity metrics on RxJava and S1 projects: (i) the RxJava classes where design decay was found were not inserted on the main branch by pull requests; (ii) since S1 is closed-source, we only had access to the source code on a git environment. Thus, we did not have any pull requests data for these two projects. Moreover, because this lack of data, we were not able to apply the analysis of the RQ1, on the Discussion activity factor, for the RxJava and S1. Finally, we will improve our project selection on future work to avoid this type of threat.

3.6.2 Conclusion and External Validity

We carefully performed our descriptive and statistical analyses. All analysis results were double-checked by two paper authors aimed to mitigate biases and the misapplication of analysis procedures. Our study focuses on investigating the design decay of Java projects only. Nevertheless, we highlight that Java is one of the most popular programming languages in both industry and academia. Additionally, although we have assessed both open and closed source projects, the number of closed source projects is quite low (only one project) when compared to the number of open projects (the other six projects). Hence, collecting data from more (in particular closed source) projects and conducting such additional analyses is part of future work.

3.7 Related Work

We discuss previous works related to the current study, as follows.

3.7.1 Empirical Studies on Factors that Affect Design Decay

There are multiple recent studies about factors related to design decay [6, 7, 18, 21, 22, 9]. Many of them only use code smells [6, 7, 9] and software metrics [18, 21, 22] as symptoms for the identification of design decay. Uchôa et al. [6] observed that the majority of code changes performed by developers in code reviews have an invariant impact on the evolution of design decay. They also analyzed the relationship between design decay and the influence of factors related to developer's participation, code review intensity, and reviewing time. They concluded that certain code review practices, such as long discussions and a high rate of reviewers' disagreement, might increase design decay risk. Despite such results, the author does not consider factors related to refactoring

actions and contribution. In addition, they do not measure the decay in terms of IQAs.

Barbosa et al. [7] investigated the impact of 11 social metrics related to two social factors on design decay: communication dynamics among developers – who play specific roles; and the discussion content itself. The authors observed that many social metrics could be used to discriminate whether code changes had an impact on design decay. Finally, the authors noticed certain metrics tend to be indicators of design decay only when analyzing both aspects together. Similar to the previous study, the authors do not consider refactoring actions and contribution factors, and neither IQAs.

Capiluppi et al. [18] investigated whether the work of multiple developers and their experiences has an effect on the structural quality metrics. The authors observed that the experience of developers plays a key role: the more inexperienced developers tend to degrade more the source code than the code changed only by experienced developers. They also observed that the decay in structural quality metrics is linked to an increase in further maintenance: when more developers work on the same code, its structure degrades and the number of further commits needed increases. This is even more visible when less experienced developers have worked (or still work) on the code itself. However, the authors do not track how the influential factors are associated with the progression of the decay, and how other developer-related factors, e.g., discussion activities, might distinguish the levels of design decay.

3.7.2 Empirical Studies on the Use of Association Rules

Different studies have used association rules for finding patterns and to extracting knowledge about the different aspects of influencing software development and evolution [29, 83, 84]. For instance, Soares et al. [29] have used association rules to identify characteristics that influence the rejection of pull requests by team members in projects with high acceptance rates. The authors observed that some key factors increase the chances of having internal contributions rejected: (i) physical characteristics and complexity of changes, as well as the location of the modified artifacts; (ii) previous experience with pull requests; and (iii) the project's contribution policy.

Mondal et al. [83] identified co-change patterns to detect hidden dependencies among different parts of the system. More specifically, the author detected a co-change pattern to support the identification of methods logically coupled with other methods. Despite using association rules, their work is centered at the method-level while we focus on the class-level. We also con-

sidered changes to coupling as a consequent on the association rules. Finally, Zimmermann et al. [84] proposed an approach that relies on association rule mining to suggest possible future changes (e.g., if class A usually co-changes with B, and a commit only changes A, a warning is given suggesting to check whether B should be modified too). Conversely, own study aims to identify recurrent patterns that affect design decay on classes, rather than recommend co-changes.

In summary, our work differs from existing ones as follows: (i) we investigate how process- and developer-related factors can be used to distinguish between varying levels of class-level design decay; (ii) we track how the influence of multiple factors simultaneously can affect decay; and (iii) we use association rules to infer and assess relationships between factors and decay.

3.8 Conclusion and Future Work

In this work, we investigated the relationship between two groups of influential factors: process- and developer-related, and design decay itself. Our results indicate that: (i) both types of factors can be used to distinguish between different decay levels in classes; (ii) changes to largely-decayed classes had a mostly mixed effect on quality, while slightly-decayed classes suffered more negative changes; (iii) extraction-type refactorings had a non-positive effect, in contrast with other types of refactorings, even when interacting with factors that represent developer experience; and (iv) the interaction of factors that represent inexperienced contributors and large changes, or refactorings, tend to harm the design quality, even when they interact with factors that may represent a code review process. We expect our findings to aid developers and researchers in improving their guidelines on how to avoid and monitor decay.

As future work, we intend to explore whether and how our observations differ in terms of closed and open source systems. Moreover, we intend to explore whether classes that are added to a system with preexisting design problems behave differently from our findings. Finally, we aim to investigate additional factors.

3.9 Summary

In the paper presented in this chapter, we address the problems addressed in this paper as follows. Before tacking each of our specific problems, we individually investigated each of our sub-factors (described in Section 2.2) to make sure that they have a relationship with design decay. This investigation

refers to the first research question of both this dissertation and the paper. In this investigation, we concluded that 7 out of our 12 sub-factors can indeed be used individually to differentiate between classes that suffered different degrees of design decay. This finding relates to our general problem – i.e., a lack of resources[13] about how to avoid or mitigate design decay – as it provides an initial view about which sub-factors should be monitored in order to avoid or prevent design decay. These results also refer to the first research questions of both this dissertation and the paper.

After establishing this relationship, we then tackled both of our specific problems, i.e., which associations (or interactions) between different factors can be inferred and how they affect design decay, and whether the behaviour of these interactions changes depending on the level of decay of the affected class. To do this, we used association rule mining to collect a number of influential interactions between factors, separated in three groups: interactions that happened in all levels of decay, those that happen in low levels of decay, and those that happen in high levels of decay. Using this technique, we reported several different findings that exemplify how some key interactions between factors can influence design decay (discussed in Sections 3.4.3 and 3.5). The next chapter summarizes the main contributions of this work, including aforementioned results and their implications. It also presents opportunities for future work aimed to refine, strengthen, and expand this research.

[13] As discussed on 1.1, we consider resources as information that can improve awareness of influences that different interactions between factors can have on design quality.

4
Final Remarks

Changes to design happen regularly through the maintenance and evolution of a software project. These changes can introduce design problems causing a phenomenon called design decay. In this work, we investigate this decay process via symptoms that manifest themselves as changes to four internal quality attributes: cohesion, coupling, complexity, and inheritance.

A plethora of factors can also affect this process, often simultaneously. Existing studies do not focus on these interactions between factors, mostly studying them individually. Since this process is assymetrical (i.e., classes are not affect equally), developers might need different strategies to mitigate or remove design problems depending on the level of decay of the target class.

This dissertation aims to reveal whether and how are two groups of factors, developer- and process-related factors, relate to decay. First, we analyzed, through statistical tests, whether their sub-factors are indeed related to design decay. Second, we used association rule mining to discover a number of meaningful interactions between factors and how their behaviour differed depending on the level of decay of the classes affected.

4.1
Contributions

In summary, the main contributions and their implications are described as follows.

Contribution 1: *Methodology Design that Avoids Common Problems from Code Smell-based Studies.* In this work, we designed a methodology which is able to observe symptoms of design decay by using the internal quality attributes. It avoids one of the main problems typically associated with code smell based approaches, which is the choice of thresholds for code smell detection. Since thresholds are subjective and require additional decision making, it can be advantageous to avoid using them them. Nevertheless, using an approach that differs from conventional ones can be valuable as it can possibly detect previously unnoticed symptoms.

Contribution 2: *A Set of Insights Relating Interactions Between Process- and Developer-Related Factors to Design Decay.* As stated in our problem statement (Section 1.1), prior to this dissertation, there was little information about how interactions between factors can affect design decay. By following the methodology described in Section 3.3.2, we were able to obtain

several insights and implications which could improve the awareness practitioners, tool builders, and researchers about how these interactions can affect decay. After a more in-depth investigation, these insights could also be used formulate practical guidelines for design decay avoidance. In summary, our main insights include:

- first, we reported on the viability of using the 12 sub-factors (Section 2.2) analyzed in this work as early indicators of design decay. We found that 7 out of the 12 sub-factors are good individual indicators of design decay. This information could be used by practitioners for early detection, as those sub-factors could be indicators that a class or change should be closely monitored, even if a meaningful interaction has not yet happened.
- second, we observed several side effects present in changes containing extraction refactorings. Interactions involving this type of refactoring often had negative effects on design quality. This contradicts previous works on refactoring. On the other hand, another type, move refactorings, had mostly a positive influence on design quality. In Section 3.5, we provide a real example of how an extraction refactoring introduced such negative effects. This finding could be used by developers as a warning to be careful when performing or reviewing code containing an extraction refactoring. Tool builders that develop tools that recommend refactorings could also take these side effects into consideration.
- third, we also reported on the negative effects of changes performed by newcomers. Interestingly, here we observed an interaction involving a sub-factor that was considered influential when in isolation: the contributors sub-factor. This interaction happened when a newcomer performed a large change, causing negative effects on design quality, which remained even when that change was reviewed by other developers. This finding could be used by code reviewers to identify riskier changes that need to be more carefully reviewed. It should also be possible to automatic this process, e.g., a bot could warn the reviewer when it identifies this (or other related) interaction.

Contribution 3: *Slightly-Decayed Classes vs. Largely-Decayed Classes.* This dissertation also provides several insights about how classes with different levels of decay behave differently. Some of these insights are described as follows.

- we observed the distribution of decay scores throughout the classes of the seven different systems, in order to find differences in their behaviour.

While we were able to group some software projects in terms of similarity, none of these groups had more than two projects. We also found that classes with low levels of decay tend to be more similar, while classes with high levels of decay vary widely. This should be considered in the development of strategies to mitigate design decay. Since classes that decayed more tend to be more varied, targeted approaches could be more adequate when trying to perform repair actions in these classes.

– in terms of association rules, we observed several behaviours which were exclusive to the group with a specific level of decay. For example, we were able to identify an interaction where large changes to classes with high decay had positive results on design quality when its code was reviewed prior to introduction in the system.

Contribution 4: *A Number of Secondary Contributions.* We also provide a number of secondary contributions, such as:

– the development of several scripts, used to collect and analyze data through this dissertation, e.g., scripts to collect historical data from git repositories, to use software or developer/process-related metrics to tag changes in order to establish co-occurrences, among others;

– the detailed design and implementation of metrics quantifying the 12 sub-factors (Section 2.2) studied in this work.

– a data set containing the raw and processed data collected and used in each Step of our methodology (Section 3.3.2). It is available as part of our replication package [70].

4.2 Future Work

For future work, we plan to consider new types of factors, such as the ones mentioned in Section 2.2 (e.g. technical factors, collaboration factors, organizational factors, etc.) and utilize novel data analysis techniques in order to understand how they affect software development, while also expanding the scope of the analysis done to previously studied types of factors (e.g. applying natural language processing to investigation whether discussion content influences decay.).

Besides that, we also plan to increase the scope of our data set, e.g. both in terms of the number of projects and, as aforementioned, of the number of factors, and consequently, sub-factors that are considered. For example, in the refactoring actions factor, we plan to consider additional types of refactorings,

which may not necessarily be related to maintainability, comprehensibility, and reusability. We also plan to consider self-affirmed refactorings, which are refactorings which are explicitly documented by developers when performing a change. Some of the additional types of factors mentioned in Chapter 2.2 are also planned to be included. We also plan to perform a qualitative analysis that would enable us to obtain a more holistic view of how design decay affects software and its developers in practice, by looking at how the developers perceive each factor and how it affects decay.

Finally, we intend to elicit which requirements automated techniques that aim to avoid or mitigate design decay have to conform with to be effective, based on our findings. After this, we plan to investigate if existing techniques or tools actually meet those requirements, and if needed, design and evaluate an automated technique or tool of our own.

5
Bibliography

1 HAHSLER, M.; KARPIENKO, R. Visualizing association rules in hierarchical groups. **Journal of Business Economics**, Springer, v. 87, n. 3, p. 317–335, 2017. Cited 4 times in pages 11, 58, 59, and 60.

2 LEHMAN, M. M. Programs, life cycles, and laws of software evolution. **Proceedings of the IEEE**, IEEE, v. 68, n. 9, p. 1060–1076, 1980. Cited 3 times in pages 14, 36, and 66.

3 PARNAS, D. L. Software aging. In: **16th ICSE**. [S.l.: s.n.], 1994. p. 279–287. Cited 4 times in pages 14, 23, 36, and 38.

4 LI, Z.; AVGERIOU, P.; LIANG, P. A systematic mapping study on technical debt and its management. **J. Syst. Softw. (JSS)**, Elsevier, v. 101, p. 193–220, 2015. Cited 3 times in pages 14, 23, and 38.

5 SOUSA, L. et al. Identifying design problems in the source code: A grounded theory. In: **40th ICSE**. [S.l.: s.n.], 2018. p. 921–931. Cited 5 times in pages 14, 23, 36, 38, and 39.

6 UCHôA, A. et al. How does modern code review impact software design degradation? an in-depth empirical study. In: **36th ICSME**. [S.l.: s.n.], 2020. p. 1–12. Cited 9 times in pages 14, 15, 19, 23, 28, 37, 39, 69, and 70.

7 BARBOSA, C. et al. Revealing the social aspects of design decay: A retrospective study of pull requests. In: **34th SBES**. [S.l.: s.n.], 2020. p. 1–10. Cited 15 times in pages 14, 15, 19, 21, 23, 28, 31, 37, 39, 40, 44, 48, 66, 70, and 71.

8 FOWLER, M. **Refactoring**. [S.l.]: Addison-Wesley Professional, 1999. Cited 5 times in pages 14, 15, 23, 30, and 38.

9 UCHÔA, A. et al. Predicting design impactful changes in modern code review: A large-scale empirical study. In: IEEE. **18th MSR**. [S.l.], 2021. p. 1–12. Cited 10 times in pages 14, 19, 21, 28, 29, 31, 39, 66, 69, and 70.

10 SOARES, V. et al. On the relation between complexity, explicitness, effectiveness of refactorings and non-functional concerns. In: **34th SBES**. [S.l.: s.n.], 2020. p. 788–797. Cited 4 times in pages 14, 21, 28, and 39.

11 KAMEI, Y. et al. A large-scale empirical study of just-in-time quality assurance. **IEEE Trans. Softw. Eng. (TSE)**, IEEE, v. 39, n. 6, p. 757–773, 2012. Cited 8 times in pages 15, 29, 31, 37, 40, 48, 66, and 69.

12 FAN, Y. et al. Early prediction of merged code changes to prioritize reviewing tasks. **Emp. Softw. Eng. (ESE)**, Springer, v. 23, n. 6, p. 3346–3393, 2018. Cited 5 times in pages 15, 29, 37, 40, and 48.

13 BIEMAN, J. M.; ANDREWS, A. A.; YANG, H. J. Understanding change-proneness in oo software through visualization. In: **11th IWPC**. [S.l.: s.n.], 2003. p. 44–53. Cited 6 times in pages 15, 29, 37, 40, 48, and 69.

14 BAVOTA, G. et al. An experimental investigation on the innate relationship between quality and refactoring. **J. Syst. Softw. (JSS)**, Elsevier, v. 107, p. 1–14, 2015. Cited 2 times in pages 15 and 36.

15 MACIA, I. et al. On the relevance of code anomalies for identifying architecture degradation symptoms. In: **16th CSMR**. [S.l.: s.n.], 2012. p. 277–286. Cited 4 times in pages 15, 29, 40, and 48.

16 FERNANDES, E. et al. On the alternatives for composing batch refactoring. In: IEEE. **3rd IWoR**. [S.l.], 2019. p. 9–12. Cited 2 times in pages 15 and 29.

17 BETTENBURG, N.; HASSAN, A. E. Studying the impact of social interactions on software quality. **Emp. Softw. Eng. (ESE)**, Springer, v. 18, n. 2, p. 375–431, 2013. Cited 4 times in pages 15, 31, 40, and 48.

18 CAPILUPPI, A.; AJIENKA, N.; COUNSELL, S. The effect of multiple developers on structural attributes: A study based on java software. **J. Syst. Softw. (JSS)**, Elsevier, p. 110593, 2020. Cited 8 times in pages 15, 19, 32, 37, 40, 48, 70, and 71.

19 BIRD, C. et al. Don't touch my code! examining the effects of ownership on software quality. In: **19th FSE**. [S.l.: s.n.], 2011. p. 4–14. Cited 5 times in pages 15, 32, 40, 48, and 69.

20 TUFANO, M. et al. When and why your code starts to smell bad (and whether the smells go away). **IEEE Transactions on Software Engineering**, IEEE, v. 43, n. 11, p. 1063–1088, 2017. Cited in page 17.

21 CHÁVEZ, A. et al. How does refactoring affect internal quality attributes? a multi-project study. In: **31st SBES**. [S.l.: s.n.], 2017. p. 74–83. Cited 7 times in pages 19, 23, 45, 49, 67, 69, and 70.

22 MARTINS, J. et al. Are code smell co-occurrences harmful to internal quality attributes? a mixed-method study. In: **34th SBES**. [S.l.: s.n.], 2020. p. 1–10. Cited 3 times in pages 19, 69, and 70.

23 OHLSSON, M. C. et al. Code decay analysis of legacy software through successive releases. In: **1999 IEEE Aerospace Conference. Proceedings**. [S.l.: s.n.], 1999. v. 5, p. 69–81. Cited 2 times in pages 19 and 37.

24 OIZUMI, W. et al. On the density and diversity of degradation symptoms in refactored classes: A multi-case study. In: **30th ISSRE**. [S.l.: s.n.], 2019. Cited in page 19.

25 PECORELLI, F. et al. Developer-driven code smell prioritization. In: **17th MSR**. [S.l.: s.n.], 2020. Cited in page 19.

26 HOZANO, M. et al. Are you smelling it? investigating how similar developers detect code smells. **Information and Software Technology**, Elsevier, v. 93, p. 130–146, 2018. Cited in page 19.

27 HAN, J.; PEI, J.; KAMBER, M. **Data mining: concepts and techniques**. [S.l.]: Elsevier, 2011. Cited 4 times in pages 20, 32, 35, and 41.

28 AGRAWAL, R.; IMIELINSKI, T.; SWAMI, A. Mining associations between sets of items in large databases. In: **ACM SIGMOD ICMD**. [S.l.: s.n.], 1993. p. 207–216. Cited 5 times in pages 20, 32, 33, 35, and 41.

29 SOARES, D. M. et al. Rejection factors of pull requests filed by core team developers in software projects with high acceptance rates. In: **14th ICMLA**. [S.l.: s.n.], 2015. p. 960–965. Cited 3 times in pages 21, 57, and 71.

30 BIBIANO, A. C. et al. How does incomplete composite refactoring affect internal quality attributes. In: **28th ICPC**. [S.l.: s.n.], 2020. Cited 6 times in pages 21, 26, 30, 45, 67, and 69.

31 BIBIANO, A. C. et al. Look ahead! revealing complete composite refactorings and their smelliness effects. In: **37th ICSME**. [S.l.: s.n.], 2021. Cited 2 times in pages 21 and 30.

32 TANG, A. et al. What makes software design effective? **Design Studies**, Elsevier, v. 31, n. 6, p. 614–640, 2010. Cited in page 23.

33 TAYLOR, R. N.; HOEK, A. Van der. Software design and architecture the once and future focus of software engineering. In: **Future of Software Engineering (FOSE)**. [S.l.: s.n.], 2007. p. 226–243. Cited in page 23.

34 MARTIN, R. C.; MARTIN, M. **Agile principles, patterns, and practices in C# (Robert C. Martin)**. [S.l.]: Prentice Hall PTR, 2006. Cited in page 23.

35 ALOMAR, E. A. et al. On the impact of refactoring on the relationship between quality attributes and design metrics. In: **19th ESEM**. [S.l.: s.n.], 2019. p. 1–11. Cited 6 times in pages 23, 24, 26, 39, 45, and 49.

36 CHIDAMBER, S. R.; KEMERER, C. F. A metrics suite for object oriented design. **IEEE Trans. Softw. Eng. (TSE)**, IEEE, v. 20, n. 6, p. 476–493, 1994. Cited 7 times in pages 24, 25, 26, 38, 39, 45, and 69.

37 HENRY, S.; KAFURA, D. Software structure metrics based on information flow. **IEEE Trans. Softw. Eng. (TSE)**, IEEE, n. 5, p. 510–518, 1981. Cited 3 times in pages 24, 45, and 69.

38 DAGPINAR, M.; JAHNKE, J. H. Predicting maintainability with object-oriented metrics-an empirical comparison. In: **10th WCRE**. [S.l.: s.n.], 2003. p. 155–164. Cited 3 times in pages 24, 25, and 45.

39 BIEMAN, J. M.; KANG, B.-K. Cohesion and reuse in an object-oriented system. **ACM SIGSOFT Software Engineering Notes**, ACM New York, NY, USA, v. 20, n. SI, p. 259–262, 1995. Cited 3 times in pages 24, 25, and 45.

40 MCCABE, T. J. A complexity measure. **IEEE Trans. Softw. Eng. (TSE)**, IEEE, n. 4, p. 308–320, 1976. Cited 3 times in pages 25, 45, and 69.

41 LORENZ, M.; KIDD, J. **Object-oriented software metrics: a practical guide**. [S.l.]: Prentice-Hall, Inc., 1994. Cited 3 times in pages 25, 45, and 69.

42 TOOLWORKS, I. S. **Understand Tool**. 2021. Available at: <https://scitools.com/>. Cited 3 times in pages 25, 27, and 46.

43 SILLITTI, A. et al. Agile processes in software engineering and extreme programming. In: **11th XP Int'l Conf.** [S.l.: s.n.], 2010. p. 1–4. Cited 2 times in pages 26 and 45.

44 DALLAL, J. A. Object-oriented class maintainability prediction using internal quality attributes. **Information and Software Technology**, Elsevier, v. 55, n. 11, p. 2028–2048, 2013. Cited in page 26.

45 FERNANDES, E. et al. Refactoring effect on internal quality attributes: What haven't they told you yet? **Information and Software Technology**, Elsevier, v. 126, p. 106347, 2020. Cited 2 times in pages 26 and 45.

46 DABBISH, L. et al. Social coding in github: transparency and collaboration in an open software repository. In: **15th CSCW**. [S.l.: s.n.], 2012. p. 1277–1286. Cited in page 28.

47 GIUFFRIDA, R.; DITTRICH, Y. Empirical studies on the use of social software in global software development–a systematic mapping study. **Inf. Softw. Technol. (IST)**, Elsevier, v. 55, n. 7, p. 1143–1164, 2013. Cited in page 28.

48 STOREY, M.-A. et al. How social and communication channels shape and challenge a participatory culture in software development. **IEEE Trans. Softw. Eng. (TSE)**, IEEE, v. 43, n. 2, p. 185–204, 2016. Cited 2 times in pages 28 and 39.

49 ZANETTI, M. S. et al. Categorizing bugs with social networks: a case study on four open source software communities. In: IEEE. **2013 35th International Conference on Software Engineering (ICSE)**. [S.l.], 2013. p. 1032–1041. Cited in page 28.

50 PALOMBA, F. et al. Beyond technical aspects: How do community smells influence the intensity of code smells? **IEEE transactions on software engineering**, IEEE, 2018. Cited in page 28.

51 RAY, B. et al. The uniqueness of changes: Characteristics and applications. In: IEEE. **2015 IEEE/ACM 12th Working Conference on Mining Software Repositories**. [S.l.], 2015. p. 34–44. Cited in page 29.

52 BIBIANO, A. C. et al. A quantitative study on characteristics and effect of batch refactoring on code smells. In: IEEE. **2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)**. [S.l.], 2019. p. 1–11. Cited 2 times in pages 30 and 45.

53 CEDRIM, D. et al. 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**. [S.l.: s.n.], 2017. p. 465–475. Cited 2 times in pages 30 and 45.

54 AGRAWAL, R.; SRIKANT, R. et al. Fast algorithms for mining association rules. In: **20th VLDB**. [S.l.: s.n.], 1994. v. 1215, p. 487–499. Cited 3 times in pages 32, 33, and 41.

55 KAUR, M.; KANG, S. Market basket analysis: Identify the changing trends of market data using association rule mining. **Procedia computer science**, Elsevier, v. 85, n. Cms, p. 78–85, 2016. Cited in page 32.

56 JIN, Z.; CUI, Y.; YAN, Z. Survey of intrusion detection methods based on data mining algorithms. In: **ICBDE**. [S.l.: s.n.], 2019. p. 98–106. Cited in page 32.

57 ALZU'BI, S. et al. A novel recommender system based on apriori algorithm for requirements engineering. In: **15th SNAMS**. [S.l.: s.n.], 2018. p. 323–327. Cited in page 32.

58 MUSE, B. A. et al. On the prevalence, impact, and evolution of sql code smells in data-intensive systems. In: **17th MSR**. [S.l.: s.n.], 2020. p. 327–338. Cited in page 32.

59 WESSEL, M. S. et al. Tweaking association rules to optimize software change recommendations. In: **31st SBES**. [S.l.: s.n.], 2017. p. 94–103. Cited in page 32.

60 TUFFÉRY, S. **Data mining and statistics for decision making**. [S.l.]: John Wiley & Sons, 2011. Cited in page 33.

61 FEITELSON, D. G.; FRACHTENBERG, E.; BECK, K. L. Development and deployment at facebook. **IEEE Internet Computing**, IEEE, v. 17, n. 4, p. 8–17, 2013. Cited in page 36.

62 TSAY, J.; DABBISH, L.; HERBSLEB, J. Let's talk about it: evaluating contributions through discussion in github. In: **22nd FSE**. [S.l.: s.n.], 2014. p. 144–154. Cited in page 36.

63 LE, D. M. et al. Relating architectural decay and sustainability of software systems. In: **13th WICSA**. [S.l.: s.n.], 2016. Cited in page 36.

64 AHMED, I. et al. An empirical study of design degradation: How software projects get worse over time. In: **9th ESEM**. [S.l.: s.n.], 2015. p. 1–10. Cited in page 37.

65 PIATETSKY-SHAPIRO, G. Discovery, analysis, and presentation of strong rules. **Knowledge discovery in databases**, Menlo Park, CA: AAI/MIT, p. 229–238, 1991. Cited in page 37.

66 MALHOTRA, R. **Empirical research in software engineering: concepts, analysis, and applications**. [S.l.]: CRC Press, 2016. Cited in page 38.

67 WOHLIN, C. et al. **Experimentation in Software Engineering**. 1st. ed. [S.l.]: Springer Science & Business Media, 2012. Cited 2 times in pages 42 and 69.

68 TSAY, J.; DABBISH, L.; HERBSLEB, J. Influence of social and technical factors for evaluating contribution in github. In: **36th ICSE**. [S.l.: s.n.], 2014. p. 356–366. Cited in page 44.

69 OIZUMI, W. et al. On the identification of design problems in stinky code: experiences and tool support. **JBCS**, Springer, v. 24, n. 1, 2018. Cited in page 45.

70 REPLICATION Package. 2021. <https://github.com/opus-research/decay_factors_replication>. Cited 4 times in pages 47, 54, 66, and 76.

71 SPADINI, D.; ANICHE, M.; BACCHELLI, A. PyDriller: Python framework for mining software repositories. In: **26th ESEC/FSE**. [S.l.: s.n.], 2018. p. 908–911. Cited in page 47.

72 Tsantalis, N.; Ketkar, A.; Dig, D. Refactoringminer 2.0. **IEEE Trans. Softw. Eng. (TSE)**, p. 1–1, 2020. Cited in page 47.

73 MARINESCU, R. Detection strategies: Metrics-based rules for detecting design flaws. In: IEEE. **20th IEEE International Conference on Software Maintenance, 2004. Proceedings.** [S.l.], 2004. p. 350–359. Cited in page 49.

74 MARTINS, J. S. **Investigando o Impacto das Coocorrências de Code Smells nos Atributos Internos de Qualidade**. Dissertação (Mestrado) — Universidade Federal do Ceará, set. 2021. Cited 2 times in pages 49 and 54.

75 WHITLEY, E.; BALL, J. Statistics review 6: Nonparametric methods. **Critical care**, Springer, v. 6, n. 6, p. 509, 2002. Cited in page 50.

76 MCDONALD, J. H. **Handbook of biological statistics**. [S.l.]: sparky house publishing Baltimore, MD, 2009. v. 2. Cited in page 50.

77 GRISSOM, R. J.; KIM, J. J. **Effect sizes for research: A broad practical approach.** [S.l.]: Lawrence Erlbaum Associates Publishers, 2005. Cited in page 50.

78 ROMANO, J. et al. Exploring methods for evaluating group differences on the nsse and other surveys: Are the t-test and cohen'sd indices the most appropriate choices. In: **Annual Meeting of the Southern Association for Institutional Research**. [S.l.: s.n.], 2006. Cited 2 times in pages 50 and 54.

79 MCGRAW, K. O.; WONG, S. P. A common language effect size statistic. **Psychological bulletin**, American Psychological Association, v. 111, n. 2, p. 361, 1992. Cited in page 50.

80 PALOMBA, F. et al. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. **Emp. Softw. Eng. (ESE)**, Springer, v. 23, n. 3, p. 1188–1221, 2018. Cited 2 times in pages 56 and 66.

81 HAHSLER, M.; CHELLUBOINA, S. Visualizing association rules: Introduction to the r-extension package arulesviz. **R**, p. 223–238, 2011. Cited in page 57.

82 PAIXÃO, M. et al. Behind the intents: An in-depth empirical study on software refactoring in modern code review. In: **Proceedings of the 17th International Conference on Mining Software Repositories**. [S.l.: s.n.], 2020. p. 125–136. Cited in page 67.

83 MONDAL, M.; ROY, C. K.; SCHNEIDER, K. A. Insight into a method co-change pattern to identify highly coupled methods: An empirical study. In: **21st ICPC**. [S.l.: s.n.], 2013. p. 103–112. Cited in page 71.

84 ZIMMERMANN, T. et al. Mining version histories to guide software changes. **IEEE Trans. Softw. Eng. (TSE)**, IEEE, v. 31, n. 6, p. 429–445, 2005. Cited 2 times in pages 71 and 72.