Anderson Gonçalves Uchôa

Unveiling Social and Technical Facets of Design Degradation in Modern Code Review

Tese de Doutorado

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

Advisor: Prof. Alessandro Fabricio Garcia

Rio de Janeiro
October 2021

Anderson Gonçalves Uchôa

Unveiling Social and Technical Facets of Design Degradation in Modern Code Review

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

Rio de Janeiro, October 4th, 2021

Anderson Gonçalves Uchôa

Substitute Professor in the Institute of Computing (IC) at the Federal University of Rio de Janeiro (UFRJ) since 2021. He obtained his PhD in Informatics from the Pontifical Catholic University of Rio de Janeiro (PUC-Rio) in 2021. He also obtained his Master's degree in Informatics from PUC-Rio in 2019, and a Bachelor's degree in Software Engineering from Federal University of Ceará (UFC-Campus Quixadá), Brazil, in cooperation with the Sapienza University of Rome, Italy in 2016. Anderson has worked for Research & Development (R&D) projects in software engineering and information systems for healthcare, in collaboration with institutions from Brazil and the United Kingdom (UK). His collaborative work resulted in more than 25 peer-reviewed papers. Anderson collaborates with various researchers in Brazil (more recently, UFPR, UFPE, and UFC) and UK (Newcastle University). His main research interests are software gamification, code review, software reuse, software evolution, software maintenance, and human aspects.

Bibliographic data

Uchôa, Anderson Gonçalves

Unveiling Social and Technical Facets of Design Degradation in Modern Code Review / Anderson Gonçalves Uchôa; advisor: Alessandro Fabricio Garcia. – 2021.

120 f: il. color. ; 30 cm

Tese (doutorado) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática, 2021.

Inclui bibliografia

1. Informática – Teses. 2. Degradação do Design. 3. Revisão de Código Moderna. 4. Práticas de Revisão de Código. 5. Aspectos Influentes. 6. Aprendizado de Máquina.. I. Garcia, Alessandro. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

Acknowledgments

First and foremost I want to thank God for being by my side at all times. My courage to move on came from my faith in Him.

I would like to express my gratitude to my family, who have supported me since the beginning of this long journey. Thanks to my father, Antonilo Uchôa Pereira, who taught me to always be the best version of myself and for always accompanying me in my decisions as hard as they were.

Thanks to my mother, Evalene Maria Gonçalves Uchôa, for her unconditional love and support over the years. Thanks to my brothers, Pâmela Gonçalves Uchôa and Alesson Gonçalves Uchôa, for being always available when I needed it, despite arguments between brothers.

My sincerest gratitude to my advisor Alessandro Garcia. I have no words to describe my admiration and gratitude. Alessandro has contributed significantly to my professional growth by providing the means for me to exceed my limits. I know how long and winding was the journey, but I remained strong and steady despite the obstacles. Thank you very much.

During these three years (without counting the two years of the masters), I had the opportunity to work with amazing researchers in Brazil. For this reason, I must thank Carla Bezerra, Wesley Assunção, Silvia Vergilio, Matheus Paixão, Thelma Colanzi, Juliana Pereira, Rafael de Mello, and Baldoino Fonseca. I also want to thank the members of my thesis defense team: Marcos Kalinowski, Simone Barbosa, Carla Bezerra, Rafael de Mello, Gleison Souza, and Alberto Raposo. I also thank all the professors from PUC-Rio for their invaluable contribution to my education during the doctoral program.

I would like to thank all of the new and old colleagues from the OPUS Research Group and from the Software Engineering Laboratory: Amadeu Neto, Anderson Oliveira, Anderson Santos, Ana Carla Bibiano, Alan Andrade, Alexander Lopez, Anne Benedicte, Bruna Moraes, Caio Barbosa, Cezar Filho, José Talavera, Chrystinne Fernandes, Daniel Tenorio, Danilo Felipe, Daniel Coutinho, Vinicius Soares, Dieinison Braga, João Victor, Raul Araujo, André Brandão, Gustavo Alexandre, Francisco Cunha, Hugo Guarín, Marx Viana, Nathalia Nascimento, Rafael de Mello, Rodrigo Laigner, and Willian Oizumi. Sorry if I forgot someone's name, but be aware that everyone contributed directly or indirectly to the conclusion of this cycle.

I am also grateful to the CNPq Foundation and PUC-Rio for financial support at different stages of this doctoral thesis. This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Finance Code 001. Finally, my sincere thanks to the administrative staff of the Department of Informatics at PUC-Rio, especially Cosme Leal for the support over these years.

Abstract

Uchôa, Anderson Gonçalves; Garcia, Alessandro (Advisor). **Unveiling Social and Technical Facets of Design Degradation in Modern Code Review**. Rio de Janeiro, 2021. 120p. Tese de Doutorado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Software design is a key concern in code review through which developers actively discuss and improve each software change. Nevertheless, code review is a collaborative task influenced by technical and social aspects. Consequently, these aspects can play a key role in how software design degrades. They can contribute to accelerating or reversing design degradation during the process of each single change's review. However, there is little understanding about: (i) the impact of code review and their practices on design degradation over time; and (ii) to what extent social and technical aspects are related to the reduction or increase of design degradation. We addressed these limitations driven by two goals. Our first goal is to provide a characterization of how modern code reviews impact design degradation during software maintenance. We consider various technical and socials aspects to study the code reviews' impact. Our second goal is to explore the role of technical and social aspects in distinguishing and predicting (un)impactful design changes during code reviews, using machine learning techniques. A design change is considered impactful when it varies the density and diversity of degradation symptoms (i.e., code smells). These goals were addressed with two empirical studies. Our first study reports a characterization of the impact of code reviews on the evolution of degradation symptoms along each code review. We also took into consideration when there was an explicit goal or discussions around software design. Our second study reports an analysis of technical and social aspects influencing changes along all the revisions within a single code review. Then, we could observe the role of social aspects in distinguishing and predicting design impactful changes. Our results show that the majority of code reviews have little or no impact on degradation, even with explicit design discussions. Long discussions and a high rate of reviewers' disagreement increase the risk of degradation. Both social and technical aspects are able to distinguish design (un)impactful changes. In summary, our results provided us with a better understanding of influential aspects that help us in deriving guidelines to mitigate degradation during code reviews. Our results also provide insights to design a new code review tools also able to warn developers early about the harmful design impact along code reviews.

Keywords

Design Degradation; Modern Code Review; Code Review Practices; Influential Aspects; Machine Learning.

Resumo

Uchôa, Anderson Gonçalves; Garcia, Alessandro. **Revelando as Facetas Sociais e Técnicas da Degradação do Design na Revisão de Código Moderna**. Rio de Janeiro, 2021. 120p. Tese de Doutorado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

O design de software é uma preocupação fundamental na revisão de código, por meio da qual os desenvolvedores discutem ativamente e melhoram cada mudança de software. No entanto, a revisão de código é uma tarefa colaborativa influenciada por aspectos técnicos e sociais. Consequentemente, esses aspectos podem desempenhar um papel fundamental em como o design de software se degrada. Eles podem contribuir para acelerar ou reverter a degradação do design durante o processo de revisão de cada mudança. No entanto, há pouco entendimento sobre: (i) o impacto da revisão do código e suas práticas na degradação do design ao longo do tempo; e (ii) em que medida os aspectos sociais e técnicos estão relacionados com a redução ou aumento da degradação do design. Abordamos essas limitações motivadas por dois objetivos. Nosso primeiro objetivo é fornecer uma caracterização de como as revisões de código modernas afetam a degradação do design durante a manutenção do software. Consideramos vários aspectos técnicos e sociais para estudar o impacto das revisões de código, utilizando técnicas de aprendizado de máquina. Nosso segundo objetivo é explorar o papel dos aspectos técnicos e sociais em distinguir e predizer mudanças de design (não) impactantes durante as revisões de código. Uma mudança de design é considerada impactante quando varia a densidade e a diversidade dos sintomas de degradação (ou seja, cheiros de código). Esses objetivos foram abordados em dois estudos empíricos. Nosso primeiro estudo relata uma caracterização do impacto das revisões de código na evolução dos sintomas de degradação ao longo de cada revisão de código. Também levamos em consideração quando havia um objetivo explícito ou discussões sobre design de software. Nosso segundo estudo relata uma análise dos aspectos técnicos e sociais que influenciam as mudanças ao longo de todas as revisões em uma única revisão de código. Então, pudemos observar o papel dos aspectos sociais em distinguir e predizer mudanças impactantes no design. Nossos resultados mostram que a maioria das revisões de código tem pouco ou nenhum impacto na degradação, mesmo com discussões explícitas de design. Longas discussões e uma alta taxa de discordância dos revisores aumentam o risco de degradação. Os aspectos sociais e técnicos são capazes de distinguir mudanças de design (não) impactantes. Em resumo, nossos resultados nos forneceram uma melhor compreensão dos aspectos influentes que nos ajudam a derivar diretrizes para mitigar a degradação durante as revisões de código. Nossos resultados também fornecem *insights* para projetar novas ferramentas de revisão de código, também capazes de alertar os desenvolvedores com antecedência sobre o impacto prejudicial do design ao longo das revisões de código.

Palavras-chave

Degradação do Design; Revisão de Código Moderna; Práticas de Revisão de Código; Aspectos Influentes; Aprendizado de Máquina.

Table of contents

List of figures

List of tables

List of Abreviations

ML – Machine Learning	
RQ – Research Question	
SVM – Support Vector Machines	
CERN – The European Organization for Nuclear Research	
FG – Fine-grained Smells	
CG – Coarse-grained Smells	
CROP – Code Review Open Platform	
DC – Diff Complexity	
DS – Diff Size	
PS – Patch Size	
PC – Patch Complexity	
NR – Number of Revisions	
DL – Discussion Length	
PRWF – Proportion of Revisions without Feedback	
CDCR – Churn during Review	
NR – Number of Reviewers	
NA – Number of Authors	
NNAV – Number of Non-Author Voters	
PRD – Proportion of Review Disagreement	
RL – Review Length	
RD – Response Delay	
ARR – Average Review Rate	
TRW – Typical Review Window	
DD – Design Discussion	
Inv – Invariant	

Pos – Positive

Neg – Negative

Mix – Mixed

NLA – Number of inserted lines in this code change

NLD – Number of deleted lines in this code change

CHURN – Number of lines added to and removed in this code change

NFA – Number of added files in this code change

NFD – Number of deleted files in this code change

NCF – Number of changed files in this code change

NMD – Number of modified directories in this code change

ME – Distribution of modified code across files in this code change

NLANG – Number of programming languages used in this code change

NFT – Number of file types in this code change

NSA – Number of added code segments in this code change

NSD – Number of deleted code segments in this code change

NSU – Number of updated code segments in this code change

FM – Number of times files in this code change were modified before

FD – Number of developers who changed files in this code change

ML – Number of words in description of this code change

BUG – Whether description of this code change contains word "bug"

FEAT – Whether description of this code change contains word "feature"

IMPR – Whether description of this code change contains word "improve"

DOC – Whether description of this code change contains word "document"

REFC – Whether description of this code change contains word "refactor

NC – Number of prior code changes submitted by the owner of this code change

NRC – NC in recent 120 days

NDC – NC that contain at least one directory affected by this code change

NR – Number of prior code changes the owner of this code change is assigned to inspect

MR – Merged rate of prior code changes submitted by the owner of this code change

RMR – MR in recent 120 days and normalized over the recent change number

DMR – MR that contain at least one directory affected by this code change

NIC – Number of inline comments made by reviewers

NWIC – Sum of the all words of each inline comment

PWIC – NWIC weighted by the number of inline comments

NGC – Number of general comments made by reviewers

NWGC – Sum of the all words of each general comments

PWGC – NWGC weighted by the number of general comments

DL – Number of general comments and inline comments written by reviewers

SD – The degree centrality for a node v is the fraction of nodes it is connected to

SCLOS – The inverse of the sum of all distances to all other nodes

SB – The sum of the fraction of all-pairs shortest paths that pass through

SE – The centrality for a node based on the centrality of its neighbors

SCLUST – The geometric average of the subgraph edge weights

SKC – Maximal subgraph that contains nodes of degree k or more

Pr – Precision

Re – Recall

F1 – F1-score

AUC – Area Under the ROC Curve values

RFECV – Recursive Feature Elimination with Cross-Validation

O sonho é que leva a gente para frente. Se a gente for seguir a razão, fica aquietado, acomodado.

Ariano Suassuna, *Pensador.*

1 Introduction

An earlier, shortened version of this chapter appears in
the Proceedings of the 28th ACM Joint European Software Engineering
Conference and Symposium on the Foundations of Software Engineering (FSE) [158].

Modern code review is a lightweight, informal, asynchronous, and tool-assisted technique aimed at detecting and removing issues introduced during software development [14]. Both industrial [16] and open-source [17] projects have been adopting modern code reviews on a daily basis as a way to improve the quality of their systems [14]. However, modern code reviews can be affected by both technical and social aspects. For instance, the technical aspects are those that characterize the source code, the file modifications, and a textual description of the change. Social aspects characterize the developer's experience, collaboration network, and participation in discussions during a code review.

A key concern of all code review stakeholders, including code owners and reviewers, is to remain aware of ongoing changes impacting the design [90, 21, 70]. In fact, previous studies [40, 159, 160] have observed that during code reviews, the involved stakeholders identify changes that impact the design in a bottom-up way, i.e., they start by identifying poor code structures – also known as code smells – at the low level and not by the top-down way, i.e., from high-level to low-level design. This approach is mainly because reviews are context-sensitive as reviewers are more aware of the details of the design decisions reviewed in the source code.

Although developers' awareness about the impact of design is an important point, many types of social and technical aspects can influence – either alone or simultaneously – how the design degradation can occur. Moreover, such aspects can influence to which extent the degradation can be slowed down or accelerated. For instance, the quality of each code change might be influenced by the developer(s) working on it and the change process itself [21, 15, 60, 90].

Software design degrades whenever one or more symptoms of poor structures end up being introduced by a change. In this context, if not properly avoided, identified, and combated, design degradation has severe consequences to software maintenance and the project continuation in the future [18, 19, 20, 34, 46, 91]. An example of a degradation symptom is when a class is overloaded with multiple unrelated functionalities, making it difficult to use. It also increases the chances of causing ripple change effects on other classes.

Despite its importance, we know little on how to design degradation evolves over time, both across and within reviews. In fact, there is a relation between design degradation and code review [15, 21, 24, 40, 73]. However, such relation was not deeply scrutinized in practice. Thus, we are likely to be biasing the research and practice of modern code review by ignoring the analysis of code review practices and the impact of modification across and within reviews. Since code reviews also aim to improve design quality, one could expect that, over time, those reviews would gradually reduce multiple degradation symptoms.

Prior studies reveal the positive influence of certain technical and social aspects in software engineering [93, 95, 100, 101]. Moreover, we lack evidence on to what extent these aspects can influence – either alone or simultaneously – the design degradation during code reviews. Hence, a better understanding of these aspects might help us derive guidelines for avoiding or mitigating degradation during code reviews. This understanding can also help to improve the current code review practices and develop a new generation of tools for assisting developers on becoming early becoming aware about the design impact during code reviews.

In this context, this Doctoral research focuses on *understanding how technical and social aspects can be explored to better support design-driven code reviews and the review process itself.* In other words, we aim to explore the social and technical facets of software design degradation in code reviews. Thus, this doctoral research aims to answer: *How to better support code review stakeholders in combating design degradation of their software systems?*. To answer this question, we aim to employ quantitative methods, including mining-based investigation.

1.1 Motivating Example

This section demonstrates a real scenario in which degradation symptoms were introduced during the code review process. We select the review 53,827 [55] from the jgit system to motivate this Doctoral Research and exemplify the phenomenon we investigate. The goal of the review task was to "delete non empty directories before checkout a path". The task was performed by one developer and reviewed by two others. As shown in Figure 1.1, through the course of this particular review, different aspects of the code change have been discussed. The main focus of the discussions was related to the functional requirements. Reviewers were concerned, for instance, with possible side effects that the functional change could introduce. Besides that, was there also a great concern about evolving the automated tests accordingly.

Only after 16 revisions in the review, there was a comment about structural design. The reviewer complained about the use of a boolean parameter in the method *checkoutEntry* from the *DirCacheCheckout* class. However, the author disagreed with the reviewer's comment, arguing that there would be no problem with the use of the boolean parameter. After that, the reviewer said that he would not insist, implying that he continues to disagree with the design decision being discussed. After that, no other comments regarding the structural design were made by the reviewers. As a possible consequence of this disagreement between those involved, many symptoms of potential degradation were ignored by the reviewers and prevailed in the system.

For example, we observed several occurrences of a fine-grained smell called *Magic Number*. This type of smell occurs when a literal number, which represents a specific meaning is used in the code by the programmer. The use of a literal number in code structures – such as if statements and assignments – are not advisable because it does not make explicit what the number really means. Instead, the recommended practice is to use constants or enumerations that make the meaning of numbers explicit. Instead of using the recommended

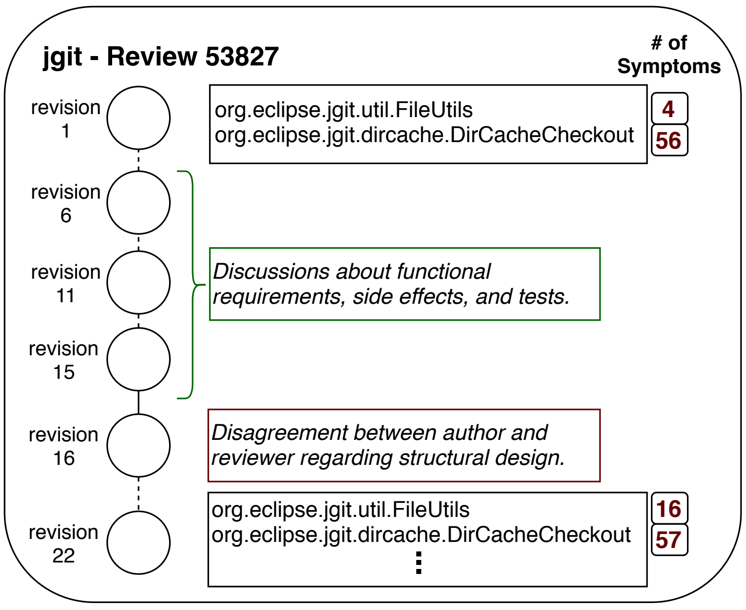
jgit - Review 53827
# of Symptoms
revision 1
org.eclipse.jgit.util.FileUtils
org.eclipse.jgit.dircache.DirCacheCheckout
4
56
revision 6
revision 11
Discussions about functional requirements, side effects, and tests.
revision 15
revision 16
Disagreement between author and reviewer regarding structural design.
revision 22
org.eclipse.jgit.util.FileUtils
org.eclipse.jgit.dircache.DirCacheCheckout
16
57

Figure 1.1: Example of a Code Review that Introduced Degradation Symptoms in the jgit System

practice, the author chose to comment on the code with an explanation of the meaning of the numbers involved.

Although the use of comments is a valid approach, it could be combined with the use of constants or enumerations to obtain a higher quality design and to prevent the same number, and its respective explanatory comment, from having to be repeated in several parts of the code. This is a problem that could be identified and removed during code review. As the repetitive use of the same number grows in the code, it also makes the risk higher in reducing program comprehension and inducing bugs. However, due to the great concern with several other aspects, such as functional requirements, non-functional requirements, and tests, the developers did not pay enough attention to the structural quality of the code.

In addition, the developers do not see the possible degradation in the changed code. As a consequence, new degradation symptoms emerged throughout the review. For instance, new occurrences of *Long Statement*, *Complex Method*, *Empty Catch Clause*, and *Magic Number* were introduced in methods of the *FileUtils* class. Moreover, other classes that were changed also presented more symptoms of possible degradation. Thus, it would be important to effectively assess the design of changed classes during code review as design problems may arise or become more severe during the review process. For instance, the co-occurrence of various *Complex Methods*, in the same classes is usually a sign of critical architectural problems, such as *Component Overload* or *Scattered Functionality* [29, 34].

1.2 Problem Statement and Limitations of Related Work

This section presents our general and specific research problems. Moreover, we discuss why related work does not address such research problems. In the previous section, we discussed and illustrated how degradation symptoms may be neglected, introduced, or removed throughout a code review in practice. The example illustrated how certain code review practices, such as disagreement among reviewers, might be a sign of the introduction or prevalence of degradation symptoms during code reviews.

On the impact of modern code review and their practices on design degradation. There are multiple studies about the perspective of developers about design degradation [28, 34, 37, 69, 91]. Other studies focus on the use of diversity and density of symptoms (i.e., code smells) as characteristics of design degradation [29, 67, 68]. For instance, Sousa *et al.* [34] identified five categories of symptoms upon which developers often rely to identify design problems. Similar to other studies [37, 40, 91], they observed that developers tend to combine multiple symptoms, by considering dimensions such as their density, diversity, and granularity to decide if there is design degradation. Oizumi *et al.* [29] investigated if degradation symptoms appear with higher density and diversity in classes refactored by developers. The authors observed that despite not being always removed by refactorings, some types of symptoms may be indeed strong indicators of design problems.

Complementary, Ahmed *et al.* [67] analyzed how open source projects get worse in terms of design degradation. The authors identified strong evidence that the density of design problems builds up over time. Finally, Mannan *et al.* [68] compared the occurrence of degradation symptoms in Android and desktop systems in terms of their variety, density, and distribution.

Other studies have investigated the impact of modern code review on design quality [11, 15, 21, 24, 40]. For instance, Morales *et al.* [24] studied the impact of both the code review coverage (proportion of change code reviewed) and the reviewer involvement in the prevalence or removal of smells. They observed that high coverage and review participation can reduce the occurrence of smells. Later, Mcintosh *et al.* [11] suggested that coverage, reviewer's participation, and expertise play high impacts on bug introduction. Concerning the severity of smells, Pascarella *et al.* [40] observed that active and participative code reviews have a significant influence on the reduction of code smell severity. And about the design of the systems, Zanaty *et al.* [21] observed that design-related discussion during code review is still rare. Finally, Paixão *et al.* [15] studied the code review impact on the high-level design. They observed that only 31% of the reviews with design discussions have a noticeable impact on the structural high-level design.

Despite the existing knowledge on design degradation and the impact of modern code review on design quality, the majority of these studies are limited in scope. A result is that there is little understanding of the impact of modern code review on reducing design degradation over time. More specifically, even though studies have been investigating the impact of modern code review on design quality [11, 15, 21, 24, 40], most of them only address the relation between modern code review – and their practices – and design degradation in a constrained manner. Such studies tend to analyze design degradation considering only single events (related to design degradation), such as the introduction of a single design problem [34, 37], or simply analyzing the degradation frequency [28, 29, 41, 40].

In addition, such studies have not assessed how the *process of design* *degradation evolution* is impacted: (i) *within each single review*, and (ii) *across multiple reviews*. Consequently, one cannot understand how certain *code review practices*, related to review intensity, developer participation, and the review duration, consistently reduce or further increase degradation as the project evolves. Hence, it remains unclear if and to what extent code review helps to combat design degradation. Moreover, there is little knowledge about the impact of developers' design discussions on degradation. Additionally, we do not know which practices may strengthen the combat or the acceleration of degradation. Therefore, aiming at deriving this knowledge, our first research problem is states as follows.

Research Problem 1. There is little knowledge about the impact of modern code review and – their practices – on design degradation over time.

On the role of social aspects on design impactful changes in modern code reviews. Software design is a key concern in code review through which developers actively discuss and improve each code change. Nevertheless, code review is predominantly a cooperative task influenced by both technical and social aspects. In fact, there are multiple studies about the effect of technical and social aspects in code reviews [73, 80, 81, 82, 85, 89]. For instance, aspects related to relationships among developers have been investigated by the following studies. Huang *et al.* [80] studied issues related to conflicts among developers in code review. They indicate that conflicts generally have side effects on developers' participation, while constructive suggestions increase retaining the developers. Later, Bosu *et al.* [85] studied the impact of interpersonal relations on the patch review. They found that the code author is one of the important factors for reviewers to decide to review a patch or not.

On the other hand, the following studies explore different facets related to participation. Ruangwan *et al.* [81] have investigated how many reviewers did not respond to a review invitation. The authors found that the more reviewers were invited to a patch, the more likely it was to receive a response. These results show the importance of the active engagement of reviewers. In a complementary way, Thongtanunam *et al.* [82] investigated patches that do not attract reviewers, patches that are not discussed, and those receiving slow initial feedback. They found that the length of the patch description plays an important role in the likelihood of receiving poor reviewer participation or discussion. Other studies have investigated aspects of communication and confusion. Barbosa *et al.* [73] observed that communication dynamics among developers and discussion content can contribute to combating or amplifying design degradation. Ebert *et al.* [89] found that missing rationale and lack of familiarity with the code are the major reasons for confusion in code review.

Despite the prior studies revealing the influence of certain technical and social aspects in code reviews, none of them investigated to what extent these aspects can influence – either alone or simultaneously – the design degradation during code reviews. For instance, the quality of each code change might be influenced by the experience of developers working on it and the change process itself [15, 21, 60, 90]. Consequently, these aspects can play a key role in how software design degrades. They can also contribute to accelerating or reversing the degradation during the process of each single code change's review.

The social and technical aspects are often captured through the extraction of different kinds of metrics. For instance, the *number of lines* that were *added* and *deleted* in a specific file helps us to measure the intensity of modification in such a file. In this context, in code review platforms, stakeholders have either technical or social information at their disposal to be used as additional information, both before or after each change. Social information includes the developers' experience, their collaboration network, and participation in discussions during a code review [7, 85, 97, 99]. Technical information includes aspects related to the source code, files, and the description of a change [49, 94, 96, 99]. Moreover, this information is available after changes and revisions are done during the review.

Despite the large, diverse, rich information from social and technical aspects in code review platforms, the contribution of using technical and social metrics is often observed to characterize and predict failures [93, 95, 100, 101]. However, no study has done so in the context of design degradation. In fact, their use to discriminate and predict design impactful changes has not been investigated in depth so far [73, 90]. In this context, such metrics can act as indicators of design impactfulness of ongoing changes along the code review process [73]. In fact, the early identification of impactful changes that degrade the design is important during code review [33, 74, 136].

In other words, if these harmful changes are not reversed early, i.e., before a code review is ended, rework will be necessary after the changes of the last merged revision. Further changes with time-consuming refactorings will have to be applied later. Given the costs of design refactorings, they are unlikely to be applied and code smells will be increasingly compounded over time, thereby accelerating the design degradation [15, 136].

However, there is no empirical evidence about which technical and social aspects captured as metrics can distinguish the impactfulness of design changes during code reviews. Moreover, there is little knowledge of whether the use of these aspects represented as features can be used as a proxy to predict design impactful changes. Therefore, we also do not know if the use of supervised machine learning techniques can assist developers to automatically determine whether a code change is impactful or not. Hence, it remains unclear which metrics, used as features, are the best predictor, as well as the effectiveness of combining social and technical features for predicting design (un-)impactful changes. Therefore, aiming at deriving this knowledge, our second research problem is states as follows.

Research Problem 2. There is a lack of empirical evidence on the role of social aspects in distinguishing and predicting (un)impactful design changes.

By studying these two specific research problems, we expect to understand how technical and social aspects can be positively explored to better support code reviewers in their work and facilitate design-driven reviews. In summary, the investigation of these specific problems discussed heretofore will contribute to better understand how to support design-driven code reviews and how technical and social aspects can be positively explored to better support the code review process.

1.3 Goal and Research Questions

The research problems aforementioned are essential to provide the knowledge required to understand (i) the impact of modern code review and their practices on design degradation, (ii) the role that social aspects play in distinguishing and predicting design impactful changes, and, finally, (iii) how to provide the support to design intelligent tools that will aid code review stakeholders to avoid design degradation in practice. Given this context, the goal of this thesis is stated as follows.

Goal. Understand which contextual information – technical and social – can be addressed to support the design-related reviews.

To achieve this goal, we mapped each research problem into two research questions (RQ$_s$) as follows. Therefore, Research Problems 1 and 2 were mapped onto two research questions, respectively.

RQ$_1$: *To what extent does modern code review impact the design degradation evolution?* – **RQ$_1$** aims at providing evidence on the impact of code reviews on the evolution of design degradation. To this end, we conducted a mining-based investigation and reported the first study that characterizes how the process of design degradation evolves within each review and across multiple reviews. Moreover, we analyzed a comprehensive suite of metrics to enable us to observe the influence of certain code review practices on combating or even accelerating design degradation.

$\mathbf{RQ}_2$: *To what extent social aspects contribute to distinguishing and predicting design impactful changes in modern code review?* – **$\mathbf{RQ}_2$** aims at investigating if code review stakeholders could benefit from approaches to distinguish and predict design impactful changes with technical and/or social aspects. To this end, we reported an investigation on the prediction of design impactful changes in modern code review. We extracted and assessed 41 different metrics based on both social and technical aspects of the changes involved in each revision of a code review. Based on different features set, we evaluated the use of interpretable Machine Learning (ML) algorithms to predict design impactful changes. Finally, we evaluated the predictive power of the selected metrics and algorithms to assist developers to determine whether a code change is impactful.

1.4 Thesis Contributions

This doctoral thesis expands the current knowledge on design degradation in modern code review during software evolution. Particularly, we provide new insights on *the social and technical facets of design degradation in modern code review.* We rely on both two large-scale quantitative studies (published) based on mining software repositories. We summarize the main contributions of this doctoral thesis as follows.

1.4.1 Empirical Characterization on How Modern Code Review Impact Software Design Degradation

As aforementioned, though studies have been investigating the impact of modern code review on design quality [11, 15, 21, 24, 40], we still know little about the impact of modern code review and – their practices – on design degradation overtime **(Research Problem 1)**. More critically, it remains unclear if and to what extent code review helps to combat design degradation. It also remains unclear how the practices related to three code review factors, namely *Review Intensity*, *Review Participation*, and *Reviewing Time*, usually have a positive, neutral or negative effect on degradation.

For this purpose, we conducted an in-depth empirical study [90] in which we retrospectively investigate 14,971 code reviews from seven systems of two large open source communities. We performed three major steps. First, we characterized the impact of code reviews on the evolution of design degradation. To this end, we explored how two degradation characteristics: density and diversity of symptoms, evolved over time. We analyzed such characteristics in the context of two major categories of degradation symptoms, which are the fine-grained and coarse-grained smells [30]. Finally, we analyzed the impact on design degradation caused by two code review factors. The first factor was the presence of explicit intent of improving the design. The second one was the presence of explicit design discussions along with the revisions of a review.

Second, we investigated how degradation symptoms evolve along with the revisions that occur along each review. To this end, we identified and investigated four different evolution patterns for degradation characteristics (i.e., density and diversity). Such investigation provided us with new insights on the design decays or improvements along the reviewing process. Finally, we explored the relationship of different code review practices with the evolution of degradation characteristics. By exploring this relationship, we evidenced that certain code review practices can be used as indicators of increasing design degradation. Additionally, we reveal if according to prior studies [8, 11, 40, 50] code reviews that are intensely scrutinized, with more team participation and lasted for a long time, usually have a positive effect on design.

As a result, we observed that: (1) when developers have an explicit concern with design along a code review, the effect on design degradation is usually positive or invariant. However, the sole presence of design discussions is not a decisive factor to avoid degradation; (2) during the revisions of each single review, there is often a wide fluctuation of design degradation. This fluctuation means that developers are both introducing and removing symptoms along a single code review. However, at the end of the review, such fluctuations often result in the amplification of design degradation, even in the context of reviews with an explicit design concern; and (3) certain code review practices increase the risk of design degradation, including long discussions and a high rate of reviewers' disagreement. The finding on long discussions shows that those discussions are introducing more than removing degradation symptoms.

Contribution 1. We report the first study that characterizes how the process of design degradation evolves within each review and across multiple reviews. Moreover, we analyze a comprehensive suite of metrics to enable us to observe the influence of certain code review practices on combating or even accelerating design degradation.

1.4.2 Exploring Social Aspects for Distinguishing and Predicting Design Impactful Changes

As aforementioned, despite prior studies revealing the positive influence of certain technical and social aspects in software engineering [93, 95, 100, 101], their use to discriminate and predict design impactful changes is rarely studied [73, 90]. As a consequence, we lack evidence on to what extent these aspects can influence – either alone or simultaneously – the design degradation during code reviews **(Research Problem 2)**. This understanding can also help to improve the current code review practices and develop a new generation of tools for assisting developers on becoming early becoming aware about the design impact during code reviews.

Aimed to investigate the use of social aspects to discriminate and predict design impactful changes within modern code reviews, we conducted a large-scale empirical study [70] in which we analyzed more than 50k code reviews of seven real-world systems. We performed three major steps. First, we mined a comprehensive set of 41 features able to capture both technical and social aspects of the changes involved in each revision of a code review. Next, we evaluated which metrics can distinguish between design impactful changes and unimpactful ones. Second, we assessed the use of supervised ML techniques to aid developers in automatically make their decisions.

Thus, we compare the performance of six interpretable ML algorithms: *Logistic Regression*, *Naive Bayes*, *SVM*, *Decision Tree*, *Random Forest*, and *Gradient Boosting*. We chose these algorithms as they provide a more straightforward way to explain the prediction classification output [78, 79]. After, for each ML algorithms we explored how effective are the social and technical features as a proxy to the design (un)impactful changes. To this end, we evaluated and compared the performance of both kinds of features. Consequently, we applied the ML algorithm using three feature sets: a set using only social features, a set using only technical ones, and a set using technical and social features together. Finally, we explored what features are the best indicators of impactful design changes across ML models by considering the different feature sets.

As a result, we observed that: (1) both social and technical metrics are able to distinguish design (un)impactful changes; (2) features related to the code change, commit message, and file history are effective for differentiating (un)impactful changes; (3) *Random Forest* and *Gradient Boosting* have shown to be the most accurate in predicting design impactful changes; (4) the use of technical features results in more accurate predictions when compared to the social ones. Moreover, such kinds of features can be used in combination with social features without reducing the performance of the ML algorithms; and (5) the technical features tend to be considered the best indicators across models when compared with social features. However, social features that quantify the developer's experience are also considered important across models.

Contribution 2. We reported the effect of certain technical and social aspects on design degradation during code reviews. We empirically explored and assessed the role of social aspects in distinguishing and predicting design impactful changes in code review.

1.5 Research Publications

The empirical studies of this Doctoral thesis were reported in papers already published. The first part of Table 1.1 lists all papers derived or strongly related from this thesis in Rows 1 to 5. The second part of Table 1.1 lists, in the remaining lines, some papers produced along with my doctoral research in cooperation with other colleagues. In the Chapters 3 and 4 we presents the studies, [90] and [70], respectively.

1.6 Study Replicability and Open Science

An essential and good practice of any scientific research is that studies must be replicable, i.e., every manuscript must give detailed information on how the study can be *repeated* or *replicated*. In this context, the open science movement [157] aims to make all research artifacts available to the public, thus, increasing *transparency* and *reproducibility* of the scientific process. To this end, for each study that composes this doctoral thesis, we make the replication package available on Zenodo (`https://zenodo.org/`), an online repository hosted at CERN which allows sharing publications and supporting data for replication. Table 1.2 lists where each replication package is hosted to promote and open science.

We emphasize that for each replication package, we make available all data collected, together with the definition of metrics, features, and statistical tests, and scripts.

Table 1.1: List of Research Publications

Paper	Ref.	Chap.
Unveiling Multiple Facets of Design Degradation in Modern Code Review. **Uchôa, A.** In Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2021), 2021, Athens, Greece, August 23–28, pages 1615–1619, 2021.	[158]	1
How Does Modern Code Review Impact Software Design Degradation? An In-depth Empirical Study. **Uchôa, A.**; Barbosa, C.; Oizumi, W.; Blenilio, P.; Lima, R.; Garcia, A.; and Bezerra, C. In Proceedings of the 36th International Conference on Software Maintenance and Evolution (ICSME), 2020, Adelaide, Australia, Sept 27-Oct 3, pages 511–522, 2020.	[90]	3
Predicting Design Impactful Changes in Modern Code Review: A Large-Scale Empirical Study. **Uchôa, A.**; Barbosa, C.; Coutinho, D.; Oizumi, W.; K. G. Assunção, W.; Regina Vergilio, S.; Alves Pereira, J.; Oliveira, A.; and Garcia, A. In Proceedings of the 18th International Conference on Mining Software Repositories (MSR 2021), 2021, Madrid, Spain, May 17–19, pages 1 – 12, 2021.	[70]	4
Behind the Intents: An In-depth Empirical Study on Software Refactoring in Modern Code Review. Paixao, M.; **Uchôa, A.**; Bibiano, A. C.; Oliveira, D.; Garcia, A.; Krinke, J.; and Arnovio, E. In Proceedings of the 17th International Conference on Mining Software Repositories (MSR), 2020, Seoul, South Korea, May, pages 1 – 11, 2020. ACM	[60]	N/A
Revealing the Social Aspects of Design Decay: A Retrospective Study of Pull Requests. Barbosa, C.; **Uchôa, A.**; Falcao, F.; Coutinho, D.; Brito, H.; Amaral, G.; Garcia, A.; Fonseca, B.; Ribeiro, M.; Soares, V.; and Sousa, L. In Proceedings of the 34th Brazilian Symposium on Software Engineering (SBES), 2020, Natal, Brazil, Oct 19-23, pages 364–373, 2020. ACM Press	[73]	N/A
Do Critical Components Smell Bad? An Empirical Study with Component-based Software Product Lines. **Uchôa, A.**, Assunçao, W. KG, and Garcia, A. In Proceedings of the 15th Brazilian Symposium on Software Components, Architectures, and Reuse (SBCARS), 2021, Joinville, Brazil **(Distinguished Paper Award – 2nd Place)**	[166]	N/A
How do Code Smell Co-occurrences Removal Impact Internal Quality Attributes? A Developers' Perspective. Martins, J.; Bezerra, C.; **Uchôa, A.**; and Garcia, A. In Proceedings of the 35th Brazilian Symposium on Software Engineering (SBES), 2021, Joinville, Brazil	[165]	N/A
Visualizing the Maintainability of Feature Models in SPLs. Lima, L.; **Uchôa, A.**; Bezerra, C.; Coutinho, E.; and Rocha, L. In Proceedings of the 8th Workshop on Software Visualization, Evolution and Maintenance (VEM 2020), 2020, Natal, Brazil, Oct 19, pages 1 – 8, 2020.	[164]	N/A
On the Relation between Complexity, Explicitness, Effectiveness of Refactorings and Non-Functional Concerns. Soares, V.; Oliveira, A.; Farah, P.; Bibiano, A.; Coutinho, D.; Garcia, A.; Vergilio, S.; Schots, M.; Oliveira, D.; and **Uchôa, A**. In Proceedings of the 34th Brazilian Symposium on Software Engineering (SBES), 2020, Natal, Brazil, Oct 19-23, pages 788–797, 2020. ACM Press	[163]	N/A
Are Code Smell Co-occurrences Harmful to Internal Quality Attributes? A Mixed-Method Study. Martins, J.; **Uchôa, A.**; Bezerra, C.; and Garcia, A. In Proceedings of the 34th Brazilian Symposium on Software Engineering (SBES), 2020, Natal, Brazil, Oct 19-23, pages 52–61, 2020. ACM Press	[69]	N/A
REM4DSPL: A Requirements Engineering Method for Dynamic Software Product Lines. Sousa, A.; **Uchôa, A.**; Fernandes, E.; Bezerra, C. I.; Monteiro, J. M.; and Andrade, R. In Proceedings of the XVIII Brazilian Symposium on Software Quality (SBQS), 2019, Fortaleza, Brazil, Oct 28-Nov 1, pages 129–138, 2019. ACM	[162]	N/A
Do Research and Practice of Code Smell Identification Walk Together? A Social Representations Analysis. de Mello, R.; **Uchôa, A.**; Oliveira, R.; Oizumi, W.; Souza, J.; Mendes, K.; Oliveira, D.; Fonseca, B.; and Garcia, A. In Proceedings of the 13th International Symposium on Empirical Software Engineering and Measurement (ESEM), 2019, Porto de Galinhas, Brazil, Sept 19-20., pages 1–6, 2019. IEEE Press	[74]	N/A

Table 1.2: Replication Package Available per Chapter

Replication Package	Chapter	Host
How Does Modern Code Review Impact Software Design Degradation? An In-depth Empirical Study	2	Zenodo [58]
Predicting Design Impactful Changes in Modern Code Review: A Large-Scale Empirical Study	3	Zenodo [59]

1.7 Thesis Outline

This introductory chapter portrayed an overview of this thesis. The remainder of the thesis is structured as follows. **Chapter 2** introduces basic concepts and related work aimed to support the understanding of this thesis. **Chapter 3** introduces our quantitative study focused on the characterization of how modern code review impacts software design degradation. We present and discuss the main study results as reported in our paper [90]. **Chapter 4** introduces our second quantitative study on the role of social aspects in distinguishing and predicting (un)impactful design changes. We also present and discuss the main study results as reported in our paper [70]. Finally, **Chapter 5** concludes this doctoral thesis by summarizing the achieved research contributions, implications, delimitations, making final considerations, and pointing out directions for future research.

2
Background and Related Work

This chapter presents the background and related work of this thesis. Section 2.1 outlines a brief history about how code reviews were applied in different forms over time. Section 2.2 discusses the modern code review process that is the focus of this doctoral thesis. Section 2.3 overviews software design degradation and its symptoms. Section 2.4. Finally, Section 2.6 presents related work that investigated code smells and other indicators of design degradation.

2.1
A Brief History of Code Reviews Evolution

The formal software inspection is a well-structured code review process defined by Fagan [167]. Software inspection requires a set of procedures, inducing synchronous discussion meetings, which are not necessarily executed in other forms of code review. By conducting a software inspection the developer can detect and remove defects before the software project is available. As a well-structured code review process, a software inspection consists of six steps: planning, overview, preparation, examination, rework, and follow-up [167].

In the **planning step**, an inspection team is formed. Basically, each team member has a specific role: author (the person who created the work product being inspected); moderator (the leader of the inspection, that plans the inspection and coordinates it); inspector (the ones who raise questions, suggest problems, and criticize the document), and recorder (the person that documents the defects that are found during the inspection). In the **overview step**, the author describes the contexts of the software artifact that will be inspected [167].

The following steps are those in which the review of the artifacts actually begins. In the **preparation step**, each inspector examines the code artifact to identify possible defects individually. Subsequently, in the **examination step**, the reader scrutinizes the artifact, line-by-line, and points out the defects in every part. Next, in the **rework step**, the author makes changes to the code artifacts according to the action plan detailed in the examination step. The changes by the author are checked to make sure everything is correct. Finally, in the **follow-up step**, the moderator verifies the fixes that were produced during the rework step.

Despite the formal software inspection been quite successful in industry and academia, mainly in the context of critical systems [173]. However, its formality brings several disadvantages [168, 169]. For instance, the inspection process as a whole is often very time consuming, since the inspection team needs to be organized and prepared to be part of various meetings [168]. Another disadvantage is that the required formalism of the inspection activity is not aligned with agile development methods [169].

Due to these and other disadvantages, new lightweight approaches were introduced. For instance, the *walkthroughs* approach are informal code reviews where an owner sets up a meeting and invites teammates to critique software artifacts. The focus of the meeting is to find and resolve problems in the artifacts. Another lightweight approach is the *email-based code review* in which the code review starts when an owner broadcasts a review request, for a code patch, through the project's mailing lists. Then, developers who are interested in the patch provide feedback by replying to the email [5, 7, 170].

In the next section, we introduce a lightweight tool-assisted code review process, nowadays called *modern code review*. In this thesis, we explored and investigated data obtained from this kind of code review. We focus on modern code review, as it has been widely used in open and closed projects, from different domains and in large and small organizations.

2.2 Modern Code Review

Nowadays, major companies, such as Facebook [26] and Microsoft [14], utilize a more lightweight code review process on a daily basis. This modern code review process is defined by Bacchelli & Bird [14] as an informal (in contrast with the process defined by Fagan [167]) tool-assisted, and asynchronous review technique, aimed at detecting and removing issues that were introduced during development tasks. This process can be seen in Figure 2.1 and it is described as follows.

Supported by tools, such as Gerrit [1] and GitHub, this modern code review process is initiated by one developer referred to as the *code author* (or *code owner*) that *(1) modifies the original codebase* and *(2) submits a new code change* to be reviewed. These code changes are reviewed by other developers, i.e., *code reviewers*, that will read and analyze them [66].

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

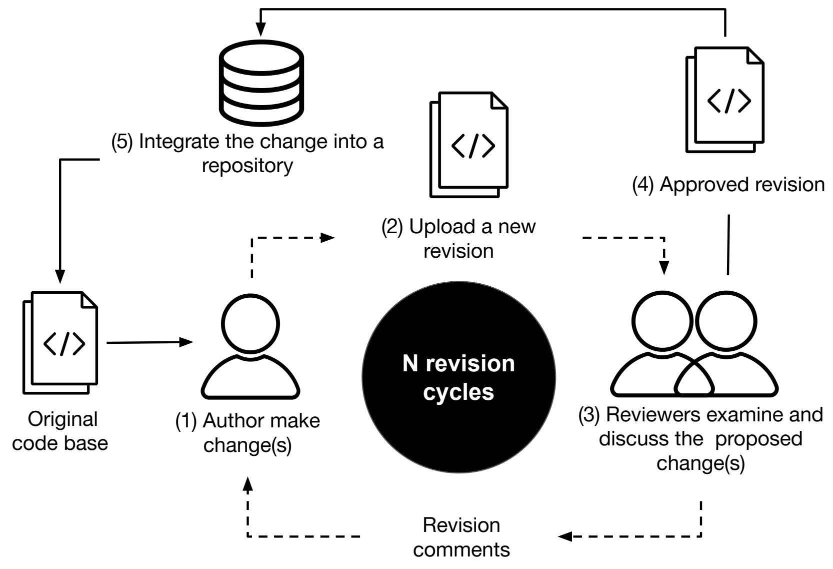
(5) Integrate the change into a repository
(4) Approved revision
(2) Upload a new revision
Original code base
(1) Author make change(s)
N revision cycles
(3) Reviewers examine and discuss the proposed change(s)
Revision comments

Figure 2.1: Overview of Modern Code Review Process

The code reviewers *(3) examine the code change* to detect issues, such as bugs, design problems, and style violations [13, 25]. After that, the code reviewers *(3) provide their review feedback*, in the form of code review comments, to the code owner. In turn, the code owner applies the requested fixes and forwards a new version of the source code for analysis, which can be followed by another code review analysis and its resulting comments. This cycle is iterative and ends up when a decision is made about either the acceptance or rejection of the integration of the change into the codebase [13, 66].

Throughout this work, we use *review* to indicate the entire process of a single code review. The process starts to from the submission the a new code change for review and ends with the approval on or rejection of integration of the change into the codebase. In addition, we use *revision* to indicate each iteration of this process during a single review.

2.3 Software Design Degradation and Its Symptoms

The design of a software results from a set of decisions made by the developers along time [92, 102]. However, those decisions can (un)intentionally degrade this design due to the introduction of poor code structures. Those design problems can be seen as evidence of that degradation process, and are also named degradation symptoms [10, 34, 136].

In this thesis, we take into account two categories of design degradation symptoms: *Fine-grained Smells* (FG) and *Coarse-grained Smells* (CG) [30].

FG smells are indicators of structural degradation in the scope of methods and code blocks [30]. For instance, the Long Method is a FG smell that occurs in methods that contain too many lines of code. On the other hand, CG smells are symptoms that may indicate structural degradation related to object-oriented principles, e.g., abstraction, encapsulation, and modularity [30, 34]. An example of CG smell is Insufficient Modularization [30]. This symptom occurs in classes that are large and complex due to the accumulation of various responsibilities. Such categories encapsulate a set of symptoms that are more perceived and used by developers in practice to identify and refactor source code locations degraded [22, 34, 36, 37].

To summarize, a degradation symptom is a characteristic of the code that can indicate the existence of a deeper design problem. Moreover, design degradation can be observed from an increase in the values of the density or diversity of the design degradation symptoms. Throughout this work, we calculate the density of design degradation symptoms by counting the number of single degradation symptoms found in the source code. Conversely, diversity is calculated as the number of different types of smells that can be found in an instance of source code.

Table 2.1 lists the 27 symptoms types investigated in the thesis, where the CG smells and FG smells are presented in the upper and bottom halves of the table, respectively. In this thesis, all selected degradation symptoms help us to measure different poor code structures. Additionally, all symptoms were automatically detected using a state-of-the-practice tool called Designite-Java [4]. This tool is successfully used in recent studies on design degradation (e.g., [29, 73]), and prior work [114] indicated a precision of 96% and a recall of 99%.

2.4 Technical and Social Aspects

Software development does not only consist of technical activities but also of social activities that emerge from the collaboration among developers [187, 188, 189, 73]. Thus, social activities, such as developers communicating with each other, the developer's experience, and their collaboration networks are central to software development. In this case, the investigation of social aspects is crucial to understand current software practices, processes, and tools as well as their impact [189].

In this context, we known that modern code review is a collaborative task influenced by both technical and social aspects [48, 85]. Consequently, both social and technical aspects can play a key role in how software design

Table 2.1: Design Degradation Symptoms Investigated in this Thesis [30]

Symptom Types and Description
Imperative Abstraction: when an operation is turned into a class
Multifaceted Abstraction: an abstraction that has more than one responsibility assigned to it
Unutilized Abstraction: an abstraction that is left unused
Unnecessary Abstraction: an abstraction that is actually not needed in the system
Deficient Encapsulation: the accessibility of one or more members of an abstraction is more permissive than actually required
Unexploited Encapsulation: when a client class that uses explicit type checks instead of exploiting the variation in types already encapsulated within a hierarchy
Broken Modularization: when data and/or methods that should have been into a single abstraction are spread across multiple abstractions
Insufficient Modularization: when an abstraction that has not been completely decomposed
Hub Like Modularization: when an abstraction has dependencies (both incoming and outgoing) with a large number of other abstractions.
Cyclic Dependent Modularization: when two or more abstractions depend on each other directly or indirectly
Rebellious Hierarchy: when a subtype that rejects the methods provided by its supertype(s)
Wide Hierarchy: when an inheritance hierarchy that is too wide
Deep Hierarchy: when an inheritance hierarchy that is excessively deep
Multipath Hierarchy: when a subtype inherits both directly as well as indirectly from a supertype leading to unnecessary inheritance paths in the hierarchy.
Cyclic Hierarchy: when a supertype in a hierarchy that depends on any of its subtypes
Missing Hierarchy: when a design segment uses conditional logic instead of polymorphism.
Broken Hierarchy: a supertype and its subtype conceptually do not share an “is a“ relationship
Abstract Function Call From Constructor: a constructor that calls an abstract method
Complex Conditional: a conditional statement that is complex
Complex Method: a method that has high cyclomatic complexity
Empty Catch Block: a catch block of an exception that is empty
Long Identifier: an identifier that is excessively long
Long Method: a method that is too long to understand
Long Parameter List: a method that accepts a long list of parameters
Long Statement: a statement that is excessively long
Magic Number: when an unexplained number is used in an expression
Missing Default: a switch statement that does not contain a default case

degrades during the code review process. For example, the quality of each code change can be influenced by the developers working on it and by the change process itself. Thus, in this thesis, we consider aspects associated with the source code being developed, as *technical aspects* [6]. An example of a technical aspect is the complexity of the change that needs to be reviewed. Conversely, we consider a *social aspects* as dimensions that characterize properties related to communication among developers on the environment that they are developing code, the developer’s experience, and their collaboration networks [6, 186]. An example of a social aspect is the experience of developers, including the change author and reviewers.

In this thesis, we explore the different facets of social and technical aspects by computing metrics that can be directly related to different dimensions of each aspect. Such metrics are widely used in software engineering literature [5, 7, 32, 49, 82, 94, 96, 97, 99, 98, 111, 112, 136]. Tables 2.2 and 2.3 overview the different metrics that we use to measure the social and technical aspects, respectively. Each of these tables is structured as follows. The first column lists the different dimensions of each aspect. The second column names the metrics that we use to measure different properties of a dimension. The last two columns describe each metric and the rationale for using this metric, respectively.

Regarding the social aspects described in Table 2.2, we split them into three dimensions: (i) *developer's experience* comprises the metrics related to the previous experience of the code change owner [7, 98, 99]; (ii) *discussion activity* comprise the metrics of communication between developers and reviewers [32, 136]; and (iii) *collaboration networks* consists of metrics of social networks [97, 98, 111]. Regarding the technical aspects (Table 2.3), we split them into five dimensions: (i) *size* consists of metrics related to source in their smallest granularity [5]; (ii) *diffusion* comprehends the features about changes distributed on two or more files (e.g., number of changed files) [94, 99]; (iii) *complexity* comprehends the metrics on the complexity of a change [111]; (iv) *file history* is composed of metrics related to the history of the files. For instance, the number of prior changes to a file can be a good indicator to detect degraded files [49, 94, 96]; and (v) *textual* consists of features that capture textual characteristics of the commit messages [82, 112].

Table 2.2: Social Aspects Investigated in this Thesis Grouped into Different Dimensions and Metrics

Dimension	Metric name	Description	Rationale
refers to metrics which are based on the experience of a code change owner			
Developer's Experience	change num (NC)	Number of prior code changes submitted by the owner of this experience code change	Developer experience may be an essential piece of information for predicting software quality. Additionally, patch writer experience significantly impacts code review quality.
	recent change num (NRC)	Number of prior code changes submitted by the owner of this code change in recent 120 days	
	dir change num (NDC)	Number of prior code changes submitted by the owner of this code change, that contain at least one directory affected by this code change	
	review num (NR)	Number of prior code changes the owner of this code change is assigned to inspect	If a code change owner has reviewed many changes submitted by other developers, he/she would be more familiar with coding standards and operations of Gerrit system. Thus, the review num help to quantify for the owner's experience
	merged ratio (MR)	Merged rate of prior code changes submitted by the owner of this code change	Acceptance rate prior to current pull request as a feature to predict pull request outcomes (i.e., merged or abandoned)
	recent merged ratio (RMR)	Number of merged code changes submitted by the owner of this code change in recent 120 days prior to this code change and normalized over recent change num.	Measuring recent performance of the owner prior to current code change.
	dir merged ratio (DMR)	Merged rate of prior code changes submitted by the owner of this code change, that contain at least one directory affected by this code change	Quantifying the owner's familiarity with the directories affected by this code change.
refers to metrics that capture code review activities in the patches			
Discussion Activity	inline comments num (NIC)	Number of inline comments made by reviewers on the code change submitted by the owner	Classes having degraded symptoms can create more discussion among the reviewers on how to refactor them.
	words inline comments num (NWIC)	Sum of the all words of each inline comment. Here we applied the pre-processing in the text removing contractions, stop words, punctuation, and replacing numbers	Discussions with a high number of comments around a code change would find possible design symptoms, improving or maintaining the quality.
	percentage words in inline comments (PWIC)	Sum of the all words of each inline comment weighted by the number of inline comments. Here we applied the preprocessing in the text removing contractions, stop words, punctuation, and replacing number.	Discussions with a high weighted number of comments around a code change may lead to design degradation.
	general comments num (NGC)	Number of general comments made by reviewers on the code change submitted by the owner.	
	words general comments num (NWGC)	Sum of the all words of each general comment. Here we applied the pre-processing 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 degradation.
	percentage words in general comments (PWGC)	Sum of the all words of each general comment weighted by the number of general comments. Here we applied the preprocessing in the text removing contractions, stop words, punctuation, and replacing number.	Discussions with a high weighted number of words are related to more complex changes, that may lead to design degradation.
	discussion length (DL)	Number of general comments and inline comments written by reviewers	Reviewing proposed changes with a long discussion would find more degradation symptoms and provide a better design solution.
refers to metrics that are used to measure this code change owner's network degree of activity in collaboration process of the corresponding project prior to this code change			
Collaboration networks	degree centrality (SD)	The degree centrality for a node v is the fraction of nodes it is connected to	Collaboration factors of the code change owners (i.e., their level of participation within the project) can influence code review outcomes.
	closeness centrality (SCLOS)	The inverse of the sum of all distances to all other nodes	
	betweenness centrality (SB)	The sum of the fraction of all-pairs shortest paths that pass through v	
	eigenvector centrality (SE)	The centrality for a node based on the centrality of its neighbors	
	clustering coefficient (SCLUST)	The geometric average of the subgraph edge weights.	
	social k coreness (SKC)	Maximal subgraph that contains nodes of degree k or more.	

Table 2.3: Technical Aspects Investigated in this Thesis Grouped into Different Dimensions and Metrics

Dimension	Metric name	Description	Rationale
refers to metrics that are based on the source code in the revisions			
Size	lines added num (NLA)	Number of inserted lines in this code change	Large patches may need more effort to review
	lines deleted num (NLD)	Number of deleted lines in this code change	
	churm num (CHURN)	Number of lines added to and removed in this code change	
	file added num (NFA)	Number of added files in this code change	
	file deleted num (NFD)	Number of deleted files in this code change	
Diffusion	changed file num (NCF)	Number of changed files in this code change	Patches, where their changes scatter across a large number of files or directories, may need more effort to review. Finding reviewers who have knowledge of such changes is difficult as well. Therefore, it is more likely that the patch will suffer from poor reviewer involvement.
	directory num (NMD)	Number of modified directories in this code change	
	modify entropy (ME)	Distribution of modified code across files in this code change	
	language num (NLANG)	Number of programming languages used in this code change	
	file type num (NFT)	Number of file types in this code change	
Complexity	segs added num (NSA)	Number of added code segments in this code change	A code change with more code segments modified is likely more complex and requires reviewer more effort and time to comprehend it.
	segs deleted num (NSD)	Number of deleted code segments in this code change	
	segs updated num (NSU)	Number of updated code segments in this code change	
refers to metrics that are based on file modification history recorded by the Gerrit systems			
File history	changes files modified (FM)	Number of times files in this code change were modified before	The number of previous changes to a file is a good indicator to detect degraded files.
	file developer num (FD)	Number of developers who changed files in this code change	Files that are previously touched by more developers are more likely to introduce degradation symptoms.
refers to metrics that capture textual characteristics of the commit message			
Textual	msg length (ML)	Number of words in description of this code change	Description length of a patch is related to its likelihood of receiving poor comments.
	has bug (BUG)	Whether description of this code change contains word “bug” and more keywords based on previous work [1,2]	Commit message containing more information about a code change may help reviewers comprehend the change more easily.
	has feature (FEAT)	Whether description of this code change contains word “feature” [1,2]	
	has improve (IMPR)	Whether description of this code change contains word “improve” and more keywords based on previous work [1,2]	
	has document (DOC)	Whether description of this code change contains word “document” and more keywords based on previous work [1,2]	
	has refactor (REFC)	Whether description of this code change contains word “refactor” and more keywords based on previous work [3]	

2.5 Supervised Machine Learning Algorithms

Machine learning (ML) is a sub-area of artificial intelligence (AI) that encapsulate "a set of methods that allow computers to learn from the data to make and improve predictions" [177]. The machine learning area has several types of algorithms. These algorithms are typically classified into four types: *supervised learning*, *unsupervised learning*, *reinforcement learning*, and *evolutionary learning* [177]. The most common types are supervised learning and unsupervised learning [177]. The main difference between these two types is the existence of prior knowledge concerning the output values (the ground truth). Thus, in supervised learning, the main goal is to learn a specific task, given a sample of data and desired output. Conversely, in unsupervised learning, we do not have a labeled output, thus, the main goal is to infer the output without an already known sample of data and ground truth [177].

In the context of software engineering, unsupervised learning is usually used when it is not possible to label the instances to be classified. However, most problems in software engineering work with instances that can be labeled, so it becomes more feasible to use supervised learning (e.g., [49, 77, 93]). Thus, in this thesis, we used different supervised ML algorithms to explore the role of social aspects in predicting design impactful changes (see Chapter 4 for more details). We emphasize that these algorithms are all interpretable, which allows us to understand the rationale behind the classification. We describe the six different (binary classification) supervised ML algorithms that we have used as follows.

Logistic Regression [178]: Logistic Regression is a ML algorithm similar to linear regression. This ML algorithm is centered on combining input values using coefficient values (i.e., weights) to predict an outcome value. Differently from linear regression, the outcome value being modeled ranges from 0 to 1. **Naive Bayes** [179]: The Naive Bayes (Gaussian) describe a set of steps to apply Bayes' theorem to classification problems. These algorithms use training data to compute the probability of each outcome based on the information extracted from feature values. Its main idea describes the probability of an event based on prior knowledge of conditions that might be related to this event.

Support Vector Machines [180]: Support Vector Machines computes is an ML algorithm that searches for the best hyper-plane to separate the training instances into their respective classes. To make this classification, SVM creates classification models that are a representation of examples as points in space. These points are mapped in such a way that the examples in each category are divided by a clear space that is as broad as possible. Each new instance is mapped in the same space and predicted as belonging to a category based on which side of space they are placed.

Decision Trees [183]: Decision Tree algorithms yield hierarchical models composed of decision nodes and leaves. Essentially, the resulting models represent a partition of the feature space. Basically, from the root of the tree, the value of a certain independent variable is evaluated and it is decided whether the next node will be the one on the right or the left. This process is repeated until you reach a leaf node that indicates the class that will be given as the result.

Random Forest [181, 182]: Random Forest is an ensemble of decision tree predictors. In other words, such an algorithm uses many decision trees with random subsets of the training data. The Random Forest adds extra randomness to the model when during the tree's creation. Instead of looking for the best feature when partitioning nodes, it looks for the best feature in a random subset of features. This process creates great diversity, which generally leads to the generation of better models, besides that this diversity also reduces the overfitting effect.

Gradient Boosting [184, 129]: The Gradient Boosting is an ML algorithm that works in a forward stage-wise fashion; it allows for the optimization of arbitrary differentiable loss functions. In each stage, the regression trees are fit on the negative gradient of the binomial or multinomial deviance loss function. Binary classification is a special case where only a single regression tree is induced.

2.6 Related Work

We emphasize that the next sections are a cut of the related work described in Chapters 3 and 4. The related work described here is the same set of studies presented in Chapter 1, but grouped into three different viewpoints. Furthermore, the differences between our work in relation to the previous ones are described in Chapters 3 and 4. Section 2.6.1 presents studies that explore the perspective of developers about design degradation. Section 2.6.2 that have investigated the impact of code review on design quality. Finally, Section 2.6.3 presents studies about the effect of technical and social aspects in code reviews.

2.6.1 Empirical Studies on Design Degradation

There are multiple studies that analyze the perspective of developers about design degradation [28, 34, 37, 69, 91]. Others study the use of diversity and density of symptoms (i.e., smells) as characteristics of design degradation [29, 67, 68]. Sousa *et al.* [34] identified five categories of symptoms upon which developers often rely to identify design problems. Similar to other studies [37, 91], they observed that developers tend to combine multiple symptoms, by considering dimensions such as their density, diversity, and granularity to decide if there is design degradation. Oizumi *et al.* [29] investigated if degradation symptoms appear with higher density and diversity in classes refactored by developers. The authors observed that despite not being removed by refactorings, some types of symptoms may be indeed strong indicators of design problems. Ahmed *et al.* [67] analyzed how open source projects get worse in terms of design degradation. The authors identified strong evidence that the density of design problems builds up over time. Mannan *et al.* [68] have compared the occurrences of degradation symptoms in Android and desktop systems in terms of their variety, density, and distribution.

2.6.2 Studies on the Impact of Modern Code Review on Design Quality

Prior studies have investigated the impact of code review on design quality [11, 15, 21, 24, 40]. Morales *et al.* [24] studied the impact of code review coverage (proportion of changes code reviewed) and reviewer involvement on smells. They observed that high coverage and review participation can reduce the occurrence of smells. Later, Mcintosh *et al.* [11] suggested that coverage, reviewer's participation, and expertise play high impacts on bug introduction. Pascarella *et al.* [40] observed that active and participative code reviews have a significant influence on the reduction of code smell severity. Zanaty *et al.* [21] observed that design-related discussion during code review is still rare. Later, Paixão *et al.* [15] studied the code review impact on the structural high-level design. They observed that only 31% of the reviews with design discussions have a noticeable impact on the structural high-level design.

2.6.3 Studies on Technical and Social Aspects in Modern Code Review

There are various studies that analyze the effect of technical and social aspects in code reviews [73, 80, 81, 82, 85, 89]. Bosu *et al.* [85] studied the impact of interpersonal relations on the patch review. They found that the code author is one of the important factors for reviewers to decide to review a patch or not. In addition, Ruangwan *et al.* [81] investigated how many reviewers did not respond to a review invitation. The authors found that the more reviewers were invited to a patch, the more likely it was to receive a response. Thongtanunam *et al.* [82] investigated patches that do not attract reviewers, not discussed, and receive slow initial feedback. They found that the length of the patch description plays an important role in the likelihood of receiving poor reviewer participation or discussion. Ebert *et al.* [89] found that missing rationale and lack of familiarity with the code are the major reasons for confusion in code review. Recently, Barbosa *et al.* [73] observed that communication dynamics among developers and discussion content decay contribute to combating or amplifying design decay.

2.7 Summary of Chapter 2

This chapter introduced a number of key concepts and related work to support the understanding of this thesis. This doctoral research goes a step further than related work studies, by explicitly investigating the relationships among the aforementioned perspectives. More specifically, we aim at understanding the social and technical facets of design degradation in modern code review and how social and technical aspects can influence – either alone or simultaneously – how to design degradation occurs.

The next chapter provides the first investigation to better understanding the modern code review impact software design degradation by considering multiples factors. First, in Section 2.1 we introduced the code review process, providing a brief overview of its history and discussing how it is applied in modern software development. Second, in Section 2.3 we also introduce the design degradation process, and described how we can detect it by using some of its symptoms. Finally, in Section 2.6, we discuss previous works related to this thesis.

3
How Does Modern Code Review Impact Software Design Degradation? An In-depth Empirical Study

An earlier version of the work in this chapter appears in the Proceedings of the 36th International Conference on Software Maintenance and Evolution (ICSME) [90].

As mentioned in Section 1.2, previous studies [11, 15, 21, 24, 40] have investigated the impact of modern code reviews by addressing only the relation between modern code review – and their practices – and design degradation in a constrained manner. For instance, considering only single events (related to design degradation), such as the introduction of a single design problem [34, 37], or simply analyzing the degradation frequency [28, 29, 41, 40]. Thus, we conducted a study, by addressing the limitations of previous studies and advancing in knowledge about the impact of modern code reviews.

More specifically, in this chapter, the paper *How Does Modern Code Review Impact Software Design Degradation? An In-depth Empirical Study* will be presented in its entirely. The paper was published in the 36th International Conference on Software Maintenance and Evolution (ICSME) [90]. This study comprehends the first contribution of this Doctoral research: *empirical characterization on how modern code Review impact software design degradation* (see Section 1.4.1).

3.1 Introduction

Modern code review is a lightweight, informal, asynchronous, and tool-assisted technique aimed at detecting and removing issues that were introduced during development tasks [14]. Both industrial [16] and open-source [17] projects have been adopting modern code review on a daily basis as a means to promote the quality of their systems [14]. Along code reviews, developers inspect and discuss the quality of each other's code changes before accepting them.

Modern code review may play a key role at both improving the design quality of a software as well as its maintainability [11, 22, 21]. Prior studies [14, 24, 40] suggest that certain code review practices, such as the lack of review participation, may increase design degradation. Other studies have shown that, along code review, developers often argue about software maintainability and suggest design improvements to the code owners [13, 21, 23].

Code review may or may not be explicitly focused on design quality [15, 21, 60]. Unfortunately, even with explicit design discussions, changes performed by developers along reviews can lead to design degradation. Design degradation is the process where design decisions progressively worsen the structural quality of a system, thereby also hampering external quality attributes such as maintainability. If not properly avoided, identified and combated, design degradation has severe consequences to the software health and also possibly contributing to its (dis)continuation in the future [18, 19, 20, 36, 34, 46].

Existing studies tend to analyze design degradation considering only single events, such as the introduction of a single design problem [34, 37], or simply analyzing the degradation frequency [28, 29, 41, 40]. Nevertheless, understanding how the design degradation evolves over time – across reviews and within reviews – is of paramount importance. Otherwise, we are misinforming the research and practice of modern code review. Since the code review also aims to improve design quality, one could expect that, over time, the reviews will gradually reduce multiple degradation symptoms.

To the best of our knowledge, there is no study that performs an in-depth investigation of the impact of modern code review – and its practices – on the design degradation evolution. Hence, it remains unclear whether and to what extent code review helps to combat design degradation. Moreover, there is little knowledge about the impact of developers' design discussions on degradation. Additionally, we do not know which practices may strengthen the combat or the acceleration of design degradation.

This paper addresses the aforementioned limitations through an in-depth empirical study that characterizes the impact of modern code review and its practices on design degradation evolution. To this end, we retrospectively investigate 14,971 code reviews from seven software systems pertaining to two large open source communities. We analyze the characteristics of design degradation across reviews and within reviews. Moreover, we assess how reviews with design discussion tend to impact design degradation. Finally, we analyze a comprehensive suite of metrics to support our observations regarding the relationship between certain code review practices and the combat (or amplification) of design degradation.

Our contributions include: (i) findings on the characterization of the code review impact on design degradation, (ii) findings on how design degradation evolves along with code reviews, and (iii) statistical analyses concerning the relationship between certain code review practices and design degradation. We summarize our study findings as follows:

1. When developers have an explicit concern with design, the effect on design degradation is usually positive or invariant. However, the sole presence of design discussions is not a decisive factor to avoid degradation;

2. During the revisions of each single review, there is often a wide fluctuation of design degradation. This fluctuation means that developers are both introducing and removing symptoms along a single code review. However, at the end of the review, such fluctuations often result in the amplification of design degradation, even in the context of reviews with an explicit design concern;

3. Certain code review practices increase the risk of design degradation, including long discussions and a high rate of reviewers' disagreement. The finding on long discussions shows that long discussions are introducing more than removing degradation symptoms.

3.2 Background and Related Work

3.2.1 Design Degradation

Design degradation is a phenomenon in which developers progressively introduce design problems in a system [44]. The degradation is caused by design decisions that negatively impact quality attributes such as maintainability and extensibility [46, 34]. An example of degradation is when a class is overloaded with multiple unrelated functionalities, making it difficult to use and increasing the chances of causing ripple effects on other classes. Given the potential harmfulness of design degradation, developers need to identify and refactor source code locations impacted by design degradation.

Empirical studies on design degradation. There are multiple studies about design degradation [28, 29, 34, 36, 37, 39, 67, 68, 69]. Oizumi et al. [29], e.g., investigated if degradation symptoms appear with higher density and diversity in classes refactored by developers. The authors observed that despite not being removed by refactorings, some types of symptoms might be indeed strong indicators of design problems. Ahmed et al. [67] analyzed how open source projects get worse in terms of design degradation. The authors identified strong evidence that the density of design problems build up over time.

None of the aforementioned studies have analyzed how code changes performed by developers during code reviews impact on design degradation. In this work, we fill this gap in the literature by investigating two categories of symptoms, which are fine-grained (FG) and coarse-grained (CG) smells [30]. *FG smells* are indicators of structural degradation in the scope of methods and code blocks [30]. For instance, the Long Method is a FG smell that occurs in methods that contain too many lines of code. This smell usually indicates modifiability and comprehensibility problems. *CG smells* are symptoms that may indicate structural degradation related to object-oriented principles such as abstraction, encapsulation, modularity, and hierarchy [34, 30]. An example of CG smell is Insufficient Modularization [30]. This symptom occurs in classes that are large and complex due to the accumulation of responsibilities.

3.2.2 Modern Code review

Modern code review is typically a lightweight, informal, asynchronous, and tool-assisted practice aimed at detecting and removing issues that were introduced during development tasks [14]. Major companies, such as Facebook [26] and Microsoft [14] use modern code review on a daily basis. Supported by tools such as Gerrit, the modern code review process is initiated by one developer referred to as the *code owner* that *modifies the original codebase* and *submits a new code change* to be reviewed. These code changes are reviewed by other developers, i.e., *code reviewers*, that will inspect it [66]. The code reviewers *inspect the code change* to detect issues such as bugs, design problems, and violations of style [13, 25].

After that, the code reviewers *provide their review feedback*, in the form of code review comments, to the code owner. In turn, the code owner applies fixes and forwards the new version of the source code for inspection, which can be followed by another code review comment. This cycle is iterative and ends up with either the acceptance or rejection of the integration of the change into the codebase [13, 66].

In our study, we use *review* to indicate the entire process of a single code review, from submitting a new code change for review to approving or rejecting the integration of the change into the codebase. In addition, we use *revision* to indicate each iteration of this process during a single review.

Empirical studies on the impact of modern code review. Multiple studies have investigated the impact of code review on software quality [15, 21, 24, 32, 40]. Morales et al. [24] investigated the relation between code review and code smells. The authors observed that software components with limited review coverage and participation are more prone to the occurrence of code smells compared to components whose review process is more active. Pascarella et. al [40] investigated if and how code review influences the severity of six types of code smells in seven Java open-source projects. The authors observed that active and participative code reviews have a significant influence on the reduction of code smell severity. Another study [15] has investigated the impact of code review on the structural high-level design. The authors observed that only 31% of the reviews with design discussions have a noticeable impact on the structural high-level design. In this work, we investigate the same set of systems analyzed by them. Nevertheless, we focused on assessing the impact of modern code reviews on design degradation. In addition, we conducted multiple new analysis, as summarized below.

In a nutshell, our work differs from the existing ones in the following points: (1) we investigated how the occurrence of design discussions during a review may affect the evolution of design degradation; (2) while most studies are focused in analyzing the degradation as single events, we investigated the manifestation and evolution of the design degradation process (increase and reduction) under different aspects, which are across reviews and along with revisions of each review; and (3) we used a multiple logistic regression model to evaluate the impact of code review practices on design degradation.

3.3 Study Settings

Section 4.3.1 presents both goal and research questions. Finally, Section 3.3.2 shows the study steps and procedures.

3.3.1 Goal and Research Questions

In this study, our goal is to characterize the impact of code review practices on the evolution of design degradation. To achieve this goal, we analyzed code reviews that were extracted from seven large open-source systems. Our study was based on three research questions (RQs) as follows.

RQ$_1$: *To what extent do modern code reviews impact design degradation?* **RQ$_1$** aims at providing evidence on the impact of code reviews on the evolution of design degradation. To achieve this goal, we focus on exploring the evolution of two degradation characteristics: density and diversity of symptoms. We analyze such characteristics in the context of two categories of degradation symptoms, which are the fine-grained and coarse-grained smells. Finally, we analyze the impact on design degradation caused by two code review factors.

The first factor is the presence of explicit intent of improving the design. The second one is the presence of explicit design discussions along with the revisions of a review. We provide more details about such factors in Section 3.3.2.

RQ$_2$: *How does design degradation evolve along with each code review?* **RQ$_2$** aims at investigating how degradation characteristics evolve along with the revisions that occur along with each code review. To answer **RQ$_2$**, we identified and investigated four different evolution patterns for degradation characteristics (i.e., density and diversity). Such investigation provides us with new insights about the evolution of design degradation throughout the reviewing process.

RQ$_3$: *How do code review practices influence design degradation?* **RQ$_3$** aims at exploring in depth the relationship of different code review practices with the evolution of degradation characteristics. A correlation between these two variables may evidence that certain code review practices can be used as indicators of increased design degradation. Also, by answering **RQ$_3$**, we will be able to reveal whether according to previous studies [8, 11, 40, 50] code reviews that are intensely scrutinized, with more team participation, and reviewed for a longer time, usually has a positive effect on design degradation.

3.3.2 Study Steps and Procedures

We describe each study step and procedures as follows.

Step 1: Select software systems that adopt modern code review. We selected systems provided by the Code Review Open Platform (CROP) [3], an open-source dataset that links code review data with their respective code changes. CROP currently provides data for 11 systems, extracted from two large open source communities: Eclipse and Couchbase. All systems in CROP employ Gerrit as their code review tool. Hence, by using CROP, we have access to a rich dataset of source code changes that goes beyond other platforms, such as Github. We selected only Java systems included in the CROP dataset due to the limitations of the DesignateJava tool [4] (see Step 2). We considered only merged reviews, since they represent changes that were integrated into the systems. In addition, we discarded reviews that did not change Java files. Table 4.1 provides details about each selected system, where the Eclipse and Couchbase systems are presented in the upper and bottom half of the table, respectively. We also detail the number of merged reviews and revisions in each system, followed by the time-span of our investigation.

Step 2: Detect degradation symptoms during code review. We used the DesigniteJava tool [4] to detect a total of 27 degradation symptoms

Table 3.1: Software Systems Investigated in this Study

Systems	# of Reviews	# of Revisions	Time span
jgit	3,736	10,718	10/09 to 11/17
egit	3,607	9,937	9/09 to 11/17
platform.ui	3,072	10,282	20/13 to 11/17
linuxtools	2,947	9,149	6/12 to 11/17
java-client	642	2,064	11/11 to 11/17
jvm-core	629	1,851	4/14 to 11/17
spymemcached	338	1,010	5/10 to 7/17

types: 17 coarse-grained (CG) smells, and 10 fine-grained (FG) smells. Hence, for each system under study, we identified these degradation symptoms by considering each review and submitted revisions that have undergone the code review process. For each submitted revision, we used CROP to access the versions of the system before and after the revision took place. Hence, we guaranteed that the introduced degradation symptoms between each version were solely introduced by the code changes in the revisions. Table 4.2 lists the 27 symptoms types investigated in our study, where the CG and FG smells are presented in the upper and bottom half of the table, respectively. We provide all descriptions, detection strategies, and thresholds for each type of symptom in our replication package [58].

Table 3.2: Degradation Symptoms Investigated in this Study

Coarse-grained Smells
Imperative Abstraction, Multifaceted Abstraction, Unutilized Abstraction, Unnecessary Abstraction, Deficient Encapsulation, Unexploited Encapsulation, Broken Modularization, Insufficient Modularization, Hub Like Modularization, Cyclic Dependent Modularization, Rebellious Hierarchy, Wide Hierarchy, Deep Hierarchy, Multipath Hierarchy, Cyclic Hierarchy, Missing Hierarchy, Broken Hierarchy.
Fine-grained Smells
Abstract Function Call From Constructor, Complex Conditional, Complex Method, Empty Catch Block, Long Identifier, Long Method, Long Parameter List, Long Statement, Magic Number, Missing Default.

Step 3: Compute degradation characteristics for each symptom category during code review. We rely on an existing grounded theory [34] that explains that developers tend to consider multiple degradation characteristics. We take into account two characteristics, namely *density* and *diversity*, as metrics to measure the level of design degradation. For each selected system, we computed these characteristics in the context of each symptom category (CG and FG smells), for all the collected reviews and revisions. For each review and revision, we also used CROP to access the versions of the system before and after each review and revision. *Density* was computed for each version of the system, before and after each revision, as the sum of the number of symptom instances in the set of source code files. Similarly, we computed the *diversity* as a sum of the number of different symptom types in the set of source code files.

The computation of density and diversity before and after revisions, allowed us to generate four different indicators of design degradation for each revision, where each indicator represents the differences in density and diversity of FG and CG smells. In summary, when the degradation characteristic, either density or diversity, after the revision, is larger than the characteristic before the revision there is an *increase in the degradation* as a result of the revision. Similarly, when the degradation characteristic after the revision is smaller than the characteristic before, there is a *reduction of the degradation* as a result of the revision. In total, we have computed the four indicators for 14,971 code reviews and 45,011 revisions.

Step 4: Identify design degradation evolution patterns across reviews. We identified the design degradation evolution across reviews by adapting a recent state-of-the-art classification provided by a previous work [15]. To find evolution patterns, we considered only reviews, identified, and filtered in Step 3, which presented an increase or decrease of degradation. For this purpose, we considered reviews that: (i) have more than one revision, and (2) present symptoms of degradation. We firstly identified the last merged revision of each review, which represents the degradation evolution that was, in fact, incorporated into the system. After that, we compared the degradation characteristics of the last merged revision with all the other previous revisions of each code review. This procedure enabled us to investigate how design degradation evolves across the revisions of each review.

Step 5: Calculate code review activity metrics. Table 3.3 shows the 16 metrics that we used to measure the code review activity. The first part of Table 3.3 describes the control variables that we computed to avoid some factors that may affect our outcome if not adequately controlled. As control variables, we used *Product* and *Process* metrics, which have been shown by previous research to be correlated with design degradation [63, 64]. The second part of Table 3.3 describes the metrics that we considered as independent variables to measure the code review activity. We have grouped each metric in three dimensions. *Review Intensity* measures the scrutiny that was applied during the code review process. *Review Participation* measures how much the development team invests in the code review process. Finally, *Reviewing Time* measures the duration of a code review. We emphasize that these metrics are extensively used by previous work (e.g., [8, 11, 24]) to measure the code review activity. Moreover, all three dimensions investigated in our study suggest practices that may be favorable or not to combat design degradation.

Table 3.3: Independent and Control Variables Used in our Study. The Rationale of each Metric is Described in our Replication Package [58]

Type	**Metric**	**Description**
		Control variables
Product	DiffComplexity (DC)	The difference of the sum of the Weighted Method per Class metric computed on the version before and after review of all classes being subject of review.
	DiffSize (DS)	The difference of the sum of the Lines of Code metric computed on version before and after review of all classes being subject of review.
	Patch Size (PS)	Total number of files being subject of review.
Process	Patch Churn (PC)	Sum of the lines added and removed in all the classes being subject of review.
		Independent variables
Review Intensity	Number of Revisions (NR)	The number of revisions for a patch prior to its integration.
	Discussion Length (DL)	Number of general comments and inline comments written by reviewers.
	Proportion of Revisions without Feedback (PRWF)	The proportion of iterations without discussions started by a reviewer, neither posting a message nor a score.
	Churn during Review (CDCR)	Number of lines that were added and deleted between revisions.
Review Participation	Number of Reviewers (NR)	Number of developers who participate in a code review, i.e., posting a general comment, or inline comment, and assigning a review score.
	Number of Authors (NA)	Number of developers who upload a revision for proposed changes. Changes revised by many authors may introduce more degradation into the system [49, 51].
	Number of Non-Author Voters (NNAV)	Number of developers who assign a review score, excluding the patch author.
	Proportion of Review Disagreement (PRD)	A proportion of reviewers that vote for a disagreement to accept the patch, i.e., assigning a negative review score.
Reviewing Time	Review Length (RL)	Time in days from the first patch submission to the reviewers' acceptance for integration [6, 42].
	Response Delay (RD)	Time in days from the first patch submission to the posting of the first reviewer message [6].
	Average Review Rate (ARR)	Average review rate (KLOC/Hour) for each revision.
	Typical Review Window (TRW)	The length of time between the creation of a review and its final approval for integration, normalized by the size of the change.

Step 6: Assess the influence of multiple code review practices on software degradation. To assess the influence of the code review activity metrics in design degradation, we created a statistical model using the *multiple logistic regression* technique. In this model, we used all code review metrics presented in Table 3.3 as predictors of the likelihood of a code review to impact design degradation; i.e., whether each code review has either a decreasing or increasing impact in the degradation characteristics. We used *multiple logistic regression* because we are dealing with multiple possible predictors, and we have a binary variable as a response. To avoid the effect of *multicollinearity* on our data, we remove the code review metrics which have a pair-wise correlation coefficient above 0.7 [1].

Moreover, we used odds ratios to report the effect of the metrics over the possibility of a review impacting design degradation. Odds ratios are the increase or decrease in the odds of a review degradation impact occurring per "unit" value of a predictor (metric). An odds ratio < 1 indicates a decrease in these odds, while > 1 indicates an increase. Most of our metrics presented a heavy skew, to reduce it, we applied a $\log_2$ transformation on the right-skewed predictors and a x^3 transformation on the left-skewed. Furthermore, we normalized the continuous predictors in the model to provide normality. As a result, the mean of each predictor is equaled to zero, and the standard deviation to one. Finally, to ensure the statistical significance of the predictors, we employed the customary *p-value* of 0.05 for each predictor in the regression model.

Step 7: Manually analyze and classify reviews. In this step, we used a subset of code reviews that were manually classified in the work of Paixão et al. [15]. We performed a cross-check analysis of such reviews considering the commit messages, discussions between developers, and source code. We also filtered the reviews according to the criteria presented in Step 1, which resulted in a subset of 1,779 manually analyzed reviews. Based on this analysis, we conducted two classifications. In the first one, we classified reviews as *design-related* or *design-unrelated*, according to the developers' intent of improving the structural design of the system. Reviews were tagged as *design-related* when design improving intent was explicit either in the review's *descriptions* or in *discussions*. The second classification consisted in identifying reviews in which explicit design discussions occurred. We considered as *reviews with design discussions* those in which developers have demonstrated awareness of the possible impact of their changes in the system's design.

We performed such classifications according to the following definition of design: *software design is the result of design decisions that affect structural* *quality attributes, either positively or negatively.* The manual classification process was performed by two authors. Each author was responsible for analyzing the 1,779 code reviews and manually classifying them. We employed a two-phase process: 1) Each author solely and separately inspected and classified the same code reviews; 2) the author discussed all the reviews for which there was a disagreement in the classification until a consensus is reached.

All the data collected in the aforementioned steps as well as the set of manually classified code reviews are available in our replication package [58].

3.4 Results and Discussions

Section 3.4.1 discusses how to modern code review affect design degradation **(RQ_1)**. Section 4.4.2 discusses the design degradation evolution patterns during the code review process **(RQ_2)**. Finally, Section 4.4.3 discusses the relationship between different code review practices and design degradation **(RQ_3)**.

3.4.1 Manifestation of Design Degradation

We address **RQ_1** by analyzing the impact of merged reviews on two degradation characteristics: (1) density and (2) diversity of symptoms. Table 3.4 shows the frequency of each type of impact in all target systems. Columns represent the symptom characteristics. We decomposed those characteristics into four distinct groups: Density of Coarse-grained Smells (CG Density), Density of Fine-grained Smells (FG Density), Diversity of Coarse-grained Smells (CG Diversity), and Diversity of Fine-grained Smells (FG Diversity). They represent the amount and heterogeneity of degradation symptoms at different granularity levels. We also categorized the impact of reviews into positive, negative, and invariant. **Positive** are those that end up reducing the degradation characteristic, while **negative** ones are those that contribute to increasing the degradation characteristic. Finally, **invariant** reviews are those that do not affect the degradation characteristic.

Table 3.4: Type of Impact of Merged Code Reviews

Number of Merged Reviews Per Type of Impact				
Impact	**CG Density**	**FG Density**	**CG Diversity**	**FG Diversity**
Positive	1,879 (12%)	2,155 (15%)	101 (<1%)	11 (<1%)
Negative	4,876 (33%)	7,081 (47%)	137 (<1%)	49 (<1%)
Invariant	8,216 (55%)	5,735 (38%)	14,733 (99%)	14,911 (99%)

Invariant reviews are predominant. Table 3.4 shows that most merged reviews are invariant regarding the evaluated characteristics. The only case in which the proportion of invariant reviews is below 50% is for the density of fine-grained smells. In this case, most of the reviews (47%) had a negative effect. This result suggests that either (i) modern code review is not enough to avoid design degradation or (ii) code review is enough despite being predominantly invariant. In Section 4.4.2, we explore these two possibilities in detail.

Low impact on the diversity of symptoms. The last two columns of Table 3.4 reveal that most reviews do not impact on the diversity of both categories of symptoms. This happens because the diversity is only impacted when all occurrences of a smell type are removed from the changed code or when a new smell type is introduced. We noticed that the introduction of new types of smell usually occurs in the early stages of development since the codebase is still small and less complex. Complete removals of specific smell types usually occur when significant changes are made to the design structure of the system. An example of this occurred in the review number 11,099 [56] of the spymemcached system.

Impact of reviewing design related tasks. As explained in Section 4.3, we classified a sub-set of reviews into two groups: design-related and design-unrelated reviews. We used the Chi-Square test to compare the impact of both groups of reviews on the degradation characteristics. Table 3.5 shows the results for the density of coarse-grained and fine-grained smells. We will not discuss the results for diversity as they were not statistically significant. Nevertheless, all the results are available in our replication package [58].

Table 3.5: Chi-Square Results for Evaluating the Dependency Between the Type of Impact and the Relation with Design

Impact	**CG Density**		**FG Density**	
	Design Related	**Design Unrelated**	**Design Related**	**Design Unrelated**
Positive	**190 (156.35) [7.24]**	**125 (158.65) [7.14]**	151 (131.53) [2.88]	114 (133.47) [2.84]
Negative	443 (477.98) [2.56]	520 (485.02) [2.52]	535 (566.33) [1.73]	606 (574.67) [1.71]
Invariant	250 (248.67) [0.01]	251 (252.33) [0.01]	197 (185.14) [0.76]	176 (187.86) [0.75]
Chi Square	X^2 = 19.4775, p-value = .000059		X^2 = 10.672, p-value = .004815	

The last line of Table 3.5 shows the Chi-Square factors (X^2) and the *p-values*. The other lines represent the impact type (positive, negative, and invariant) of classified reviews. The 2nd and 3rd columns show the distribution of reviews into the two compared groups regarding their impact on the density of coarse-grained smells, while the last two columns show the same information for the density of fine-grained smells. The numbers in parentheses represent the number of reviews that are statistically expected in each cell, given their classification regarding impact (lines) and design-relation (columns). Outside of the parentheses is the number of reviews that, in fact, were observed in each cell. Finally, in brackets is a value that represents how much each the observed number of reviews contributed to the composition of the Chi-Square factor. The higher the difference between expected and observed number of reviews in the cell, the higher will be the value.

Table 3.5 shows that the number of design-related reviews with a positive or invariant impact on the density of smells is higher than expected. Moreover, the number of design-related reviews with negative impact is also lower than expected. Conversely, there were more design-unrelated reviews with a negative impact than would be expected. This result is consistent and statistically significant for both coarse-grained and fine-grained smells. We interpret this result as evidence that design-related reviews tend to have a more positive and neutral impact than other types of review. This means that when there is an explicit intention to improve the design, the degradation can be reduced or at least remain invariant.

Impact of design discussions. To better characterize the reviews, we conducted another comparison. In this case, we compared reviews where there were explicit design discussions (With DD) with reviews without discussions related to design (Without DD). Once again, we applied the Chi-Square test to compare both groups regarding their impact on the density and diversity of symptoms. The results were not statistically significant for the diversity of CG smells and for both symptoms of FG smells. Table 3.6 shows the result of this comparison for the density of coarse-grained smells.

Table 3.6: Chi-Square Results for Evaluating the Dependency Between the Type of Impact and the Presence of Design Discussions

Impact	**Coarse-grained Smells**	
	With DD	**Without DD**
Positive	63 (64.10) [0.02]	252 (250.90) [0.00]
Negative	**235 (195.96) [7.78]**	728 (767.04) [1.99]
Invariant	**64 (101.95) [14.12]**	437 (399.05) [3.61]
Chi-Square	X^2 = 27.5229, p-value <.001	

Our results reveal that design discussions tend to be associated with a negative impact on the density of coarse-grained smells. We hypothesize that this result occurred as structurally degraded code usually draws more attention from reviewers, often causing some type of design discussion. Nevertheless, as we observed in our manual analysis, such discussions may not contribute to the reduction of severe design problems.

Finding 1: Reviews with explicit intents of design improvement tend to reduce or avoid design degradation. However, the sole presence of design discussions is not enough for avoiding design degradation.

3.4.2 Degradation Evolution along a Single Review

We address **RQ$_2$** by identifying four patterns of design degradation evolution along with a single review. The procedure that we followed to identify these patterns is defined in Section 3.3.2 (Step 4). For each degradation characteristic (density and diversity), we classified reviews into four patterns: invariant, positive, negative, and mixed. Such patterns can be summarized as follows. **Invariant** is composed of reviews in which the characteristic remained the same across all the revisions submitted during the code review. **Positive** is the pattern for reviews in which the last revision reduces the degradation characteristic when compared to the previous ones. **Negative** groups reviews for which the last revision presents an increase in the degradation characteristic when compared to the previous ones. Finally, **mixed** is composed of reviews with signs of reduction and an increase of the characteristic along the revisions.

Table 3.7 shows the degradation evolution within reviews grouped by symptom category, i.e., *coarse-grained* and *fine-grained* smells, and by characteristic, i.e., *density* and *diversity*. We also group the reviews by type, i.e., *design-related* or *design-unrelated*, and different levels of design discussion. In this table, we show information about a subset composed only of reviews that improved (*Improvement* columns) or degraded (*Degradation* columns) the structural design, according to the degradation characteristics. For each case, we present the ratio of reviews classified into the four aforementioned patterns: invariant *(Inv)*, positive *(Pos)*, negative *(Neg)*, and mixed *(Mix)*.

Table 3.7: The Ratio of Reviews Grouped by Type, Level of Design Discussion, Symptom Category, Characteristic and Degradation Evolution

Coarse-grained Smells

Type	**Design discussion**	**Density**								**Diversity**							
		Improvement				**Degradation**				**Improvement**				**Degradation**			
		Inv	**Pos**	**Neg**	**Mix**	**Inv**	**Pos**	**Neg**	**Mix**	**Inv**	**Pos**	**Neg**	**Mix**	**Inv**	**Pos**	**Neg**	**Mix**
Design-related	**Never**	77%	0%	1%	22%	60%	0%	1%	38%	50%	0%	0%	50%	50%	0%	0%	50%
	Description	77%	0%	0%	23%	65%	3%	0%	32%	100%	0%	0%	0%	100%	0%	0%	0%
	Comments	11%	0%	0%	89%	17%	0%	6%	77%	50%	0%	0%	50%	-	-	-	-
	Both	29%	0%	14%	57%	25%	3%	5%	68%	-	-	-	-	0%	0%	100%	0%
	All	71%	0%	1%	27%	54%	1%	2%	43%	60%	0%	0%	40%	54%	0%	8%	38%
Design-unrelated	**Never**	69%	4%	5%	21%	61%	1%	3%	35%	71%	0%	7%	21%	71%	0%	0%	29%
	Description	60%	40%	0%	0%	65%	0%	0%	35%	-	-	-	-	0%	0%	0%	100%
	Comments	0%	0%	0%	100%	20%	0%	0%	80%	-	-	-	-	100%	0%	0%	0%
	Both	33%	0%	0%	67%	17%	0%	0%	83%	-	-	-	-	-	-	-	-
	All	66%	6%	5%	24%	57%	1%	3%	40%	71%	0%	7%	21%	71%	0%	0%	29%

Fine-grained Smells

Type	**Design discussion**	**Density**								**Diversity**							
		Improvement				**Degradation**				**Improvement**				**Degradation**			
		Inv	**Pos**	**Neg**	**Mix**	**Inv**	**Pos**	**Neg**	**Mix**	**Inv**	**Pos**	**Neg**	**Mix**	**Inv**	**Pos**	**Neg**	**Mix**
Design-related	**Never**	73%	1%	1%	25%	59%	1%	2%	39%	100%	0%	0%	0%	67%	0%	0%	33%
	Description	59%	9%	0%	32%	56%	0%	4%	40%	-	-	-	-	100%	0%	0%	0%
	Comments	11%	22%	0%	67%	14%	0%	0%	86%	-	-	-	-	-	-	-	-
	Both	13%	0%	0%	88%	24%	3%	8%	66%	-	-	-	-	0%	0%	0%	100%
	All	62%	4%	1%	33%	52%	1%	3%	44%	100%	0%	0%	0%	56%	0%	0%	44%
Design-unrelated	**Never**	52%	2%	4%	41%	52%	2%	4%	41%	0%	0%	0%	100%	43%	0%	0%	57%
	Description	60%	40%	0%	0%	34%	3%	3%	59%	0%	0%	0%	100%	0%	0%	0%	100%
	Comments	38%	0%	0%	63%	14%	0%	0%	86%	-	-	-	-	0%	0%	0%	100%
	Both	100%	0%	0%	0%	13%	0%	0%	88%	-	-	-	-	-	-	-	-
	All	68%	1%	5%	26%	48%	2%	4%	46%	0%	0%	0%	100%	30%	0%	0%	70%

Degradation characteristics and their variation across revisions. For design-related reviews that presented signs of reduction in degradation characteristics (3rd and 5th main columns of Table 3.7), we observed that density and diversity of coarse-grained smells remain invariant in 71% and 60% of the cases, respectively. A similar observation applies when we consider fine-grained smells. The density and diversity of such category of smells are invariant in 62% and 100% of the cases, respectively. On the other hand, for design-related reviews with signs of degradation (4th and 6th main columns of Table 3.7), the ratio of invariant reviews for coarse-grained smells is 54% for both density and diversity. For fine-grained smells, the values of density and diversity are invariant in 52% and 56% of the cases, respectively.

These observations indicate that 64% of design-related reviews, tend to remain invariant, i.e., preserve the same design impact throughout all revisions. Conversely, 52% of design-unrelated reviews tend to remain invariant. We also observed that, for reviews with signs of improvement, the degradation impact tends to change more when the reviewers provide design feedback. In such cases, for design-related reviews, the impact on the density and diversity of coarse-grained smells of the latest merged revision was different than the previous ones in 88% and 50% of the cases, respectively. For fine-grained smells, we observed that in 88% of the cases, the density changed in the latest merged revision. A similar pattern was observed in reviews that presented signs of degradation.

Finding 2: For design-related and design-unrelated reviews, the degradation impact on the latest merged revision in comparison with all previous ones tends to remain invariant in 64% and 52% of the cases, respectively.

Signs of degradation reduction. Table 3.7 presents other observations. For design-related reviews that reduce the density or diversity of coarse-grained smells (3rd and 5th main columns), we did not observe any positive evolution, i.e., changes that improve the structural quality. This happens even when the reviewers provide feedback on the design quality. On the other hand, for the density of fine-grained smells, we observed positive and negative evolution patterns in 4% and 1% of the cases, respectively. Such results are surprising since the ratio of reduction of 4% only for fine-grained smells was below expectations and different from studies that investigate the impact of refactoring, i.e., a technique that is commonly used during code review [15, 60, 61].

As expected, the ratio of design-related reviews with a negative evolution is higher in reviews that present signs of degradation (4th and 6th main columns of Table 3.7). In such cases, for the density of coarse-grained smells, we observed positive and negative evolution patterns in 1% and 2% of the cases, respectively. Conversely, we observed a negative evolution pattern in 8% of degradation reviews for diversity. Regarding the density of fine-grained smells, we observed positive and negative evolution patterns in 1% and 3% of the cases, respectively. Again, we did not observe any positive or negative evolution related to diversity.

We also observed that for design-related reviews with signs of either improvement or degradation, in which the design is discussed in both the description and comments, there were not successive decreases of the density of coarse-grained smells. Conversely, we observed an increase in the density of coarse-grained smells with a ratio of negative evolution is 14%. This is a surprising finding as we expected that design feedback during code review would lead to improvements, i.e., a reduction of the design degradation. On the other hand, considering the level of design discussion in reviews with fine-grained smells, we observed that when the reviewers provide design feedback during code review, the evolution patterns are predominantly positive (22%) and mixed (67%).

These results indicate that design discussions during code review may influence the review's impact on degradation characteristics. However, such impact tends to be positive only for the density of fine-grained smells, indicating that design discussions provided by developers during code review do not help the developers to decrease coarse-grained smells, i.e., such symptoms are often aggravated rather than minimized. In fact, fine-grained smells are simpler to remove and refactor as they represent smaller readability and understandability problems [23]. Conversely, coarse-grained smells are often hard to remove as they represent more severe problems, requiring more complex refactorings [60].

Finding 3: Code reviews usually do not reduce coarse-grained smells, even when there is design feedback.

Degradation symptoms and their fluctuation during code review. Finally, we observed that the ratio of design-related reviews with a sign of improvement in density and diversity of coarse-grained smells are often classified as mixed evolution in a ratio of 27% and 40% of the cases, respectively. Regarding the impact on the density of fine-grained smells, the ratio of reviews with a mixed evolution is also the highest in 33% of the cases when compared to the reviews with positive and negative evolution. Surprisingly, this pattern of evolution holds even when the developer provides some feedback on the design quality. A similar observation applies when we consider the design-related reviews that presented signs of degradation, in which nearly 42% of reviews are classified as mixed.

In our manual analysis, we observed that mixed evolution usually occurs as a side effect of two factors: (i) deleting of duplicated or unnecessary code; and (ii) reorganizing the code to make it more reusable. For instance, consider the code review 9,015 from the linuxtools, which caused a significant improvement regarding coarse-grained smells. This review has a total of 11 revisions, in which fluctuation occurred after the following feedback provided by reviewer: *"This class and ProviderOptionsTab are almost identical except for a few small differences [...]. Would it be possible to define some abstract class and have these inherit override just what they need?"*. Such suggestion led to an increase in smells that affect abstraction and encapsulation issues.

These observations suggest that even if the developers identify and remove fine- and cross-grained smells, they still will not be able to see all the ramifications of the impact of their changes along revisions. However, at the end of the code review process, i.e., in the last merged revisions, the developers tend to preserve the positive impact on the system's internal structure. We conjecture that this happens because the existing modern code review tools still lack a mechanism to provide developers a just-in-time recommendation about the impact of their changes on software design degradation [33].

Finding 4: Nearly 34% of design-related reviews present a mixed evolution. This happens even in reviews that present signs of improvement and degradation. This result motivates the proposition of recommenders to better support developers, in the improvement of design quality and prevention of design problems during code review.

3.4.3 Code Review Practices and Design Degradation

We address **RQ**$_3$ by assessing the influence of code review practices on software degradation. We have applied a *multiple logistic regression* to support this assessment (Step 6 of Section 3.3.2). Table 3.8 summarizes the main results. Each row represents the results for each project, separated by symptom category (coarse-grained (CG) and fine-grained (FG) smells) and degradation characteristic (density and diversity). The "all" row represents the results for the data of all projects combined. The gray cells represent the metrics that presented statistical significance relation in a specific combination of symptoms and characteristics. Moreover, we used the $\uparrow$ symbol to indicate a degradation risk-increasing effect, and the $\downarrow$ symbol to indicate a degradation risk-decreasing effect.

The data of three projects were omitted from the Table 3.8, but can be fully seen on our replication package [58]. We removed these data because only a few metrics were statistically significant, but they are considered on the “all” row. Additionally, the control variables (Section 3.3) were omitted, but when applied on the model, only the `DiffComplexity` and `DiffSize` variables were statistically significant across projects. To understand if the control variables were collinear with the code review metrics, we executed the model only with the control variables, and the results were similar. This implies that our code review metrics were capable of working as a standalone model to verify the risk-increase or risk-decrease effect on degradation in each system. Next, considering only statistically significant cases, we discuss the code review practices by risk-increasing and risk-decreasing effects as follows.

Table 3.8: Code Review Activity Metrics of Reviews with Positive and Negative Impact, Grouped by System, Symptom, Characteristic and Dimension. The ↑ Symbol to Indicate a Risk-increasing Effect, and the ↓ Symbol to Indicate a Risk-decreasing Effect

System	**Symptom**	**Characteristic**	**Review Intensity**			**Review Participation**			**Review Time**			
			DL	**PRWF**	**CDCR**	**NA**	**NNAV**	**PRD**	**RL**	**TRW**	**ARR**	**RD**
jgit	**CG**	**Density**	1.023	1.129	1.713	1.008	0.772 ↓	1.143	1.191	0.856	1.127	1.04
		Diversity	1.046	1.134	1.705	1.007	0.768 ↓	1.132	1.181	0.865	1.118	1.034
	FG	**Density**	1.334	1.04	1.472	1.014	0.777 ↓	0.93	1.028	1.279 ↑	0.897	1.035
		Diversity	1.334	1.04	1.472	1.014	0.777 ↓	0.93	1.028	1.279 ↑	0.897	1.035
egit	**CG**	**Density**	1.497 ↑	0.941	0.342 ↓	0.97	1.116	1.01	1.306 ↑	0.701 ↓	1.094	1.018
		Diversity	1.321	0.994	0.4 ↓	0.961	1.107	1.075	1.279 ↑	0.587 ↓	1.294 ↑	1.009
	FG	**Density**	1.416 ↑	0.976	0.743	0.956	0.76 ↓	1.012	1.322 ↑	1.069	0.903	1.169 ↑
		Diversity	1.354 ↑	0.974	0.701	0.961	0.767 ↓	1.025	1.366 ↑	1.069	0.903	1.153 ↑
linuxtools	**CG**	**Density**	1.112	1.569 ↑		0.864	1.166	1.257 ↑	0.912	1.249	0.713 ↓	1.118
		Diversity	1.116	1.582 ↑		0.854	1.151	1.268 ↑	0.898	1.216	0.732 ↓	1.14
	FG	**Density**	0.873	1.533 ↑		0.842 ↓	1.231 ↑	1.394 ↑	0.94	1.021	0.826	1.11
		Diversity	0.873	1.533 ↑		0.842 ↓	1.231 ↑	1.394 ↑	0.94	1.021	0.826	1.11
platform.ui	**CG**	**Density**	1.073	1	0.723 ↓	0.971	1.175	0.932	1.032	0.827	1.268	0.872 ↓
		Diversity	1.071	1	0.717 ↓	0.946	1.176	0.938	1.031	0.82	1.275 ↑	0.877 ↓
	FG	**Density**	1.051	0.998	0.737 ↓	0.984	1.053	0.979	1.265 ↑	0.899	1.272 ↑	0.909
		Diversity	1.051	0.998	0.737 ↓	0.984	1.053	0.979	1.265 ↑	0.899	1.272 ↑	0.909
All	**CG**	**Density**	1.115 ↑	1.145		0.96 ↓	1.16	1.005	1.045 ↑	0.887	1.076 ↑	0.917
		Diversity	1.106 ↑	1.162		0.948 ↓	1.158	1.016	1.037 ↑	0.845 ↓	1.12 ↑	0.913
	FG	**Density**	1.155 ↑	1.092		0.962	1.043	1.025 ↑	1.079 ↑	1.101	0.957	0.954
		Diversity	1.144 ↑	1.091		0.963	1.045	1.028 ↑	1.084 ↑	1.102	0.955	0.953

Risk-increasing effect on software degradation. We observed that the metrics `DL`, `PRWF`, `NNAV`, `PRD`, `RL`, `TRW`, `ARR`, and `RD` presented a risk-increase tendency. By analyzing each metric individually, we observed that four metrics sustained a risk-increasing tendency across projects: Discussion Length (`DL`), Proportion of Revisions without Feedback (`PRWF`), Proportion of Review Disagreement (`PRD`), and Review Length (`RL`). Such behavior was expected for the `PRWF` and `PRD` metrics, since they confirm the rationale presented in Table 3.3. Conversely, the results for the `DL`, `RL` and `TRW` metrics are unexpected, since they differ from that were reported by related studies [6, 24, 42], we will discuss latter in this section. By considering the degradation symptoms, all metrics presented themselves as good predictors for all symptoms and characteristics, except for a few metrics: the Number of Non-Authors Votes (`NNAV`) metric, that on the project linuxtools was only related to fine-grained smells, as a risk-increasing factor; and Average Review Rate (`ARR`) that was related to coarse-grained smells on 66% of the cases as a risk-increasing factor. Moreover, the Proportion of Revisions without Feedback (`PRWF`) metric stood out as the metrics with the highest risk-increasing effect, followed by the Discussion Length (`DL`) metric, reaching values of 1.58, and 1.49, respectively.

Long discussions are not reflected as concerns about structural degradation. By considering the intensity dimension metrics (Table 3.8), we observed that two of the three metrics presented a degradation risk-increase tendency in at least one project. The metrics Discussion Length (`DL`) and Proportion of Revisions without Feedback (`PRWF`) preserved this behavior in most of the significant cases. This observation indicates that developers tend to introduce more instances and more types of degradation symptoms in reviews that either has longer discussions or do not have any discussions started by human participants. As illustrated in our motivating example (Section 4.2), and also confirmed in our manual analysis, long discussions do not necessarily indicate that developers and reviewers are concerned about the structural quality of the code. Reviewers can often be concerned with functional aspects of the system, paying less attention to possible signs of degradation. Thus, when there are extensive discussions about specific features, the code tends to undergo further modifications that increase design degradation.

Finding 5: Reviews for which the practice of long discussions was applied are often associated with a higher risk of software degradation.

A high proportion of review disagreement leads to a degradation risk-increasing effect. By considering the metrics of the participation dimension, only the Proportion of Review Disagreement (`PRD`) metric presented a risk-increase tendency across all projects. Nevertheless, the Number of Non-Author Voters (`NNAV`) metric raised some concerning results on the linuxtools system. As illustrated in our motivating example (Section 4.2), this result indicates that reviews with a high rate of acceptance discrepancy tend to introduce more instances and more types of degradation symptoms.

Finding 6: Reviews following the practice of participation with a higher rate of review disagreement lead to a higher risk of software degradation.

Lack of reviewers' attention and code review speed increase the risk of degradation. Table 3.8 also shows that the longer (`RL`) and faster (`ARR`) review takes to be finished, the higher the risk of degradation. At first, this result seems to be counter-intuitive. However, in our manual analysis, we observed that certain reviews often take a longer time due to the lack of attention of the reviewers during the code review. This observation suggests that reviewers will be able to identify more design problems that are overlooked during the code review, whether they perform a careful code inspection with an appropriate code review rate. By considering our motivating example again, we can observe that this review takes a long time from the first patch submission to the reviewer's acceptance for integration (Aug 16, 2015 to Oct 9, 2015). Moreover, we observed a lack of attention from the reviewer of revision 7 to revision 10. For instance, the code author has addressed the reviewer's lack of attention with the following comment: *"[...] do you have time for a review?"*. After that comment, the code author only got a response from the reviewer after 16 days. Within this time, the code author continued to modify the source code without feedback from the reviewers.

Finding 7: Reviews tend to be longer due to the lack of attention from the reviewer during the code review process, and this increases the risk of software degradation.

Risk-decreasing effect on software degradation. Table 3.8 shows that the Churn During Code Review (`CDCR`), Number of Authors (`NA`), Number of Non-Authors Votes (`NNAV`), Time Review Window (`TRW`), Average Review Rate (`ARR`), and Response Delay (`RD`) metrics present a risk-decreasing tendency. Moreover, the metrics `NA`, and `NNAV` also presented consistent results, as the NNAV showing a risk-decreasing likelihood for most target systems, except for the linuxtools system. Moreover, this metric was risk-decreasing in 75% of the statistically significant cases. Thus, it can be considered as a reliable predictor of reduction in structural degradation. `CDCR` showed good results on egit and platform.ui projects, performing very good results on egit (0.34). Looking over the dimensions, there is evidence that the CDCR metric is a reliable predictor of risk-decreasing for the Intensity Dimension, the NNAV represents the Participation Dimension, on Time Dimension no metric really outcome as a reliable predictor. We can see a minor difference between symptoms; the risk-decreasing appeared more (60%) on the coarse-grained symptom.

Active engagements of multiple reviewers decrease the risk of degradation. By considering the participation dimension, the Number of Authors (NA), and the Number of Non-Authors Votes (NNAV) metrics showed to be reliable predictors of risk-decreasing. However, only the NA preserved this behavior across projects. Thus, the higher the number of authors, the higher will be the likelihood of degradation risk-decreasing. This observation is aligned with previous studies [40, 50, 11, 8], by suggesting the greater the number of authors revising the proposed changes during reviews, the more design issues could be identified and removed, especially coarse-grained smells, whose identification and removal often required a better understanding of the codebase.

Finding 8: Reviews with active engagements of multiples reviewers tend to present a degradation risk-decreasing effect, especially for coarse-grained smells.

3.5 Threats to Validity

We discuss threats to the study validity [2] as follows. **Construct and Internal Validity.** Aspects such as the precision and recall of degradation symptoms may have influenced the results of this study. We tried to mitigate this threat by selecting a detection tool that has been successfully used in recent studies involving design degradation [4, 29, 52].

We have selected a set of 12 code review activity metrics that helped us measure different dimensions of code review practices, i.e., intensity, participation, and time. The rationales for using metrics are supported by previous studies (e.g., [8, 24]). We wrote scripts to automate the design degradation evolution pattern computation and code review metrics. These scripts were validated by two of the paper authors. Regarding the code review activity metrics, we measured some metrics based on heuristics. For instance, we have assumed that the review length is the time that elapses between the time a patch has been uploaded and when it has been approved for integration. Thus, although there is a limitation of measuring the code review practices, we rely on state-of-the-art practices based on heuristics to recover this kind of information.

Conclusion and External Validity. We carefully performed our descriptive and statistical analysis. About the descriptive analysis, four paper authors contributed to the analysis of code review impact on design degradation. For the statistical analysis, we rely on the *Multiple Logistic Regression*, as previously stated in Section 4.4.3, we reduced the heavy skew of our metrics applying a $\log_2$ transformation on the right-skewed predictors and a x^3 transformation on the left-skewed. Moreover, we normalized the continuous predictors in the model to provide normality, and, to ensure the statistical significance, we employed the customary *p-value* of 0.05 for each predictor in the regression model. Furthermore, in our statistical model, we controlled some factors that may affect our outcomes via product and process metrics.

Regarding the qualitative analysis of design-related reviews, we employed a two-phase manual classification procedure. In the first, all reviews were classified by two authors. In the second phase, for all reviews in disagreement, both authors discussed to reach a unified classification. Finally, the analysis of code review impact on design degradation is based on two degradation characteristics – density and diversity of symptoms. One might expect different results using other characteristics. We relied on density and diversity because they are widely-adopted for design degradation analysis and have been evaluated in previous studies [29, 34, 53].

3.6 Conclusion and Future Work

In this work, we analyzed the impact of modern code review on evolution of design degradation, by mining code review data from two large open source communities. Our findings pointed out that design discussions may not be enough for avoiding design degradation. Conversely, reviews with explicit intent to improve the design tend to have a positive or invariant impact on design degradation. We also observed that, during the revisions of each review, there is often a wide fluctuation of design degradation. Such fluctuations often result in the amplification of design degradation, even in design-related reviews. Finally, we observed that certain code review practices might increase the risk of design degradation, including long discussions, and a high rate of reviewers' disagreement.

As future works, we aim to (i) evaluate the effect of code reviews on other types of degradation symptoms, and different characteristics of design degradation; (ii) expand the code review metrics and dimensions to understand their role on software degradation, and (iii) investigate mechanisms to better support developers, in the continuous improvement of design quality during code review.

3.7 Summary of Chapter 3

In order to address our **first research problem** (see Section 1.2) this chapter presented a seminal study that characterizes how the process of design degradation evolves within each review and across multiple reviews. To this end, we investigate 14,971 code reviews from seven software projects. Moreover, we analyze a comprehensive suite of metrics to enable us to observe the influence of certain code review practices on combating or even accelerating design degradation.

Our results show that the majority of code reviews had little to no design degradation impact in the analyzed projects. More interestingly, this observation also applies to reviews with an explicit concern on design. Surprisingly, the practices of long discussions and high proportion disagreement in code reviews were found to increase design degradation. Finally, we also discuss how the study findings shed light on how to improve the research and practice of modern code review.

4
Predicting Design Impactful Changes in Modern Code Review: A Large-Scale Empirical Study

An earlier version of the work in this chapter appears in
the Proceedings of the 18th International Conference on Mining
Software Repositories (MSR) [70].

The first published work performed through this doctoral thesis (Chapter 3) revealed various findings regarding the impact of modern code review on software design degradation. Specifically, the Finding 4 about the presence of a mixed evolution degradation during code reviews, gave us insight about the need to propose of mechanisms to better support developers, in the improvement of design quality and prevention of design problems during code review.

With this in mind, and knowing that code review is affected by social and technical aspects, in this chapter, the paper *Predicting Design Impactful Changes in Modern Code Review: A Large-Scale Empirical Study* will be presented in its entirely. The paper was published in the 18th International Conference on Mining Software Repositories (MSR) [70]. This study comprehends the second contribution of this Doctoral research: *The understanding of the role of social aspects in distinguishing and predicting (un)impactful design changes.* (see Section 1.4.2).

4.1 Introduction

Modern code review is a practice that has been widely adopted by major companies [14, 26]. It is typically a lightweight, informal, asynchronous, and tool-assisted practice aimed at monitoring, detecting and removing issues that were introduced during development tasks [14]. Supported by platforms such as Gerrit and GitHub, the code review process is initiated by one developer referred to as the *owner*, which modifies the original codebase and submits a new code change to be reviewed by other developers – the so-called *reviewers*.

A key concern of all stakeholders involved in a code review, including code owners, reviewers, and team managers, is to become aware of ongoing changes impacting the design [15, 60, 136]. The underlying motivation is to monitor and inspect those design impactful changes so that stakeholders can anticipate, find, and remove signs of design degradation before the end of a code review. Otherwise, those design harmful changes can become prevalent after the code review [24, 40]. If design impactful changes are not discriminated and brought to attention early, it will increase the likelihood of those changes finding their way into the system for several reasons. These include reviewers deviating their effort to other quality checks in later review stages or even starting to focus only on the main purpose of the issue being resolved. Moreover, design-related changes become harder to revert towards the end of the review as many other inter-related modifications were already realized as the review progresses. In fact, these reasons explain why many design impactful changes get unnoticed as nearly 40% of pull request rejections are related to design issues [156].

Early identification of impactful changes that degrade the software design is important during code review [33, 74, 136]. Code reviewers are expected to inspect ongoing changes and provide prompt feedback to code owners in the form of comments. In turn, the code owner should fix and forward the new version of the code for inspection. Such a procedure is repeated in multiple iterations, which are called *revisions*. This sequence of revisions ends up with either the acceptance or rejection of the change into the codebase [13, 66].

Despite its importance, recent studies found these modern code review practices are far from being sufficient to prevent design-degrading changes [32, 136]. Design degradation occurs whenever a change introduces poor structural decisions, i.e., design smells [34, 46, 69, 134]. Tools for detecting design smells tend to be inefficient when a change is still at its early stages. For instance, the full addition of a new feature can be complex and its realization needs many revisions to be accomplished. Moreover, other types of change can be complex as well. In fact, several changes in software projects are fully realized only after many revisions in a single review.

If these harmful changes are not reversed early, i.e., before a code review is ended, rework will be necessary after the changes of the last merged revision. Further changes with time-consuming refactorings will have to be applied later. Given the costs of design refactorings, they are unlikely to be applied and smells will be compounded over time, thereby accelerating the design degradation [15, 29, 136].

Due the importance of the early identification of design relevant changes during the review process, stakeholders must use all available information during the code reviewing process. In code review platforms, stakeholders have either technical or social information at their disposal to be used as additional information, both before or after each change. Social information are often available as soon as code review starts. Social information includes: number of prior code changes submitted by the code owner, and centrality of the code owner on the collaboration graph [7, 97, 98, 99]. Technical information are available after changes and revisions done during the review. Examples of technical information include: number of times a file has been changed and types of change [49, 94, 96, 99].

The advantage of using technical and social metrics to characterize and predict failures have widely been studied [93, 95, 100, 101]. However, their use to discriminate and predict design impactful changes is rarely studied [32, 136]. In fact, such metrics can act as indicators of design impactfulness of ongoing changes along the code review process. Hence, to the best of our knowledge, there is no study on which types of metrics can be used as effective features in Machine Learning (ML) algorithms to accurately predict design impactful changes.

This paper presents results of a large-scale empirical study that investigates whether and how technical and social metrics can be used to predict design impactful changes. To this end, we analyzed more than 50k code reviews of seven real-world systems from two large open source communities. We mined and examined if a comprehensive suite of technical and social metrics can discriminate design (un)impactful changes. Then, we explored the use of these metrics, as features for six interpretable ML algorithms, which tend to offer an effective prediction for different tasks and contexts, e.g., [114, 93]. Finally, we evaluated the predictive power of the selected features and algorithms to assist developers to automatically determine whether a code change is impactful.

Our key findings and contributions are: (1) both social and technical metrics are able to distinguish design (un)impactful changes; (2) the use of technical features results in more accurate predictions, when compared to the social ones; (3) features related to the code change, commit message, and file history dimensions are effective for differentiating (un)impactful changes; (4) *Random Forest* and *Gradient Boosting* have shown to be the most accurate in predicting design impactful changes; and (5) an enriched dataset and replication package that allows researchers to investigate the context and motivations behind design impactful changes during code reviews.

4.2
Motivating Example

Next we show the importance of considering design impact and using social and technical aspects during code review. To this end, we rely on two scenarios of the jgit system in which code review is conducted on the Gerrit platform.

Scenario A. Let us consider the review 3345 [131], composed of seven revisions, in which two developers performed a major change to "*Replace TinyProtobuf with Google Protocol Buffers*". After the last revision, 12,215 insertions and 2,404 deletions were performed in 58 files. Additionally, 104 design smells were introduced, leading to structural degradation related to the lack of abstraction (46), encapsulation (17), and modularity (42). Interestingly, design impact was not mentioned during revisions of the reviews of this major change. In other words, the replacement of a third-party component was conducted without an explicit concern with possible side effects that the change could introduce into the system.

Scenario B. Now consider the review 825 [132], which aims to "*Implement a Dircache checkout (needed for merge)*". This review had four reviewers and 18 revisions. After the first revision (R_1), we observed a inclusion of three smells: two Unutilized Abstraction and one Insufficient Modularization. Such smells were perceived by a developer, according to the following comment: "*One problem I faced here: we do have an abstraction to access the WorkTree when walking (reading) on it.*" Additionally, during the revisions (R_3 to R_{10}), we observed removals and reintroductions of the smells Unutilized Abstraction and Insufficient Modularization, characterizing a high fluctuation of design degradation during revisions. In other words, despite the developers identify degradation symptoms, they still are not able to see all the ramifications and impacts of their changes along with revisions.

Both scenarios illustrate the need for mechanisms to aid developers on identifying and preventing design impactful changes. Such mechanisms can rely on the large, diverse, rich information from social and technical aspects of the system and stakeholders in the code review. In **Scenario A**, when developers are unaware of the design impact of their change, a mechanism could have supported reviewers by automatically analyzing the discussions that took place on previous revisions and the components involved in the changes to predict the impact of changes in the current revision on the system design. In **Scenario B**, a mechanism could have analyzed the previous behavior of the code owner, and reviewers could better understand which changes are harming the source code.

In this context, one could argue: why not only using existing tools, such as Designite [4], to identify design degradation? Despite useful, such tools are limited. Besides the reasons already mentioned in Section 4.1, they rely only on static analysis in which detection strategies do not adapt per revision, not exploring the history of changes and multiple sources of information. They ignore the variation of technical and social aspects inherent to the code review process [32, 98, 136].

In summary, the motivations for our work are: (i) a real-world need for mechanisms to aid stakeholders in identifying impactful design changes during code review, (ii) availability of large and rich sources of information that can be used to make stakeholders aware of their changes' impact, and (iii) limitation of existing tools on using historical and dynamic information to support stakeholders during code review.

4.3 Study Settings

Section 4.3.1 presents both goal and research questions. Finally, the next sections shows the study steps and procedures.

4.3.1 Research Questions

Our study is guided by four research questions (RQs).

RQ$_1$: *Are design impactful changes significantly different from unimpactful ones in terms of social and technical metrics?* – Social and technical aspects may be avoiding or amplifying design degradation. To capture such aspects, we used a set of metrics detailed in Section 4.3.5. **RQ$_1$** aims at investigating which metrics are able to distinguish between design impactful changes and unimpactful ones.

RQ$_2$: *What is the performance of ML algorithms to predict design impactful and unimpactful changes?* – Once we show empirical evidence that distinguishes impactful and unimpactful changes, **RQ$_2$** aims at investigating the use of supervised ML techniques to assist developers in automatically make their decisions. In practice, some prediction algorithms perform better than others, depending on the task. Thus, we compare the performance of six interpretable ML algorithms: Logistic Regression, Naive Bayes, SVM, Decision Tree, Random Forest, and Gradient Boosting. We chose these algorithms since they provide an intuitive and easy to explain model [78, 79].

RQ$_3$: *How effective are the social and technical features as a proxy to the design impactfulness changes?* – **RQ$_3$** aims at evaluating and comparing the performance of both kinds of features. To this end, we applied the ML algorithm using three feature sets: a set using only social features, a set using only technical ones, and a set using technical and social features together. By answering **RQ$_3$**, we will be able to identify which kind of features are the best predictor, as well as the effectiveness of combining social and technical features. Furthermore, we also evaluated the effectiveness of a feature selection step for the three sets.

RQ$_4$: *What features are the best indicators of impactful design changes?* – **RQ$_4$** aims at understating which features are considered the most relevant by the models. Such knowledge is essential because, in practice, a model should be as simple and require as little data as possible. By answering **RQ$_4$**, we will be able to provide insights to practitioners and researchers as to what factors best indicate design impactful changes.

4.3.2 Code Review Data

To answer our RQ$_s$, we need not only information of the source code to distinguish design impactful changes, but also to analyze every code revision submitted along the code review process and investigate technical and social information related to those revisions. Thus, instead of mining code review data ourselves, we used the data provided by the Code Review Open Platform (CROP) [3], an open-source dataset that links code review data to software changes. All systems in CROP employ Gerrit as their code review tool. Hence, by using CROP, we have access to a rich dataset of code changes. To this end, given a certain system, CROP provides a complete copy of the entire codebase for each revision and its respective parent, which represents the system's codebase at the time of review. In other words, unlike the Git repository of a system, which contains only accepted revisions (i.e., the changes in the final revisions in a review), the CROP stores all revisions.

In our study, we adopt all Java systems included in the CROP dataset: four systems from the Eclipse community and three systems from the Couchbase community, as presented in Table 4.1. For sake of completeness, we remove the reviews whose status is "Open" since they may not have been assigned to reviewers, and the set of reviewers may still change.

Table 4.1: Software Systems Investigated in this Study

Community	System	# of Reviews	# of Revisions	Time span
Eclipse	**jgit**	5,304	13,578	10/09 to 11/17
	egit	5,220	12,814	9/09 to 11/17
	platform.ui	4,527	13,418	20/13 to 11/17
	linuxtools	4,074	11,418	6/12 to 11/17
Couchbase	**java-client**	909	2,622	11/11 to 11/17
	jvm-core	828	2,269	4/14 to 11/17
	spymemcached	536	1,379	5/10 to 7/17

4.3.3 Detection of Degradation Symptoms within Code Reviews

We investigated two categories of degradation symptoms, which are fine-grained (FG) and coarse-grained (CG) smells [30]. Although we do not focused on architectural smells, we empirically observed they follow similar trends in a complementary analysis. *FG smells* are indicators of structural degradation in the scope of methods and code blocks [30]. For instance, the Long Method is a FG smell that occurs in methods that contain too many lines of code. *CG smells* are symptoms that may indicate structural degradation related to object-oriented principles, e.g., abstraction, encapsulation, and modularity [30, 34]. An example of CG smell is Insufficient Modularization [30]. This symptom occurs in classes that are large and complex due to the accumulation of responsibilities. Such categories encapsulate a set of symptoms that are more perceived and used by developers in practice to identify and refactor source code locations degraded [22, 34, 36, 37].

For automatically detecting symptoms of such categories, we used a state-of-the-practice tool called DesigniteJava [4], which detected a total of 27 degradation symptoms types: 17 CG smells, and 10 FG smells. Hence, for each system, we identified these symptoms by considering each submitted revision that has undergone the code review process. Thus, we used CROP to access the versions of the system before and after the revision took place. Next, we detected the degradation symptoms in each version before and after revision. By following this methodology, we are guarantee that the introduced degradation symptoms between each version were solely introduced by the code changes in the revision. Table 4.2 lists the 27 symptoms types investigated in our study. We provide all descriptions, detection strategies, and thresholds for each type of symptom in the replication package [58].

Table 4.2: Degradation Symptoms Investigated in this Study

Coarse-grained Smells
Imperative Abstraction, Multifaceted Abstraction, Unutilized Abstraction, Unnecessary Abstraction, Deficient Encapsulation, Unexploited Encapsulation, Broken Modularization, Insufficient Modularization, Hub Like Modularization, Cyclic Dependent Modularization, Rebellious Hierarchy, Wide Hierarchy, Deep Hierarchy, Multipath Hierarchy, Cyclic Hierarchy, Missing Hierarchy, Broken Hierarchy.
Fine-grained Smells
Abstract Function Call From Constructor, Complex Conditional, Complex Method, Empty Catch Block, Long Identifier, Long Method, Long Parameter List, Long Statement, Magic Number, Missing Default.

4.3.4 Identification of Design Impactful Change Instances

We identified design impactful change instances in two steps: (i) identification of smelliness files by considering each revision of a code change, where each revision was compared to its parent, i.e., the codebase's version before any revision; and (ii) the computation of design degradation indicators.

To illustrate the first step, let $R_s = \{\mathrm{r}_1, \mathrm{r}_2, ..., \mathrm{r}_n\}$ be the set of submitted revisions for a given review s. For each revision in R_s (i.e., $\forall r_i$, $r_i \in R_s$), we use the CROP to retrieve the system's versions before and after r_i. Next, we check the presence of degradation symptoms (both CG and FG smells) in the files of r_i. The two file sets (before and after) might not be exactly the same, due to files created and deleted during the review process. Since the before version of each revision is its parent, we guarantee that the introduced degradation symptoms between each version were solely introduced by the code changes in r_i, avoiding the collateral effects of the rebase [113]. The output of this step is, for each revision in R_s, the version of the files impacted before and after the revision.

In the second step, we rely on an existing grounded theory [34] that explains that developers tend to consider multiple degradation characteristics in terms of *density* and *diversity* of symptoms. In addition, the use of *density* and *diversity* of symptoms for detecting design degradation is supported by other studies [36, 29]. Such studies show that degraded code elements tend to be affected by higher *diversity* and *density* of symptoms when compared to other code elements. However, before selecting *diversity* and *density* of symptoms as metrics, we compared the two lists of smells (by density and diversity) before and after revisions. As a result, we observed a small average variation (<1 type of smell/revision). Thus, computing the degradation in(de)crease with density imposes only a minor threat. Therefore, we take into account only the *density*, as a metric to measure the level of design degradation.

Therefore, for each selected system, we computed this characteristic in the context of each symptom category (CG and FG smells), for all the collected revisions. *Density* was computed for each version before and after revision, as the sum of the number of symptom instances in the set of smelliness source code files. The computation of *density* before and after revisions, allowed us to generate two different indicators of design degradation for each revision, where each indicator represents the differences in *density* of FG and CG smells.

In summary, a positive difference in the density of symptoms indicates an *increase in the degradation* as a result of the revision, therefore, harming the design. Similarly, a negative difference indicates a *reduction of the degradation* as a result of the revision. Finally, no variation indicates that there has been no structural design change. We consider that a design change is impactful when an *increase* or *reduction* in design degradation was observed as a result of submitted changes. Conversely, unimpactful changes are submitted changes that do not affect design degradation. Table 4.3 shows the number of revisions identified as design (un)impactful changes for each system and symptom category.

Table 4.3: The Number of Revisions Identified as Design Impactful Changes

Project	**Coarse-grained Smells**			**Fine-grained Smells**		
	Impactful	**Unimpactful**	**All**	**Impactful**	**Unimpactful**	**All**
java-client	751 (29%)	1,871 (71%)	2,622	1,340 (51%)	1,282 (49%)	2,622
jvm-core	630 (28%)	1,639 (72%)	2,269	1,054 (46%)	1,215 (54%)	2,269
spymemcached	423 (31%)	956 (69%)	1,379	586 (42%)	793 (58%)	1,379
platform.ui	2,431 (18%)	10,987 (82%)	13,418	4,460 (33%)	8,958 (67%)	13,418
egit	2,973 (23%)	9,841 (77%)	12,814	5,192 (41%)	7,622 (59%)	12,814
jgit	3,995 (29%)	9,583 (71%)	13,578	6,082 (45%)	7,496 (55%)	13,578
linuxtools	3,321 (29%)	8,097 (71%)	11,418	4,837 (42%)	6,581 (58%)	11,418
Total	**14,524 (25%)**	**42,974 (75%)**	**57,498**	**23,551 (41%)**	**33,947 (59%)**	**57,498**

4.3.5 Features for Design Impactful Change Prediction

We extracted a set of features able to capture both technical and social aspects of the changes involved in each revision of a code review. Each feature corresponds to a metric. We detail each feature and its description per dimension on Table 4.4.

From the **technical perspective**, we extracted 21 features related to source code, modification history of the files, and the textual description of the change. Moreover, these features were categorized into five dimensions: (i) **Size** consists of features related to source in their smallest granularity. Prior studies have found that large patches may need more effort to review [5]; (ii) **Diffusion** comprehends the features about changes distributed on two or more files (e.g., number of changed files). Prior studies also found that revisions, where their changes scatter across a large number of files or directories, may need more effort to review [94, 99]. Thus, we expected that the diffusion of a change could influence the likelihood of the change being impactful; (iii) **Complexity** comprehends the features on the complexity of a change. A code change with more code segments modified is likely more complex and requires more effort and time to be reviewed [111]; (iv) **File history** is composed of features related to the history of the files. The number of prior changes to a file can be a good indicator to detect degraded files. Moreover, files that are previously touched by more developers are more likely to introduce degradation symptoms [49, 94, 96]; and (v) **Textual** consists of features that capture textual characteristics of the commit message. Previous studies [82, 112] found that the description length of a patch is related to its likelihood of receiving poor comments. Additionally, the commit message

Table 4.4: Technical and Social Features Adopted in our Study

Technical Features		
Dimension	**Name**	**Description**
Size	**NLA**	Number of inserted lines in this code change
	NLD	Number of deleted lines in this code change
	CHURN	Number of lines added to and removed in this code change
	NFA	Number of added files in this code change
	NFD	Number of deleted files in this code change
Diffusion	**NCF**	Number of changed files in this code change
	NMD	Number of modified directories in this code change
	ME	Distribution of modified code across files in this code change
	NLANG	Number of programming languages used in this code change
	NFT	Number of file types in this code change
Complexity	**NSA**	Number of added code segments in this code change
	NSD	Number of deleted code segments in this code change
	NSU	Number of updated code segments in this code change
File history	**FM**	Number of times files in this code change were modified before
	FD	Number of developers who changed files in this code change
Textual	**ML**	Number of words in description of this code change
	BUG	Whether description of this code change contains word "bug"[1]
	FEAT	Whether description of this code change contains word "feature"[1]
	IMPR	Whether description of this code change contains word "improve"[1]
	DOC	Whether description of this code change contains word "document"[1]
	REFC	Whether description of this code change contains word "refactor"[2]
Social Features		
Dimension	**Name**	**Description**
Developer's Experience	**NC**	Number of prior code changes submitted by the owner of this code change
	NRC	**NC** in recent 120 days
	NDC	**NC** that contain at least one directory affected by this code change
	NR	Number of prior code changes the owner of this code change is assigned to inspect
	MR	Merged rate of prior code changes submitted by the owner of this code change
	RMR	**MR** in recent 120 days and normalized over the recent change number
	DMR	**MR** that contain at least one directory affected by this code change
Discussion Activity	**NIC**	Number of inline comments made by reviewers.
	NWIC	Sum of the all words of each inline comment.[3]
	PWIC	**NWIC** weighted by the number of inline comments.[3]
	NGC	Number of general comments made by reviewers.
	NWGC	Sum of the all words of each general comment.[3]
	PWGC	**NWGC** weighted by the number of general comments.[3]
	DL	Number of general comments and inline comments written by reviewers
Collaboration networks	**SD**	The degree centrality for a node v is the fraction of nodes it is connected to.[4]
	SCLOS	The inverse of the sum of all distances to all other nodes.[4]
	SB	The sum of the fraction of all-pairs shortest paths that pass through v.[4]
	SE	The centrality for a node based on the centrality of its neighbors.[4]
	SCLUST	The geometric average of the subgraph edge weights.[4]
	SKC	Maximal subgraph that contains nodes of degree k or more.[4]

[1]And more keywords based on previous work [155, 154]
[2]And more keywords based on previous work [133]
[3]We discarded comments made by non-human participants and applied the preprocessing in the text removing contractions, stop words, punctuation, and replacing numbers
[4]The initial node is the committer of the revision and all other nodes are the reviewers of each revision

may contain more information about a code change that may help reviewers comprehend the change more easily.

From the **social perspective**, we extracted 20 features that characterize the developer's experience, collaboration network, and participation in discussions. Moreover, these features were grouped into three dimensions: (i) **Developer's experience** comprises the features related to the previous experience of the code change owner. Previous studies [7, 98, 99] found that developer experience is essential information for predicting design issues. Such studies claim that if a developer often submits changes in recent times prior to the change, they will be more familiar with the recent developments of the system, and thus the code change may be fewer design issues; (ii) **Discussion activity** comprise the features of communication between developers and reviewers. In fact, classes having degraded symptoms can create more discussion among the reviewers [32, 136]. As well as, discussions with a high number of comments around code changes would find possible design symptoms, improving or maintaining the quality; and (iii) **Collaboration networks** consists of features of social networks. Previous studies [97, 98, 111] found that collaboration factors (i.e., level of participation within the system) could influence code review outcomes. For this reason, we constructed a network based on the collaboration of owners and reviewers to use the features proposed by [97].

4.3.6 Development of the Impactfulness Prediction Models

We experimented with six different (binary classification) supervised ML algorithms: Logistic Regression, Naive Bayes, SVM, Decision Tree, Random Forest, and Gradient Boosting. We chose to use these algorithms to classify whether a code change is impactful or not since it provides an intuitive and easy to explain classification model [78].

Training and testing the models. We trained and tested the models as follows. Firstly, we collected the design (un)impactful changes instances for a given system. We merged them into a single dataset, where design impactful changes instances are marked with a *true* value, and design unimpactful changes instances are marked with a *false* value. Secondly, before training the models, we dealt with imbalanced data, a common issue with software engineering data [103]. In our case, the number of design impactful changes instances varies; i.e., the design impactful changes instances might be greater than or smaller than the number of unimpactful ones. To that end, we relied on the under-sampling algorithm, which randomly selects instances of the oversampled class. Thirdly, we scaled all the features to a [0, 1] range to speed up the learning process of the algorithms [118]. Fourthly, we used the grid search to tune the hyperparameters of each model using five folds. Grid search is an exhaustive search that examines all of the combinations of a specified set of candidate settings to find the best combination [115].

Finally, to train the model, we employed a 10-fold cross-validation strategy using the hyperparameters established by the search [116]. This strategy randomly partitions the dataset into 10 folds of equal size, in which each fold has the same proportion of the various criticality classes. A single fold is then used as a test set, while the remaining ones are employed for training the model, i.e., they are independent of each other.

Performance evaluation. We evaluated the performance of each generated model, by analyzing confusion matrices, obtained from the testing strategy described above, and reporting the values of well-known measures [130]. *Precision* is the percentage of detected code changes that are actually impactful ($Pr = \frac{TP}{TP+FP}$). *Recall* is the percentage of correctly predicted impactful design change relative to all of the changes that are actually unimpactful ($Re = \frac{TP}{TP+FN}$). The *F1-score ($F1$)* is the harmonic mean of precision and recall. Additionally, to mitigate the limitation of choosing a fixed threshold when calculating precision and recall, we compute the *Area Under the ROC Curve (AUC)* values. AUC is computed by measuring the area under the curve that plots the true positive rate against the false positive rate, while varying the threshold that is used to determine if a design change is predicted as impactful or unimpactful.

Replication package. All data described previously, features used for training and testing the ML algorithms, hyperparameters analyzed, generated ML models, as well as the confusion matrix and statistical analysis are available in [58].

4.4 Results and Discussions

4.4.1 Design Impactful Changes vs. Unimpactful ones

To answer $\mathbf{RQ}_1$, we used the *Wilcoxon Rank Sum Test* [105] and the *Cliff's Delta (d)* measure [107] to verify which metrics are able to discriminate between impactful and unimpactful design changes. To this end, we explore each metric described in Section 4.3.5. The *Cliff's Delta (d)* measure [107] shows how strong is the difference between design impactful changes and unimpactful ones in terms of the analyzed metrics. Since we are performing multiple comparisons, we need to adjust the *p-values* to consider the increased chance of rejecting the null hypothesis simply due to random chance. To do so, we apply the widely used *Bonferroni correction* [106], which controls the familywise error rate. For this method, we consider that each system is a family, which means that we perform the correction in the *p-values* of the features at the system level.

Table 4.5 shows the results obtained for coarse- (CG) and fine-grained (FG) smells, where the 1st column contains the type of metric while the

2nd column shows the evaluated metric. The 3rd to 16th columns show the Wilcoxon Test and d results for each system, each group of two columns represents the results for CG and FG smells, respectively. Statistical significant differences (*p-value* < 0.05) are highlighted as gray cells. To interpret the *Cliff's Delta (d)* effect size, we employ a well-known classification [108], that defines four categories of magnitude, which are represented in Table 4.5: *negligible* (without symbol), *small* (*), *medium* (**), and *large* (***). The positive d magnitudes are represented by the (+) symbol and the negative ones are represented by the ($-$) symbol.

Table 4.5: Statistical Significance of the Wilcoxon Rank Sum Test and the Cliff's Delta (d) for Coarse and Fine-grained Smells

Dimension	Metric	spymemcached		java-client		jvm-core		platform.ui		jgit		egit		linuxtools		all	
		CG	FG	CG	FG	CG	FG	CG	FG	CG	FG	CG	FG	CG	FG	CG	FG
Developer's Experience	NC	(−)	(−)	(−)	(−)	(−)	(−)*	(−)	(−)	(−)	(−)	(−)	(−)*	(−)	(−)	(−)	(−)
	NRC	(+)	(−)	(−)	(−)	(−)*	(−)	(−)*	(−)*	(−)	(−)	(−)	(−)	(−)	(−)	(−)	(−)
	NDC	(+)*	(+)	(+)*	(+)*	(+)**	(+)**	(+)*	(+)*	(+)*	(+)*	(+)*	(+)*	(+)*	(+)*	(+)*	(+)*
	NR	(−)	(−)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(−)	(−)	(+)	(+)
	MR	(−)	(−)	(+)	(+)	(+)	(−)	(+)	(+)	(−)	(+)	(−)	(−)	(−)	(−)	(−)	(−)
	RMR	(−)*	(−)	(−)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(−)	(−)	(−)	(−)	(−)	(+)
	DMR	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)
Discussion Activity	NIC	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)
	NWIC	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)
	PWIC	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)
	NGC	(−)	(−)	(+)	(−)	(−)	(−)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)
	NWGC	(+)	(+)	(+)	(+)	(+)	(−)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)
	PWGC	(+)	(+)	(+)	(+)	(+)	(−)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)
	DL	(+)	(+)	(+)	(+)	(−)	(−)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)
Collaboration Networks	SD	(−)*	(+)	(−)	(−)	(−)	(−)	(−)	(−)	(−)	(−)	(−)	(−)	(−)	(−)	(−)	(−)
	SCLOS	(−)*	(+)	(−)	(−)	(−)	(−)	(−)	(−)	(−)	(−)	(−)	(−)	(−)	(−)	(−)	(−)
	SB	(−)	(−)	(−)	(−)	(+)	(+)	(−)	(−)	(−)	(−)	(−)	(−)	(−)	(−)	(−)	(−)
	SE	(−)*	(−)	(−)	(−)	(−)	(+)	(−)	(−)*	(−)	(−)	(−)	(−)	(−)	(−)	(−)	(−)
	SCLUST	(+)*	(+)	(+)	(−)	(−)	(−)	(+)	(+)	(−)	(−)	(−)	(−)	(+)	(+)	(+)	(+)
	SKC	(+)*	(+)	(−)	(−)	(−)	(−)	(−)	(−)	(+)	(+)	(+)	(+)	(+)	(−)	(+)	(−)
Size	NLA	(+)***	(+)***	(+)***	(+)***	(+)***	(+)***	(+)***	(+)***	(+)***	(+)***	(+)***	(+)***	(+)***	(+)***	(+)***	(+)***
	NLD	(+)*	(+)	(+)*	(+)**	(+)**	(+)***	(+)*	(+)*	(+)	(+)	(+)*	(+)*	(+)**	(+)**	(+)*	(+)*
	CHURN	(+)***	(+)***	(+)***	(+)***	(+)***	(+)***	(+)***	(+)***	(+)***	(+)***	(+)***	(+)***	(+)***	(+)***	(+)***	(+)***
	NFA	(+)***	(+)*	(+)***	(+)**	(+)***	(+)***	(+)***	(+)*	(+)***	(+)*	(+)***	(+)*	(+)***	(+)*	(+)***	(+)*
	NFD	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)	(+)
Diffusion	NFC	(+)**	(+)*	(+)**	(+)**	(+)***	(+)***	(+)*	(+)*	(+)*	(+)*	(+)**	(+)**	(+)**	(+)**	(+)**	(+)**
	ND	(+)***	(+)**	(+)***	(+)***	(+)***	(+)***	(+)***	(+)**	(+)**	(+)**	(+)***	(+)**	(+)***	(+)**	(+)***	(+)**
	ME	(+)***	(+)**	(+)***	(+)***	(+)***	(+)***	(+)***	(+)**	(+)**	(+)**	(+)***	(+)**	(+)***	(+)**	(+)***	(+)**
	NLANG	(+)	(+)	(+)*	(+)*	(+)*	(+)*	(+)*	(+)*	(+)*	(+)*	(+)*	(+)*	(+)*	(+)*	(+)*	(+)*
	NFT	(+)	(+)	(+)	(−)	(+)	(+)	(+)**	(+)*	(+)*	(+)	(+)***	(+)*	(+)**	(+)*	(+)**	(+)*
Complexity	NSA	(+)***	(+)***	(+)***	(+)***	(+)***	(+)***	(+)***	(+)**	(+)***	(+)***	(+)***	(+)***	(+)***	(+)***	(+)***	(+)***
	NSD	(+)	(+)	(+)*	(+)*	(+)*	(+)*	(+)*	(+)*	(+)	(+)	(+)*	(+)*	(+)*	(+)*	(+)*	(+)
	NSU	(+)*	(+)*	(+)*	(+)*	(+)**	(+)**	(+)*	(+)*	(+)	(+)*	(+)*	(+)*	(+)*	(+)*	(+)*	(+)*
File History	FD	(+)***	(+)**	(+)***	(+)***	(+)***	(+)***	(+)**	(+)**	(+)*	(+)*	(+)**	(+)**	(+)**	(+)**	(+)**	(+)**
	FM	(+)**	(+)*	(+)**	(+)**	(+)**	(+)**	(+)**	(+)**	(+)*	(+)*	(+)**	(+)*	(+)**	(+)**	(+)**	(+)*
Textual	ML	(−)	(+)	(+)*	(+)**	(+)	(+)*	(+)*	(+)*	(+)*	(+)**	(+)*	(+)*	(+)*	(+)*	(+)*	(+)*

We emphasize that **RQ**$_1$ does not consider the FEAT, IMPR, IMPR, DOC, and REFC metrics since the *Wilcoxon Test* and the *Cliff's Delta* measure are not suitable for boolean metrics.

Correlation between inline comments and impactful changes. Among all metrics in the dimension of discussion, we highlight those related to inline comments, *inline comments (NIC)*, *words in inline comments (NWIC)*, and *percentage of words in inline Comments (PWIC)*. For CG smells, they were statistically significant for most systems, except for the java-client (all three) and jvm-core (PWIC) systems. Such metrics also showed positive magnitude in all cases. This means that impactful changes, represented by both CG and FG smells, are often associated with a higher volume of inline comments. Nevertheless, the magnitude of these metrics was negligible in all cases. Thus, we conclude that discussion activity metrics in isolation are not enough for differentiating impactful changes.

Developer's experience dimension. The *directory changes (NDC)* metric presented consistent results in most systems both for CG e FG smells. *NDC* was the only metric in the developer's experience dimension that presented small and medium (and positives) magnitudes across all projects. This result means that *NDC* is the best metric in its dimension for differentiating impactful changes. **Collaboration networks dimension.** Results were similar on both CG e FG smells. Magnitudes were negative in 76% and 80% of cases also for both smells. Such a result indicates that prior collaboration between author and reviewers contributes to the production of changes that do not impact the design. However, in many cases, the results were not statistically significant.

Finding 9: The usefulness of the metrics from the discussion activity, developer's experience, and collaboration networks dimensions to differentiate design impactful changes is limited. However, the *number of directory changes* presents promising results.

Code metrics as strong indicators of impactful changes. The most relevant metrics for distinguishing between impactful and unimpactful changes were the ones related to code. Their results were not statistically significant in only three cases for FG smells. Moreover, with the exception of the *files deleted (NFD)* metric in the java-client system, all code metrics presented positive magnitudes. Among all code metrics, we highlight the *lines added (NLA)*, *changed lines (CHURN)*, and *segments added (NSA)* metrics, which presented statistically significant results with large magnitude in all cases, for both types of smells. If we restrict our analysis to CG smells, the *files added*

(NFA), *segments deleted (NSD)*, and *modify entropy (ME)* metrics also stand out. The highlighted metrics are closely related to the size (*NLA*, *CHURN*, *NFA*) and complexity (*NSA*, *NSD*, *ME*) of the changes. Thus, such metrics can be used by reviewers to decide when a change requires more attention to the design impact.

Textual dimension. The *message length (ML)* metric showed statistically significant results with small or medium positive magnitudes in most cases. We conjecture that this is because changes with detailed descriptions tend to be more complex, leading to a higher impact on design. **File history dimension.** The *file developers (FD)* and *file modifications (FM)* metrics, showed statistically significant results with positive magnitudes. In this case, the *FD* metric showed medium or large magnitudes in most cases. This result suggests that the more people interacting with a file set, the greater the chance that the next changes in such files will impact the design.

Finding 10: Code, textual, and file history dimensions contain metrics that are relevant to differentiate impactful changes. The most relevant ones can be combined to predict changes that require more attention to design.

4.4.2 ML Performance for Predicting Design Impactful Changes

Table 4.6: Performance of Using Different Learning Algorithms to Predict Design Impactful Changes

Coarse-grained Smells																								
System	**SVM (linear)**				**Decision Tree**				**Random Forest**				**Naive Bayes (gaussian)**				**Gradient Boosting**				**Logistic Regression**			
	Pr	**Re**	**F1**	**AUC**	**Pr**	**Re**	**F1**	**AUC**	**Pr**	**Re**	**F1**	**AUC**	**Pr**	**Re**	**F1**	**AUC**	**Pr**	**Re**	**F1**	**AUC**	**Pr**	**Re**	**F1**	**AUC**
spymemcached	0.77	0.55	0.64	0.69	0.92	0.93	0.92	0.92	0.96	0.93	0.94	0.95	0.84	0.28	0.42	0.61	0.94	0.94	0.94	0.94	0.80	0.62	0.69	0.73
java-client	0.80	0.73	0.76	0.77	0.96	0.95	0.96	0.96	0.97	0.97	0.97	0.97	0.88	0.41	0.56	0.68	0.97	0.95	0.96	0.96	0.80	0.74	0.77	0.78
jvm-core	0.85	0.76	0.80	0.81	0.95	0.96	0.95	0.95	0.97	0.96	0.96	0.96	0.68	0.92	0.78	0.73	0.96	0.96	0.96	0.96	0.85	0.80	0.82	0.83
platform.ui	0.78	0.51	0.62	0.68	0.90	0.93	0.92	0.92	0.94	0.95	0.94	0.94	0.59	0.88	0.67	0.61	0.93	0.95	0.94	0.94	0.71	0.65	0.68	0.69
egit	0.76	0.62	0.68	0.71	0.89	0.91	0.90	0.90	0.93	0.93	0.93	0.93	0.75	0.56	0.64	0.69	0.93	0.94	0.93	0.93	0.75	0.69	0.72	0.73
jgit	0.72	0.61	0.66	0.69	0.90	0.91	0.91	0.91	0.95	0.93	0.94	0.94	0.67	0.58	0.59	0.63	0.94	0.94	0.94	0.94	0.74	0.70	0.72	0.73
linuxtools	0.75	0.62	0.68	0.71	0.92	0.94	0.93	0.93	0.96	0.96	0.96	0.96	0.82	0.18	0.29	0.57	0.95	0.96	0.96	0.96	0.73	0.71	0.72	0.72
All	0.70	0.61	0.66	0.68	0.86	0.87	0.86	0.86	0.93	0.92	0.93	0.93	0.65	0.64	0.64	0.64	0.93	0.92	0.93	0.93	0.70	0.70	0.70	0.70
Average	**0.77**	**0.63**	**0.69**	**0.72**	**0.91**	**0.93**	**0.92**	**0.92**	**0.95**	**0.94**	**0.95**	**0.95**	**0.74**	**0.56**	**0.57**	**0.65**	**0.94**	**0.95**	**0.95**	**0.95**	**0.76**	**0.70**	**0.73**	**0.74**
Median	**0.77**	**0.62**	**0.67**	**0.70**	**0.91**	**0.93**	**0.92**	**0.92**	**0.96**	**0.94**	**0.94**	**0.95**	**0.72**	**0.57**	**0.62**	**0.64**	**0.94**	**0.95**	**0.94**	**0.94**	**0.75**	**0.70**	**0.72**	**0.73**
Fine-grained Smells																								
System	**SVM (linear)**				**Decision Tree**				**Random Forest**				**Naive Bayes (gaussian)**				**Gradient Boosting**				**Logistic Regression**			
	Pr	**Re**	**F1**	**AUC**	**Pr**	**Re**	**F1**	**AUC**	**Pr**	**Re**	**F1**	**AUC**	**Pr**	**Re**	**F1**	**AUC**	**Pr**	**Re**	**F1**	**AUC**	**Pr**	**Re**	**F1**	**AUC**
spymemcached	0.89	0.80	0.84	0.85	0.95	0.96	0.95	0.95	0.97	0.97	0.97	0.97	0.92	0.38	0.53	0.67	0.97	0.97	0.97	0.97	0.90	0.82	0.86	0.86
java-client	0.84	0.74	0.79	0.80	0.90	0.95	0.92	0.92	0.94	0.97	0.95	0.95	0.92	0.50	0.64	0.73	0.93	0.96	0.95	0.95	0.84	0.76	0.80	0.81
jvm-core	0.89	0.74	0.81	0.82	0.93	0.96	0.95	0.94	0.96	0.97	0.97	0.97	0.80	0.76	0.77	0.78	0.95	0.97	0.96	0.96	0.88	0.77	0.82	0.83
platform.ui	0.80	0.55	0.65	0.71	0.92	0.96	0.94	0.94	0.96	0.97	0.97	0.97	0.75	0.44	0.50	0.62	0.96	0.97	0.97	0.97	0.79	0.66	0.72	0.74
egit	0.85	0.72	0.78	0.80	0.91	0.95	0.93	0.93	0.95	0.96	0.96	0.96	0.84	0.62	0.71	0.75	0.95	0.97	0.96	0.96	0.84	0.76	0.80	0.81
jgit	0.90	0.62	0.73	0.77	0.91	0.93	0.92	0.92	0.96	0.95	0.95	0.95	0.91	0.36	0.52	0.66	0.96	0.95	0.96	0.96	0.86	0.71	0.78	0.80
linuxtools	0.82	0.65	0.72	0.75	0.92	0.95	0.93	0.93	0.95	0.97	0.96	0.96	0.90	0.23	0.36	0.60	0.95	0.97	0.96	0.96	0.81	0.72	0.76	0.77
All	0.74	0.67	0.70	0.72	0.88	0.90	0.89	0.89	0.95	0.94	0.95	0.95	0.68	0.67	0.67	0.67	0.95	0.94	0.94	0.94	0.75	0.70	0.72	0.73
Average	**0.84**	**0.69**	**0.75**	**0.78**	**0.92**	**0.95**	**0.93**	**0.93**	**0.96**	**0.96**	**0.96**	**0.96**	**0.84**	**0.50**	**0.59**	**0.69**	**0.95**	**0.96**	**0.96**	**0.96**	**0.83**	**0.74**	**0.78**	**0.79**
Median	**0.85**	**0.70**	**0.76**	**0.79**	**0.92**	**0.95**	**0.93**	**0.93**	**0.96**	**0.97**	**0.96**	**0.96**	**0.87**	**0.47**	**0.59**	**0.67**	**0.95**	**0.97**	**0.96**	**0.96**	**0.84**	**0.74**	**0.79**	**0.81**

We address **RQ**$_2$, by reporting and comparing different models after the 10 stratified cross-fold executions. We apply stratified sampling in all the cross-fold executions to make sure both training and test datasets contain the same amount of design (un)impactful changes instances. Table 4.6 shows the precision (Pr), recall (Re), F1-score (F1), and AUC values of each ML algorithm for each target system and smell levels. The row “all” represents the generated model, when training and testing in the entire dataset and using all features.

The performance of ML algorithms for predicting design impactful changes. Table 4.6 shows that for coarse-grained smells, the average of precision values across models ranges between 0.74 and 0.95, while the recall values range between 0.56 and 0.95. Similarly, the F1-score values range between 0.57 and 0.95, while the AUC values range between 0.65 and 0.95. A similar observation applies when we consider fine-grained smells, in which the average of precision, recall, F1-score, and AUC values ranges between 0.83 and 0.96, 0.50 and 0.96, 0.59, and 0.96, and 0.69, and 0.96, respectively.

To assess statistical differences between the models, we applied Friedman non-parametric test [149] with Nemenyi’s post hoc multiple pairwise comparison (p-value $\leq$0.05). We observed that both *Random Forest* and *Gradient Boosting* outperformed the other ML models at coarse-grained (CG) and fine-grained (FG) levels, with significant difference according to the statistical test. Furthermore, both *Random Forest* and *Gradient Boosting* present a similar performance, i.e., without a statistical difference, considering the CG level, with a similar average of 0.95 for both F1 and AUC. However, with a slight difference of 1% for the average of precision and recall values. A similar observation applies when at FG level. Moreover, to ascertain if the level of accuracy is adequate, we compared our model performance results with other approaches both for ML-based smell detection [147, 145] and code review analysis [148]. On average, our two best models achieved similar or better performance results than those previous work.

Finding 11: *Random Forest* and *Gradient Boosting* are the most accurate in predicting design impactful changes within code reviews. Both algorithms achieve an average of F1-score of 0.95 and 0.96, for predicting design impactful changes at a CG and FG level, respectively.

4.4.3 The Role of Social and Technical Features as Predictors

We address **RQ**$_3$, by investigating the performance of different feature sets as a proxy to predict design impactful changes at CG and FG smell levels, namely (i) social features only, (ii) technical features only, and (iii) social + technical features together. Additionally, and given the great number of features, we investigate the difference of the impactfulness prediction between to use or not a step for feature ranking and selection. Instead of simply removing the highly-correlated features by following a filtering method, we decided to apply a wrapper method to feature selection. We applied feature ranking with recursive feature elimination and cross-validated selection of the best number of features for our data. For feature ranking and selection we used the RFECV function available in the scikit-learn's feature selection package [129]. We run RFECV using 5-fold cross-validation and SVC linear as the estimator. After the cross-validation process, and RFECV recommendations, three new sets, namely feature selection sets, were generated, a set with 19, 9, and 40 features respectively for the social, technical, and social + technical dimensions. Similarly to **RQ**$_2$, we check the statistical difference of the results using Friedman test [149] and Nemenyi's post hoc multiple pairwise comparison, with a confidence level of 95%.

The effectiveness of social and technical features as predictors to design impactful changes. To evaluate the effectiveness of social and technical features, we rely on the best ML algorithms, i.e., *Random Forest* and *Gradient Boosting*, based on the **RQ**$_2$ results. Table 4.7 shows the mean values of precision (Pr), recall (Re), F1-score (F1), and AUC for both algorithms, grouped by dimension and feature set.

We observed that social features for impactful changes, at both levels of granularity, reached mean values of Pr, Re, F1, and AUC around 0.8. A similar performance is reached when feature selection was used. Technical features reach mean values of Pr, Re, F1, and AUC around 0.96 and 0.94 for impactful changes at, respectively, coarse- and fine-grained levels. But we also observed that the use of features selection at the coarse-grained level leads to a performance reduction, decreasing values of Pr, F1, and AUC. The Friedman test did not point a significant difference when using feature selection compared to all features, as well as between the two levels of granularity, in the same set of features. In summary, we can conclude that both sets of features are good predictors. They reach values of Pr, Re, F1, and AUC ≥ 0.79 at both levels.

Table 4.7: Performance of Social and Technical Features as Proxy to Predict Design Impactful Changes

Random Forest									
Dimension	**Feature set**	**Coarse-grained Smells**				**Fine-grained Smells**			
		Pr	**Re**	**F1**	**AUC**	**Pr**	**Re**	**F1**	**AUC**
Social	all features	0.80	0.81	0.81	0.80	0.79	0.79	0.79	0.79
	feature selection	0.80	0.81	0.81	0.80	0.79	0.79	0.79	0.79
Technical	all features	**0.96**	**0.96**	**0.96**	**0.96**	**0.94**	**0.94**	**0.94**	**0.94**
	feature selection	0.90	0.93	0.91	0.91	**0.94**	**0.94**	**0.94**	**0.94**
Social + technical	all features	0.95	0.94	0.95	0.95	0.93	0.92	0.93	0.93
	feature selection	0.87	0.91	0.89	0.88	0.93	0.92	0.93	0.93

Gradient Boosting									
Dimension	**Feature set**	**Coarse-grained Smells**				**Fine-grained Smells**			
		Pr	**Re**	**F1**	**AUC**	**Pr**	**Re**	**F1**	**AUC**
Social	all features	0.82	0.83	0.82	0.82	0.80	0.81	0.80	0.80
	feature selection	0.81	0.83	0.82	0.82	0.80	0.81	0.80	0.80
Technical	all features	**0.95**	**0.96**	**0.96**	**0.96**	**0.94**	**0.94**	**0.94**	**0.94**
	feature selection	0.90	0.92	0.91	0.91	**0.94**	**0.94**	**0.94**	**0.94**
Social + technical	all features	**0.95**	0.94	0.94	0.94	0.93	0.92	0.93	0.93
	feature selection	0.88	0.91	0.90	0.89	0.93	0.92	0.93	0.93

Finding 12: Both social and technical features are effective as a proxy to detect impactful changes. In this way, code review stakeholders may choose the set of features to be used according to their interests and roles.

Social features vs. technical features vs. social + technical features. The set of technical features are better predictors than the set of social features in terms of Pr, Re, F1, and AUC for coarse-grained and fine-grained smells, and both algorithms, with or without feature selection. We observe a significant statistical difference between both sets. Nevertheless, there is no statistical difference, in terms of performance, between technical and social + technical sets. The combination of both kinds of features leads to results that are statistically equivalent to the best results, obtained by the technical feature set. This happens even for the set of social and technical features together when the double number of features is used.

Finding 13: The use of technical features leads to the best results. Moreover, such kind of features can be used in combination with social features without reducing the performance of the ML algorithms.

4.4.4 The Best Features for Predicting Design Impactful Changes

We address $\mathbf{RQ}_4$, by reporting how often each feature appears among the top-1 and top-5 most important features of all the generated models without feature selection. To better understand the importance of social and technical features for predicting design impactful changes per symptom category, we generate different sets of rankings. To this end, we vary the configuration of each model according to the different feature sets, i.e., social features only, technical features only, and both together, and reported the five most frequent features per ranking and feature sets. We used scikit-learn's implementations to extract the feature importance of the SVM (linear), Decision Tree, Random Forest, Gradient Boosting, and Logistic Regression models [129]. We highlight that some models, e.g., SVM, might return the importance of a feature as a negative number, indicating that the feature is important for the prediction of the design unimpactful changes, in our case. Thus, we consider such a feature also important to the models, and thus, we build the ranking using the absolute value of feature importance returned by the models.

The best features using social and technical features in isolation for predicting design impactful changes. Table 4.8 lists the ranking of the best features across ML algorithms and systems grouped by smell category, i.e., *coarse-grained (CG)* and *fine-grained (FG) smells*, and feature set, i.e., *technical feature only*, and *social feature only*. For each ranking, we show the corresponding dimension (Dim.) and the frequency (Freq.) in which each feature appears by feature set.

Table 4.8: The Ranking of the Most Important Features Across ML Algorithms using Social and Technical Features in Isolation

Coarse-grained Smells						
Ranking	**Technical Features Only**			**Social Features Only**		
	Dim.	**Feature Name**	**Freq.**	**Dim.**	**Feature Name**	**Freq.**
1	Size	# Lines Added	13	Dev Exp	# Directory Changes	23
	Size	# Files Added	11	Disc Act	# Inline Comments	5
	Size	# Changed Lines	6	Dev Exp	# Changes	3
	Complexity	# Segments Added	2	Colab Net.	Social Closeness	2
	Diffusion	# Changed Files	2	Dev Exp	# Recent Changes	1
5	Size	# Lines Added	29	Dev Exp	# Directory Changes	31
	Size	# Files Added	28	Dev Exp	# Changes	25
	Size	# Changed Lines	23	Dev Exp	Merged Ratio	22
	Complexity	# Segments Added	20	Dev Exp	# Recent Changes	20
	Diffusion	Modify Entropy	13	Dev Exp	Recent Merged Ratio	11
Fine-grained Smells						
Ranking	**Technical Features Only**			**Social Features Only**		
	Dim.	**Feature Name**	**Freq.**	**Dim.**	**Feature Name**	**Freq.**
1	Size	# Lines Added	16	Dev Exp	# Directory Changes	28
	Size	# Changed Lines	10	Disc Act	# Inline Comments	4
	Complexity	# Segments Added	6	Disc Act	# Words in General Comments	1
	File History	# File Modifications	2	Disc Act	% Words in General Comments	1
	Diffusion	# Changed Files	1	Disc Act	Discussion Length	1
5	Size	# Lines Added	30	Dev Exp	# Directory Changes	33
	Size	# Changed Lines	23	Dev Exp	# Changes	22
	File History	# File Modifications	23	Dev Exp	# Recent Changes	22
	Textual	Message Length	17	Dev Exp	Merged Ratio	20
	Complexity	# Segments Added	16	Disc Act	# Inline Comments	11

By considering FG smells with technical features only, we observed that features quantifying the size of the code changes such as, *lines added (NLA)*, and *changed lines (CHURN)* frequently appear in the top-1 ranking, followed by features that quantify complexity (*segments added (NSA)*), history information about files modified (*file modifications (FM)*), and diffusion of a change (*changed files (NCF)*). A similar observation applies in the top-5 ranking, but with the presence of one textual feature, *message length (ML)*, that appears 17 times. Interestingly, when we compare both types of smells, we observed that *NLA*, *CHURN*, *NSA*, *NCF* also appear as important for both symptom categories in the top-1 and top-5 rankings, except for the features *files added (NFA)*, and *modify entropy (ME)* that only appear for CG smells.

On the other hand, when we consider social features only: for FG smells, we observed that the feature *directory changes (NDC)*, that quantifies the developer's experience in terms of the number of prior code changes submitted by the owner that contains at least one directory affected by the current submitted code change, is the most important social feature in the top-1 ranking appearing 28 times across models. Next, less frequently, features that quantify *inline comments (NIC)* made by reviewers on the code change submitted by the owner, and features that quantify discussion activities among the owner and reviewers, such as, *# words in general comments (NGC)*, *% words in general comments (PWGC)*, and *discussion length (DL)* also appear as important features.

Interestingly, by looking at the top-5 ranking, we observed that 4 out of 7 (57%) features that quantify different aspects of the developer's experience are considered as important across models, followed by the *NIC* that appears 11 times. By comparing both categories, we observed that *NDC* keeps its importance in the top-1 and top-5 rankings. Finally, two social features appear as important for CG smells, *social closeness (SCLOS)*, and *recent merged ratio (RMR)*. These observations also reinforce that senior developers should be allocated as reviewers to keep the quality of code review high [136, 40] and promote the knowledge transfer along revisions [14, 135].

Finding 14: In isolation, social features that quantify the developer's experience and discussion activities are indeed considered important across models in both smells categories. Also, as expected, technical features that quantify size, complexity, and diffusion of the code changes are considered important across all models.

The best features using social and technical features together for predicting design impactful changes. Table 4.9 also lists the ranking of the best features across ML algorithms grouped by CG and FG smells. However, in this table, we consider social and technical features as a single feature set. We also show the corresponding dimension (Dim.) and the frequency (Freq.) in which each feature appears.

Table 4.9: The Ranking of the Most Important Features Across ML Algorithms Using Social and Technical Features Together

Coarse-grained Smells						
Ranking	**Dim.**	**Technical features**	**Freq.**	**Dim.**	**Social features**	**Freq.**
1	Size	# Lines Added	12	-	-	-
	# Files Added	# Files Added	12	-	-	-
	Size	# Changed Lines	7	-	-	-
	Diffusion	# Changed Files	1	-	-	-
	Diffusion	Modified Entropy	1	-	-	-
5	Size	# Lines Added	29	Dev Exp	# Changes	2
	Size	# Files Added	29	Dev Exp	# Recent Change	2
	Size	# Changed Lines	21	Dev Exp	Recent Merged Ratio	2
	Complexity	# Segments Added	19	Dev Exp	# Directory Changes	1
	Diffusion	Modified Entropy	13	-	-	-
Fine-grained Smells						
Ranking	**Dim.**	**Technical features**	**Freq.**	**Dim.**	**Social features**	**Freq.**
1	Size	# Lines Added	12	-	-	
	Size	# Changed Lines	12	-	-	
	Complexity	# Segments Added	7	-	-	
	File History	# File Modifications	2	-	-	
	Diffusion	# Changed Files	1	-	-	
5	Size	# Lines Added	30	Dev Exp	# Changes	2
	Size	# Changed Lines	26	Dev Exp	# Recent Change	2
	Complexity	# Segments Added	18	Dev Exp	Recent Merged Ratio	2
	Diffusion	# Languages	17	Disc Act	% Words in General Comments	1
	File History	# File Modifications	13	Dev Exp	# Directory Changes	1

For FG smells with social and technical features aggregated, we observe that the same technical features listed in Table 4.8 appears in the top-1 and top-5 rankings, i.e., when we consider the technical features in isolation, the same features also appear as the most important features for FG smells, except for the *message length (ML)*, that only appears when the technical features are in isolation, and the *number of languages (NLANG)* feature, which only appears in the rankings with the combined features. Both of those features appear 17 times each in their respective top-5 rankings. We also observed that when the social and technical features are considered together, social features only appear in the top-5 ranking.

This result indicates that technical features, when combined with social ones, tend to be the most important features across models, especially in the top-1 ranking. However, social features that appear in the top-5 ranking are majority features that quantify the developer's experience. This result, reinforces our previous finding, on the importance of the developer's experience to predicting fine-grained design impactful changes.

We also observed similar behavior for CG smells, in which the same set of technical features when we considered the technical features in isolation, also appear as the most important features across models. The exception to this is the feature *modified entropy (ME)*, which appears 1 time in the top-1 ranking in Table 4.9. Finally, similar to FG smells the social features only appear in the top-5 ranking, with the prevalence of features that quantify the developer's experience.

Finding 15: Technical features tend to be considered the most important features across models when compared with social features. However, social features that quantify the developer's experience are also considered important across models in both symptom categories.

Some features never appear in any of the rankings. Considering the full rankings of features importance, we observed that for both FG and CG smells, when technical and social features are considered in isolation, the technical features that capture textual characteristics of the commit message such as, *has bug (BUG)*, *has feature (FEAT)*, *has improvement (IMPR)*, *has document (DOC)*, and *has refactor (REFC)* do not appear even when we consider the top-5 ranking. A similar observation applies when we consider all social and technical features together, except for the feature *FEAT*. On the social features, for both smell types the features *directory merged ratio (DMR)* and *social betweenness (SB)* never appear in the top-1 and top-5 feature importance ranking. Additionally, the feature *percentage words in general comments (PWGC)* also does not appear in the rankings for CG smells.

On the other hand, when we combine all social and technical features together, eight social features do not appear among the rankings for both symptoms category. These features, four are from the collaboration network dimension, *social closeness (SCLOS)*, *social clustering (SCLUST)*, *social eigenvector*, and *social betweenness (SB)*. Another group of three features that do not appear are from the discussion activity dimension: *inline comment (NIC)*, *general comments (NGC)*, and *discussion length (DL)*. Finally, *reviews (NR)* from the developer's experience dimension also does not appear. Specifically for FG smells, we observed that *merged ratio (MR)*, *recent merged ratio (RMR)*, *directory merged ratio (DMR)*, *social k coreness (SKC)*, *# words in inline comments (NWIC)*, *# words in general comments (NWGC)*, and *% words in general comments (PWGC)* also never appear in the rankings.

Finding 16: Features from the textual dimension consistently did not appear in any of the rankings when technical features are used in isolation. Conversely, when both types of features are used in conjunction, features related to collaboration networks and discussion activity tend to not appear for both symptom categories.

4.5 Threats to Validity

Construct and Internal Validity. The precision and recall of degradation symptoms detection may have influenced our results. We mitigate this threat by selecting a detection tool, successfully used in recent studies on design degradation [4, 52, 136, 32], and previous work [114] indicated a precision of 96% and a recall of 99%. Moreover, there is evidence that developers tend to refactor code elements with a high density and diversity of the selected symptoms [29]. Although, there are more instances of design unimpactful changes in our dataset we mitigate this imbalanced dataset by removing instances from the over-represented class through random under-sampling strategy. Moreover, the selection of the ML algorithms and their parameter settings may affect the accuracy and influence interpretability. We mitigate this threat by selecting the most widely used interpretable ML algorithms and, for a fair comparison, we searched for their best parameters through an extensive hyperparameter search via grid search, and 10-fold cross-validation strategy. Finally, we selected and computed a wide number of social and technical features that helped us measure different social and technical dimensions of the changes involved in each code revision, e.g., developer experience, and file history. The rationales for using metrics are supported by prior studies, e.g., [5, 97, 110, 111]. We wrote scripts to automate compute these metrics, and all implementations were validated by three paper authors.

Conclusion and External Validity. About the descriptive analysis, four paper authors contributed to the analysis of design (un)impactful changes. For the statistical analysis, we rely on the *Wilcoxon Rank Sum Test*, *Bonferroni correction*, and *Cliff's Delta (d)* measure to verify which metrics are able to discriminate between (un)impactful design changes. We also computed largely used performance measures [130], precision, recall, F1-score, and AUC score. Furthermore, we rely on the Friedman non-parametric test, and Nemenyi-tests to avoid subjective opinion regarding the best accurate model and the role of social and technical features as predictors.

About the feature importance, our ML pipeline does not currently have a way to get the feature importance of Naive Bayes, but we have no reason to believe the lack of this model affected the conclusion of RQ_4. We are aware that there is an inherent threat to transferring our findings to other systems, i.e., our results may be subject to the degradation characteristics of these specific systems. However, we focused on seven systems to be able to make robust and reliable statements about if learning approaches can be used in such settings.

4.6 Final Remarks

In summary, our main findings pointed out that: (i) both social and technical features are able to distinguish between design impactful changes and unimpactful ones; (ii) Random Forest and Gradient Boosting are the most accurate algorithms; and (iii) both social and technical features are effective as a proxy to predict impactful changes. Our findings also provide insights for new studies and be the basis for tool builders creating a new generation of tools to aid developers in automatically predicting design impactful changes during code reviews. We also show that: (i) existing detection tools should be more interactive, in a stepwise manner, to anticipate, find, and remove signs of degradation before finishing a review; (ii) In addition to only technical features, the combination with social features is promising for predicting design (un)impactful changes; and (iii) qualitative studies should be performed to explain other aspects governing the decision-making process discriminating and predicting design impactful changes.

4.7 Summary of Chapter 4

In order to address our **second research problem** (see Section 1.2), this chapter presented a study to investigate the role of social aspects in distinguishing and predicting (un)impactful design changes. To this end, we performed a large-scale investigation by analyzing 57,498 reviewed code changes from seven open-source systems. We reported an investigation on the prediction of design impactful changes in modern code review using technical and social aspects.

The study was based on the extraction and assessment of 41 different metrics and corresponding features associated with social and technical aspects. We identified which metrics are able to distinguish between design impactful changes and unimpactful ones (Section 4.4.1). We also evaluated the use of six ML algorithms to predict design impactful changes (Section 4.4.2). Next, we investigated the performance of different feature sets (social features only, technical features only, and social and technical features together) as a proxy to predict design impactful changes (Section 4.4.3). Finally, we investigate what features are the best indicators of impactful design changes (Section 4.4.4).

In the next chapter, we revisit the main doctoral thesis contributions and present implications, new challenges and opportunities for improvement. We also present possible future work that we identified as needed emerged along the studies conducted in the context of this Doctoral thesis.

5
Conclusion and Future Work

As mentioned in Chapter 1, the software design is a key concern in code review through which developers actively discuss and improve each code change. Additionally, code review is predominantly a cooperative task influenced by both technical and social aspects. Consequently, these aspects can play a key role in how software design degrades as well as contributing to accelerating or reversing the degradation during the process of each single code change's review. However, we had limited knowledge about the real impact of code reviews and their practices on design degradation over time. For instance, we did not know how the process of design degradation evolution is impacted: (i) within each single review, and (ii) across multiple reviews.

Consequently, one can not understand how certain code review practices – including review intensity, developer participation, and the review duration – consistently reduce or further increase degradation as the software project evolves (Research Problem 1). We also did not know to what extent social and technical aspects involved during the code review process are related to either the reduction or the increase of design degradation. Additionally, there is no knowledge whether the social aspects represented as features in ML techniques can contribute to effective predictions of design impactful changes combining social and technical features for predicting design (un)impactful changes (Research Problem 2).

Therefore, the lack of knowledge about the topics aforementioned led us to understand how to support design-driven code reviews and how technical and how social aspects can be positively explored to better support the code review process. In this context, the goal of this thesis was to step-wisely understand how to support design-driven code review and how contextual information – technical and social – can be addressed to support the code review process. To achieve such a goal, we conducted two large empirical studies, which are summarized as follows.

5.1 Revisiting the Thesis Contributions

In our quest to unveiling the social and technical facets of design degradation in modern code review, our first step was to investigate and provide empirical evidence of the impact of modern code review and their practices on design degradation. Thus, we conducted a retrospective study (Chapter 3) to answer the following research question presented in Section 1.3: *To what extent do modern code review impact the design degradation evolution?*

To answer this research question we analyzed 14,971 code reviews from seven systems of two large open source communities. In this analysis, we explored the evolution of two degradation characteristics: density and diversity of symptoms. We analyzed such characteristics in the context of two categories of degradation symptoms: fine-grained and coarse-grained smells [30]. Then, we analyzed the impact on design degradation even in the presence of explicit intent of improving the design. This analysis also includes the presence of explicit design discussions along with the revisions of a review.

Finally, we investigated how degradation characteristics evolve along with the revisions that occur along each code review. To this end, we identified and investigated four different evolution patterns for degradation characteristics (i.e., density and diversity). As a result of this analysis, we provided new insights on the design degradation evolution along the reviewing process. In addition, we observed that certain code review practices can be used as indicators of increased design degradation. In this context, the first contribution of this thesis is:

First Contribution. Empirical evidence that characterizes how the process of design degradation evolves within each review and across multiple reviews. This knowledge includes the influence of certain code review practices on combating or even accelerating design degradation.

Our results indicate that in most cases that code reviews had little or no impact on software design degradation. Moreover, by analyzing different design degradation pattern evolution during the revisions of each single review, we found the existence of a wide fluctuation of design degradation. This fluctuation means that developers are both introducing and removing symptoms along a single code review. However, at the end of the review, such fluctuations often result in the amplification of design degradation even in the context of reviews with an explicit design concern. This result led us to investigate a mechanism to better support developers on the prevention of design problems during code review.

In this follow-up study (Chapter 4), we aimed to answer the following research question: *To what extent do social aspects contribute to distinguishing and predicting design impactful changes in a modern code review?* To answer this research question, we reported an investigation on the prediction of design impactful changes in modern code review. We extracted and assessed 41 different features based on both social and technical aspects of the changes involved in each revision of a code review. Based on different features set, we evaluated the use of interpretable Machine Learning (ML) algorithms to predict design impactful changes. Finally, we evaluated the predictive power of the selected features with those algorithms to assist developers on determining whether a code change is impactful.

As a result of this investigation, we observed that *Random Forest* and *Gradient Boosting* are the best algorithms. We also observed that the use of technical features results in more precise predictions. However, the use of social features alone, which are available even before the code review starts (e.g., for team managers or change assigners), also leads to highly-accurate prediction. Therefore social and/or technical prediction models can be used to support further design inspection of suspicious changes early in a code review process. Thus, the second contribution of this thesis is:

Second Contribution. A seminal report on the effect of certain technical and socials aspects on distinguishing and predicting design impactful changes in code review

In addition to the previous contributions, this doctoral thesis provides several insights about the use of different metrics that can be used to quantify both social and technical aspects that emerge during code reviews. In other words, we provide a set of metrics/features that can be by other researchers to reveal and understand design degradation that affects the quality of design during code reviews. Thus, our third contribution of this thesis was:

Third Contribution: Set of social and technical aspects that emerge during code reviews. We provide a catalog with more than 30 metrics in Chapters 3 and 4 that can be used by other researchers to investigate different phenomena that affect the quality of design during code reviews.

Finally, as mentioned in Chapter 1 an essential and good practice of any scientific research is that studies must be replicable. Thus, in this doctoral thesis, we provided the full replication packages of each study that we have performed. The replication packages were available on Zenodo (https://zenodo.org/), an online repository hosted at CERN which allows sharing publications and supporting data (see Table 1.2). Therefore, in this doctoral thesis we provide enriched datasets that allow researchers to investigate the context behind design degradation during the code review process. For each dataset, we make available all data collected, together with the definition of metrics, features, and statistical tests, and scripts.

Fourth Contribution: A novel datasets that allow researchers to investigate the context behind design degradation during the code review process.

In summary, we highlight that this research contains four main novel contributions, which map onto the research questions and are related to the artifacts generated along our research: (i) an empirical characterization of modern code review impact on design degradation; (ii) A seminal report on the effect of certain technical and socials aspects on distinguishing and predicting design impactful changes in code review; (iii) a catalog about the effect of technical and socials aspects on design degradation during code reviews; and (iv) A novel datasets that allow researchers to investigate the context behind design degradation during the code review process. The latter main contribution is crosscutting in Chapters 3 and 4. Therefore, the contributions listed here can advance both state-of-the-art and state-of-the-practice. Researchers and industry practitioners can use the discussions in this thesis to define mechanisms that drive news solutions for supporting developers during code reviews. Finally, The knowledge provided here can help researchers in building more suitable mechanisms that are aligned with how developers perform code reviews in practice.

5.2 Thesis Delimitations

This doctoral thesis focused on the analysis of social aspects and their influence on design degradation. Given the wide scope of social aspects, they have been investigated in many disciplines, including social sciences and psychology. Thus, we need to describe what are self-imposed boundaries set by the context of the field of software engineering studies, which also applies to this doctoral research.

Delimitation 1: *The measurement of social aspects during code reviews using only social metrics.* To gain the perspectives of social aspects concerning design degradation during code reviews, in this doctoral thesis, we measure social aspects based on the computation of metrics collected from software repositories only. Such metrics were identified in previous studies [7, 32, 97, 98, 99, 111] in the software engineering field or were defined during this doctoral thesis scope. Thus, the use of metrics only to quantify social aspects in distributed software development does not allow the researcher to gain other unmeasurable views of those social aspects through the code review process.

Delimitation 2: *A limited view of software engineering about social aspects.* The observations obtained in the scope of this doctoral thesis are strictly related to a narrow subset of social aspects, and their definitions are constrained here, take into consideration broader definitions of these aspects in the social sciences and psychology. Thus, if we have considered the view of professionals in the social sciences and psychology other aspects or relations could be discovered and explored in this doctoral thesis. However, we decided to focus on measurable elements of collaborative software development that can be realistically obtained from software repositories. The reason is that prediction models must remain simple and applicable in distributed software development. In this case, the communication and cooperation of software developers and reviewers are only based on discussions in online development platforms.

5.3 Thesis Implications

This doctoral thesis provides several findings which lead to implications for researchers, tool developers, and practitioners. Those main implications are discussed as follows.

Implication 1: *Developers should be aware of code review practices that can lead to design degradation.* During code reviews developers can modify the source code with different proposes, such as, adding a new feature or improving the quality of source code. As reported in Chapter 3, exist a wide range of code review practices may influence the increase or decrease the risk of design degradation. Our results show that practices such as long discussions and a high rate of review disagreement are often associated with a higher risk of software degradation. Conversely, reviews with active engagements of multiple reviewers tend to exert a risk-decreasing effect on design degradation. Therefore, the code review stakeholders should give additional attention to certain code review practices and their effect on design degradation.

Implication 2: *The developers need to promote good design discussions during code reviews.* In complement to the previous implication, we also observed that reviews with explicit intents of design improvement tend to reduce or avoid design degradation. However, the sole presence of design discussions is not enough for avoiding design degradation. Such results indicate that monitoring mechanisms could be applied to detect the presence of design discussions. In ongoing design discussions that are not reflecting on a decrease of design degradation, developers and reviewers should be warned. Moreover, reviewers and developers, need to double-check the occurrence of constructive comments, providing actionable suggestions to mitigate the issues, without neglecting the focus on design improvement.

Implication 3: *The combination of social and technical features is promising for predicting design (un)impactful changes.* In Chapter 4, we show that the existing tools for detecting design smells tend to ignore the presence and influence of social aspects on the prediction of design (un)impactful changes. On one hand, our results show that indeed the use of technical features results in precise predictions. On the other hand, the use of social features alone, which are available even before the code review starts (e.g., for team managers or change assigners), also leads to highly-accurate predictions. Therefore, social and/or technical prediction models can be used to guide further design-driven reviewers of suspicious changes early in a code review process.

Implication 4: *Existing detection tools should be more interactive within code reviews.* The findings reported in Chapters 3 and 4 suggest that there is a need for tools that support code reviewers in a stepwise manner within a single code review. In fact, existing code review tools, such as Gerrit, lack mechanisms that enable the reviewers and code authors in along the revisions of a code review, to anticipate, find and remove signs of degradation before conducting a review and submitting a change request. These mechanisms could rely on their current diff visualization to alert developers about the existence of degradation symptoms before the start of the next code revision. The alerts could be explained in terms of technical and social metrics that exceed previously tailored or automatically computed outliers.

5.4 Future Work

New challenges and opportunities have emerged along the studies conducted in the context of this thesis. Based on them, we describe further directions for future work according to two perspectives: (i) strictly related to this doctoral thesis and its delimitations, and (ii) beyond the delimitations presented in Section 5.2.

First, We present below the recommendations of future work strictly related to this doctoral thesis and its delimitations as follows.

Future work 1: *Automatic classification of a design discussion in code reviews.* In Chapter 3, we manually analyzed data from code review (commit messages, discussions between developers, and the source code) to classify a code review as *design-related* or *design-unrelated*. We identified the intent of developers in improving the structural design of their system. Although we have following a rigorous method to perform the manual analysis, this method does not scale either for a study of larger proportions or for piratical settings in software projects. Hence, the automatic classification of the code review in *design-related* or *design-unrelated* would greatly enhance empirical studies and tool support such as those discussed in this thesis.

Future work 2: *Evaluate the effect of code reviews on other types of degradation symptoms, and different characteristics of design degradation.* In Chapters 3 and 4 we used two types of degradation symptoms (fine-grained smells and coarse-grained smells) as a proxy to measure the structural design degradation. Additionally, we used the density and diversity of symptoms as characteristics of design degradation. However, other types and characteristics of degradation symptoms have been proposed in the literature, such as architectural smells or internal quality attributes. Hence, future studies could build upon the methodology and data we made available in this thesis to extend such studies by incorporating different degradation symptoms.

Future work 3: *Expand the social and technical aspects and dimensions to understand their role on software degradation.* In Chapters 3 and 4, we have used different metrics and features grouped into different dimensions to investigate how different aspects influence the design degradation in code reviews. Even though we have used a large set of metrics/features grouped into different dimensions, further studies can extend it or use a quite different set in order to identify different correlations that can reveal other possible influences on design software degradation.

Future work 4: *Qualitative studies should be performed to explain aspects governing the decision-making process on the discrimination and prediction of design impactful changes.* Both studies presented in Chapters 3 and 4 are quantitative studies. Thus, our results and implications are based on quantitative data collected from open source repositories. Hence, future studies may use the insights from this doctoral research to design qualitative studies with developers to better explain the influence of aspects governing the decision on design impactful changes.

Future work 5: *Integration of our prediction model into a code review platform.* In Chapter 4, we proposed and investigated a prediction model based on the use of social and technical aspects to make developers aware about the design impact of their code changes during the code review process. In Chapter 4, we present a set of features that are able to distinguish (un)impactful design changes, including the best indicators and best combinations of feature sets. All this knowledge could be used to integrating a prediction model in a code review platform, such as Gerrit and GitHub to be used by practitioners during the code review process. The integration should take into consideration our findings to determine when and how developers should be warned.

We presented the suggestion of future work beyond the thesis delimitations as follows.

Future work 6: *Exploring the coordination and collaboration aspects involved during the code review process.* In Chapters 3 and 4, we explore different types of metrics that help us to quantify social aspects. Despite we have grouped these metrics into different dimensions to capture the social aspects, another approach could be explored in future work. For instance, social aspects could be defined and structured in terms of the 3C Model of collaboration [171] or the Teamwork Quality model [172]. The 3C model is based on the conception that to collaborate, the members of a group need to communicate, coordinate, and cooperate [171]. Therefore, future studies could use the 3C model as the basis for modeling and developing coordination and collaboration strategies during the code review process. Since we do not cover all aspects and dimensions of the 3C model.

Future work 7: *Exploring the reason behind the disagreement between reviewers.* In Chapter 3, we have observed that a high disagreement among reviewers and long discussions have a risk-increasing effect on software design degradation. However, these results were obtained using statistical analysis and a lightweight manual validation only (available in our replication packages). Hence, future studies should perform a more in-depth qualitative analysis concerning the different scenarios of disagreement among reviewers. For instance, future studies can use the reviews tagged as a high disagreement and observe forms of interactions among reviewers. One could analyze if these disagreements and interaction forms are related to particular degraded code elements.

Future work 8: *Conducting empirical studies using psychometric instruments to measure the social aspects during code review activities.* Although the software development activities, including code reviews activities, is an inherently human activity, research in software engineering has long focused on building on processes and tools, without taking into consideration the human factors behind the development activities. Thus, given that we have investigated the social aspects (developer experience, discussion activity, and collaboration networks) via the computation of software repository metrics, future studies could use other approaches, such as psychometric instruments. Such instruments are widely used in social sciences and psychology [174, 175, 176]. Therefore, future studies could explore the proper use of the knowledge on these disciplines to better understand social aspects that emerge during code reviews and how they establish a cause-effect relationship with design degradation.

Future work 9: *The dashboard proposition to help development team managers.* One of the potential outputs of this thesis is the proposition of a dashboard to helps managers to keep attention or monitor the health of their team concerning the social aspects during the software development activities [185]. For instance, future work can build a dashboard that shows the set of social aspects (represented by metrics) investigated in this thesis. Thus, managers can monitor, for instance, whether there is high team participation in review activities, or whether there are design discussions or not. Such information can give insights to managers about how to improve their team in different contexts or thinking about strategies to minimize certain practices or aspects that may influence the quality of systems.

Bibliography

[1] DORMANN, C. F.; ELITH, J.; BACHER, S.; BUCHMANN, C.; CARL, G.; CARRÉ, G.; MARQUÉZ, J. R. G.; GRUBER, B.; LAFOURCADE, B.; LEITÃO, P. J. ; OTHERS. **Collinearity: a review of methods to deal with it and a simulation study evaluating their performance**. Ecography, 36(1):27–46, 2013.

[2] WOHLIN, C.; RUNESON, P.; HÖST, M.; OHLSSON, M.; REGNELL, B. ; WESSLÉN, A.. **Experimentation in Software Engineering**. Springer Science & Business Media, 1st edition, 2012.

[3] PAIXAO, M.; KRINKE, J.; HAN, D. ; HARMAN, M.. **Crop: Linking code reviews to source code changes**. In: PROCEEDINGS OF THE 15TH INTERNATIONAL CONFERENCE ON MINING SOFTWARE REPOSITORIES (MSR), p. 46–49, 2018.

[4] SHARMA, T.; MISHRA, P. ; TIWARI, R.. **Designite: a software design quality assessment tool**. In: PROCEEDINGS OF THE 1ST INTERNATIONAL WORKSHOP ON BRINGING ARCHITECTURAL DESIGN THINKING INTO DEVELOPERS' DAILY ACTIVITIES (BRIDGE), p. 1–4. ACM, 2016.

[5] WEISSGERBER, P.; NEU, D. ; DIEHL, S.. **Small patches get in!** In: PROCEEDINGS OF THE 5TH INTERNATIONAL CONFERENCE ON MINING SOFTWARE REPOSITORIES (MSR), p. 67–76. ACM, 2008.

[6] TSAY, J.; DABBISH, L. ; HERBSLEB, J.. **Influence of social and technical factors for evaluating contribution in github**. In: PROCEEDINGS OF THE 36TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE), p. 356–366. ACM, 2014.

[7] JIANG, Y.; ADAMS, B. ; GERMAN, D. M.. **Will my patch make it? and how fast?: Case study on the linux kernel**. In: PROCEEDINGS OF THE 10TH INTERNATIONAL CONFERENCE ON MINING SOFTWARE REPOSITORIES (MSR), p. 101–110. IEEE Press, 2013.

[8] THONGTANUNAM, P.; MCINTOSH, S.; HASSAN, A. E. ; IIDA, H.. **Investigating code review practices in defective files: An empir-**

ical study of the qt system. In: PROCEEDINGS OF THE 12TH INTERNATIONAL CONFERENCE ON MINING SOFTWARE REPOSITORIES (MSR), p. 168–179. IEEE Press, 2015.

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

[11] MCINTOSH, S.; KAMEI, Y.; ADAMS, B. ; HASSAN, A. E.. **An empirical study of the impact of modern code review practices on software quality**. Emp. Softw. Eng. (ESE), 21(5):2146–2189, 2016.

[13] BELLER, M.; BACCHELLI, A.; ZAIDMAN, A. ; JUERGENS, E.. **Modern code reviews in open-source projects: Which problems do they fix?** In: PROCEEDINGS OF THE 11TH INTERNATIONAL CONFERENCE ON MINING SOFTWARE REPOSITORIES (MSR), p. 202–211, 2014.

[14] BACCHELLI, A.; BIRD, C.. **Expectations, outcomes, and challenges of modern code review**. In: PROCEEDINGS OF THE 35TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE), p. 712–721, 2013.

[15] PAIXAO, M.; KRINKE, J.; HAN, D.; RAGKHITWETSAGUL, C. ; HARMAN, M.. **The impact of code review on architectural changes**. IEEE Trans. Softw. Eng. (TSE), p. 1–19, 2019.

[16] RIGBY, P. C.; BIRD, C.. **Convergent contemporary software peer review practices**. In: PROCEEDINGS OF THE 9TH ACM JOINT EUROPEAN SOFTWARE ENGINEERING CONFERENCE AND SYMPOSIUM ON THE FOUNDATIONS OF SOFTWARE ENGINEERING (FSE), p. 202–212, 2013.

[17] RIGBY, P. C.. **Open source peer review–lessons and recommendations for closed source**. IEEE Software, 2012.

[18] BESKER, T.; MARTINI, A. ; BOSCH, J.. **Time to pay up: Technical debt from a software quality perspective**. In: PROCEEDINGS OF THE XX IBEROAMERICAN CONFERENCE ON SOFTWARE ENGINEERING (CIBSE), p. 235–248, 2017.

[19] MARTIN, R. C.. **Agile software development: principles, patterns, and practices**. Prentice Hall, 2002.

[20] BRUNET, J.; MURPHY, G. C.; TERRA, R.; FIGUEIREDO, J. ; SEREY, D.. **Do developers discuss design?** In: PROCEEDINGS OF THE 11TH

INTERNATIONAL CONFERENCE ON MINING SOFTWARE REPOSITORIES (MSR), p. 340–343. ACM, 2014.

[21] ZANATY, F. E.; HIRAO, T.; MCINTOSH, S.; IHARA, A. ; MATSUMOTO, K.. **An empirical study of design discussions in code review.** In: PROCEEDINGS OF THE 12TH INTERNATIONAL SYMPOSIUM ON EMPIRICAL SOFTWARE ENGINEERING AND MEASUREMENT (ESEM), p. 11. ACM, 2018.

[22] PALOMBA, F.; BAVOTA, G.; DI PENTA, M.; OLIVETO, R. ; DE LUCIA, A.. **Do they really smell bad? a study on developers' perception of bad code smells.** In: PROCEEDINGS OF THE 30TH INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE AND EVOLUTION (ICSME), p. 101–110. IEEE, 2014.

[22] KONONENKO, O.; BAYSAL, O. ; GODFREY, M. W.. **Code review quality: how developers see it.** In: PROCEEDINGS OF THE 38TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE), p. 1028–1038. IEEE, 2016.

[23] MÄNTYLÄ, M. V.; LASSENIUS, C.. **What types of defects are really discovered in code reviews?** IEEE Trans. Softw. Eng. (TSE), 35(3):430–448, 2008.

[24] MORALES, R.; MCINTOSH, S. ; KHOMH, F.. **Do code review practices impact design quality? a case study of the qt, vtk, and itk projects.** In: PROCEEDINGS OF THE 22ND INTERNATIONAL CONFERENCE ON SOFTWARE ANALYSIS, EVOLUTION, AND REENGINEERING (SANER), p. 171–180. IEEE, 2015.

[25] TRIFU, A.; MARINESCU, R.. **Diagnosing design problems in object oriented systems.** In: PROCEEDINGS OF THE 12TH WORKING CONFERENCE ON REVERSE ENGINEERING (WCRE), p. 10–pp, 2005.

[26] FEITELSON, D.; FRACHTENBERG, E. ; BECK, K.. **Development and deployment at facebook.** IEEE Internet Comput., 17(4):8–17, 2013.

[28] EPOSHI, A.; OIZUMI, W.; GARCIA, A.; SOUSA, L.; OLIVEIRA, R. ; OLIVEIRA, A.. **Removal of design problems through refactorings: are we looking at the right symptoms?** In: PROCEEDINGS OF THE 27TH INTERNATIONAL CONFERENCE ON PROGRAM COMPREHENSION (ICPC), p. 148–153. IEEE Press, 2019.

[29] OIZUMI, W.; SOUSA, L.; OLIVEIRA, A.; CARVALHO, L.; GARCIA, A.; COLANZI, T. ; OLIVEIRA, R.. **On the density and diversity of degradation symptoms in refactored classes: A multi-case study**. In: PROCEEDINGS OF THE 30TH INTERNATIONAL SYMPOSIUM ON SOFTWARE RELIABILITY ENGINEERING (ISSRE), 2019.

[30] SHARMA, T.; SPINELLIS, D.. **A survey on software smells**. J. Syst. Softw. (JSS), 138:158–173, 2018.

[32] BARBOSA, C.; UCHÔA, A.; FALCAO, F.; COUTINHO, D.; BRITO, H.; AMARAL, G.; GARCIA, A.; FONSECA, B.; RIBEIRO, M.; SOARES, V. ; SOUSA, L.. **Revealing the social aspects of design decay: A retrospective study of pull requests**. In: PROCEEDINGS OF THE 34TH BRAZILIAN SYMPOSIUM ON SOFTWARE ENGINEERING (SBES), p. 1 – 10, 2020.

[33] FERNANDES, E.; UCHÔA, A.; BIBIANO, A. C. ; GARCIA, A.. **On the alternatives for composing batch refactoring**. In: PROCEEDINGS OF THE 3RD INTERNATIONAL WORKSHOP ON SOFTWARE REFACTORING (IWOR), p. 9–12, 2019.

[34] SOUSA, L.; OLIVEIRA, A.; OIZUMI, W.; BARBOSA, S.; GARCIA, A.; LEE, J.; KALINOWSKI, M.; DE MELLO, R.; FONSECA, B.; OLIVEIRA, R. ; OTHERS. **Identifying design problems in the source code: A grounded theory**. In: PROCEEDINGS OF THE 40TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE), p. 921–931, 2018.

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

[36] OIZUMI, W.; GARCIA, A.; SOUSA, L.; CAFEO, B. ; ZHAO, Y.. **Code anomalies flock together: Exploring code anomaly agglomerations for locating design problems**. In: PROCEEDINGS OF THE 38TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE), 2016.

[37] YAMASHITA, A.; ZANONI, M.; FONTANA, F. A. ; WALTER, B.. **Inter-smell relations in industrial and open source systems: A replication and comparative analysis**. In: PROCEEDINGS OF THE 31ST INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE AND EVOLUTION (ICSME), p. 121–130. IEEE, 2015.

[39] SOUSA, L.; OLIVEIRA, R.; GARCIA, A.; LEE, J.; CONTE, T.; OIZUMI, W.; DE MELLO, R.; LOPES, A.; VALENTIM, N.; OLIVEIRA, E. ; OTHERS. **How do software developers identify design problems? a qualitative analysis**. In: PROCEEDINGS OF THE 31ST BRAZILIAN SYMPOSIUM ON SOFTWARE ENGINEERING (SBES), p. 54–63, 2017.

[40] PASCARELLA, L.; SPADINI, D.; PALOMBA, F. ; BACCHELLI, A.. **On the effect of code review on code smells**. In: PROCEEDINGS OF THE 27TH INTERNATIONAL CONFERENCE ON SOFTWARE ANALYSIS, EVOLUTION, AND REENGINEERING (SANER), p. 1–12, 2020.

[41] OLIVA, G. A.; STEINMACHER, I.; WIESE, I. ; GEROSA, M. A.. **What can commit metadata tell us about design degradation?** In: PROCEEDINGS OF THE 13TH INTERNATIONAL WORKSHOP ON PRINCIPLES OF SOFTWARE EVOLUTION (IWPSE), p. 18–27, 2013.

[42] PORTER, A.; SIY, H.; MOCKUS, A. ; VOTTA, L.. **Understanding the sources of variation in software inspections**. ACM Trans. Softw. Eng. Methodol. (TOSEM), 7(1):41–79, 1998.

[44] PARNAS, D. L.. **Software aging**. In: PROCEEDINGS OF THE 16TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE), p. 279–287. IEEE, 1994.

[46] LI, Z.; AVGERIOU, P. ; LIANG, P.. **A systematic mapping study on technical debt and its management**. J. Syst. Softw. (JSS), 101:193–220, 2015.

[48] BOSU, A.; CARVER, J. C.; HAFIZ, M.; HILLEY, P. ; JANNI, D.. **Identifying the characteristics of vulnerable code changes: An empirical study**. In: PROCEEDINGS OF THE 22ND ACM JOINT EUROPEAN SOFTWARE ENGINEERING CONFERENCE AND SYMPOSIUM ON THE FOUNDATIONS OF SOFTWARE ENGINEERING (FSE), p. 257–268, 2014.

[49] GRAVES, T. L.; KARR, A. F.; MARRON, J. S. ; SIY, H.. **Predicting fault incidence using software change history**. IEEE Trans. Softw. Eng. (TSE), 26(7):653–661, 2000.

[50] MCINTOSH, S.; KAMEI, Y.; ADAMS, B. ; HASSAN, A. E.. **The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects**. In: PROCEEDINGS OF THE 11TH INTERNATIONAL CONFERENCE ON MINING SOFTWARE REPOSITORIES (MSR), p. 192–201, 2014.

[51] BIRD, C.; NAGAPPAN, N.; MURPHY, B.; GALL, H. ; DEVANBU, P.. **Don't touch my code! examining the effects of ownership on software quality**. In: PROCEEDINGS OF THE 19TH ACM JOINT EUROPEAN SOFTWARE ENGINEERING CONFERENCE AND SYMPOSIUM ON THE FOUNDATIONS OF SOFTWARE ENGINEERING (FSE), p. 4–14, 2011.

[52] ALENEZI, M.; ZAROUR, M.. **An empirical study of bad smells during software evolution using designite tool**. i-Manager's Journal on Software Engineering, 12(4):12, 2018.

[53] OLIVEIRA, A.; SOUSA, L.; OIZUMI, W. ; GARCIA, A.. **On the prioritization of design-relevant smelly elements: A mixed-method, multi-project study**. In: PROCEEDINGS OF THE 13TH BRAZILIAN SYMPOSIUM ON SOFTWARE COMPONENTS, ARCHITECTURES, AND REUSE (SBCARS), p. 83–92, 2019.

[55] ECLIPSE. `https://git.eclipse.org/r/#/c/53827/`, 2020. Accessed in: July 2019.

[56] COUCHBASE. `http://review.couchbase.org/#/c/11099/`, 2020. Accessed in: July 2019.

[58] UCHÔA, A.; BARBOSA, C.; OIZUMI, W.; BLENÍLIO, P.; LIMA, R.; GARCIA, A. ; BEZERRA, C.. **Replication package for the paper: "how does modern code review impact software design degradation? an in-depth empirical study"**. https://zenodo.org/record/3989787, 2020. Accessed: 2020-08-18.

[59] UCHÔA, A.; BARBOSA, C.; COUTINHO, D.; OIZUMI, W.; K. G. ASSUNÇÃO, W.; VERGILIO, S.; ALVES PEREIRA, J.; OLIVEIRA, A. ; GARCIA, A.. **Replication Package for the paper: "Predicting Design Impactful Changes in Modern Code Review: A Large-Scale Empirical Study"**. https://doi.org/10.5281/zenodo.4563214, Feb. 2021.

[60] PAIXÃO, M.; UCHÔA, A.; BIBIANO, A. C.; OLIVEIRA, D.; GARCIA, A.; KRINKE, J. ; ARVONIO, E.. **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 (MSR), 2020.

[61] CEDRIM, D.; GARCIA, A.; MONGIOVI, M.; GHEYI, R.; SOUSA, L.; DE MELLO, R.; FONSECA, B.; RIBEIRO, M. ; CHÁVEZ, A.. **Under-**

standing the impact of refactoring on smells: A longitudinal study of 23 software projects. In: PROCEEDINGS OF THE 11TH ACM JOINT EUROPEAN SOFTWARE ENGINEERING CONFERENCE AND SYMPOSIUM ON THE FOUNDATIONS OF SOFTWARE ENGINEERING (FSE), p. 465–475, 2017.

[63] PALOMBA, F.; BAVOTA, G.; DI PENTA, M.; FASANO, F.; OLIVETO, R. ; DE LUCIA, A.. **On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation**. Emp. Softw. Eng. (ESE), 23(3):1188–1221, 2018.

[64] KHOMH, F.; DI PENTA, M.; GUÉHÉNEUC, Y.-G. ; ANTONIOL, G.. **An exploratory study of the impact of antipatterns on class change- and fault-proneness**. Emp. Softw. Eng. (ESE), 17(3):243–275, 2012.

[66] HIRAO, T.; IHARA, A.; UEDA, Y.; PHANNACHITTA, P. ; MATSUMOTO, K.-I.. **The impact of a low level of agreement among reviewers in a code review process**. In: PROCEEDINGS OF THE 12TH INTERNATIONAL CONFERENCE ON OPEN SOURCE SYSTEMS (OSS), p. 97–110, 2016.

[67] AHMED, I.; MANNAN, U. A.; GOPINATH, R. ; JENSEN, C.. **An empirical study of design degradation: How software projects get worse over time**. In: PROCEEDINGS OF THE 10TH INTERNATIONAL SYMPOSIUM ON EMPIRICAL SOFTWARE ENGINEERING AND MEASUREMENT (ESEM), p. 1–10. IEEE, 2015.

[68] MANNAN, U. A.; AHMED, I.; ALMURSHED, R. A. M.; DIG, D. ; JENSEN, C.. **Understanding code smells in android applications**. In: PROCEEDINGS OF THE 3RD INTERNATIONAL CONFERENCE ON MOBILE SOFTWARE ENGINEERING AND SYSTEMS (MOBILESOFT), p. 225–236. IEEE, 2016.

[69] MARTINS, J.; BEZERRA, C.; UCHÔA, A. ; GARCIA, A.. **Are code smell co-occurrences harmful to internal quality attributes? a mixed-method study**. In: PROCEEDINGS OF THE 34TH BRAZILIAN SYMPOSIUM ON SOFTWARE ENGINEERING (SBES), p. 1 – 10, 2020.

[70] UCHÔA, A.; BARBOSA, C.; COUTINHO, D.; OIZUMI, W.; ASSUNÇAO, W. K.; VERGILIO, S. R.; PEREIRA, J. A.; OLIVEIRA, A. ; GARCIA, A.. **Predicting design impactful changes in modern code review: A large-scale empirical study**. In: PROCEEDINGS OF THE 18TH

INTERNATIONAL CONFERENCE ON MINING SOFTWARE REPOSITORIES (MSR), p. 1–12. IEEE, 2021.

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

[74] DE MELLO, R.; UCHÔA, A.; OLIVEIRA, R.; OIZUMI, W.; SOUZA, J.; MENDES, K.; OLIVEIRA, D.; FONSECA, B. ; GARCIA, A.. **Do research and practice of code smell identification walk together? a social representations analysis**. In: PROCEEDINGS OF THE 13TH INTERNATIONAL SYMPOSIUM ON EMPIRICAL SOFTWARE ENGINEERING AND MEASUREMENT (ESEM), p. 1–6. IEEE, 2019.

[77] LEWIS, C.; LIN, Z.; SADOWSKI, C.; ZHU, X.; OU, R. ; WHITEHEAD, E. J.. **Does bug prediction support human developers? findings from a google case study**. In: PROCEEDINGS OF THE 35TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE), p. 372–381. IEEE, 2013.

[78] KOTSIANTIS, S. B.; ZAHARAKIS, I. D. ; PINTELAS, P. E.. **Machine learning: a review of classification and combining techniques**. Artificial Intelligence Review, 26(3):159–190, 2006.

[79] MENZIES, T.; SHEPPERD, M.. **"bad smells" in software analytics papers**. Inf. Softw. Technol. (IST), 112:35–47, 2019.

[80] HUANG, W.; LU, T.; ZHU, H.; LI, G. ; GU, N.. **Effectiveness of conflict management strategies in peer review process of online collaboration projects**. In: PROCEEDINGS OF THE 19TH CONFERENCE ON COMPUTER SUPPORTED COOPERATIVE WORK (CSCW), p. 717–728, 2016.

[81] RUANGWAN, S.; THONGTANUNAM, P.; IHARA, A. ; MATSUMOTO, K.. **The impact of human factors on the participation decision of reviewers in modern code review**. Emp. Softw. Eng. (ESE), 24(2):973–1016, 2019.

[82] THONGTANUNAM, P.; MCINTOSH, S.; HASSAN, A. E. ; IIDA, H.. **Review participation in modern code review**. Emp. Softw. Eng. (ESE), 22(2):768–817, 2017.

[85] BOSU, A.; CARVER, J. C.; BIRD, C.; ORBECK, J. ; CHOCKLEY, C.. **Process aspects and social dynamics of contemporary code review: Insights from open source development and industrial practice at microsoft.** IEEE Trans. Softw. Eng. (TSE), 43(1):56–75, 2016.

[89] EBERT, F.; CASTOR, F.; NOVIELLI, N. ; SEREBRENIK, A.. **Confusion in code reviews: Reasons, impacts, and coping strategies.** In: PROCEEDINGS OF THE 26TH INTERNATIONAL CONFERENCE ON SOFTWARE ANALYSIS, EVOLUTION, AND REENGINEERING (SANER), p. 49–60. IEEE, 2019.

[90] UCHÔA, A.; BARBOSA, C.; OIZUMI, W.; BLENILIO, P.; LIMA, R.; GARCIA, A. ; BEZERRA, C.. **How does modern code review impact software design degradation? an in-depth empirical study.** In: PROCEEDINGS OF THE 36TH INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE AND EVOLUTION (ICSME), p. 511–522. IEEE, 2020.

[91] OIZUMI, W.; GARCIA, A.; SOUSA, L. D. S.; CAFEO, B. ; ZHAO, Y.. **Code anomalies flock together: Exploring code anomaly agglomerations for locating design problems.** In: PROCEEDINGS OF THE 38TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE), p. 440–451. IEEE, 2016.

[92] TANG, A.; ALETI, A.; BURGE, J. ; VAN VLIET, H.. **What makes software design effective?** Design Studies, 31(6):614–640, 2010.

[93] ALVES PEREIRA, J.; ACHER, M.; MARTIN, H. ; JÉZÉQUEL, J.-M.. **Sampling effect on performance prediction of configurable systems: A case study.** In: PROCEEDINGS OF THE 11TH INTERNATIONAL CONFERENCE ON PERFORMANCE ENGINEERING (ICPE), p. 277–288, 2020.

[94] KAMEI, Y.; SHIHAB, E.; ADAMS, B.; HASSAN, A. E.; MOCKUS, A.; SINHA, A. ; UBAYASHI, N.. **A large-scale empirical study of just-in-time quality assurance.** IEEE Trans. Softw. Eng. (TSE), 39(6):757–773, 2012.

[95] KRISHNAN, S.; STRASBURG, C.; LUTZ, R. R. ; GOŠEVA-POPSTOJANOVA, K.. **Are change metrics good predictors for an evolving software product line?** In: PROCEEDINGS OF THE

7TH INTERNATIONAL CONFERENCE ON PREDICTIVE MODELS IN SOFTWARE ENGINEERING (PROMISE), p. 1–10, 2011.

[96] MATSUMOTO, S.; KAMEI, Y.; MONDEN, A.; MATSUMOTO, K.-I. ; NAKAMURA, M.. **An analysis of developer metrics for fault prediction.** In: PROCEEDINGS OF THE 6TH INTERNATIONAL CONFERENCE ON PREDICTIVE MODELS IN SOFTWARE ENGINEERING (PROMISE), p. 1–9, 2010.

[97] ZANETTI, M. S.; SCHOLTES, I.; TESSONE, C. J. ; SCHWEITZER, F.. **Categorizing bugs with social networks: a case study on four open source software communities.** In: PROCEEDINGS OF THE 35TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE), p. 1032–1041. IEEE, 2013.

[98] BAYSAL, O.; KONONENKO, O.; HOLMES, R. ; GODFREY, M. W.. **The influence of non-technical factors on code review.** In: PROCEEDINGS OF THE 20TH WORKING CONFERENCE ON REVERSE ENGINEERING (WCRE), p. 122–131. IEEE, 2013.

[99] MOCKUS, A.; WEISS, D. M.. **Predicting risk of software changes.** Bell Labs Technical Journal, 5(2):169–180, 2000.

[100] YAZDI, H. S.; MIRBOLOUKI, M.; PIETSCH, P.; KEHRER, T. ; KELTER, U.. **Analysis and prediction of design model evolution using time series.** In: PROCEEDINGS OF THE 26TH INTERNATIONAL CONFERENCE ON ADVANCED INFORMATION SYSTEMS ENGINEERING (CAISE), p. 1–15. Springer, 2014.

[101] D'AMBROS, M.; LANZA, M. ; ROBBES, R.. **An extensive comparison of bug prediction approaches.** In: PROCEEDINGS OF THE 7TH INTERNATIONAL CONFERENCE ON MINING SOFTWARE REPOSITORIES (MSR), p. 31–41. IEEE, 2010.

[102] TAYLOR, R. N.; VAN DER HOEK, A.. **Software design and architecture the once and future focus of software engineering.** In: PROCEEDINGS OF THE FOSE'07, p. 226–243. IEEE, 2007.

[103] SONG, Q.; GUO, Y. ; SHEPPERD, M.. **A comprehensive investigation of the role of imbalanced learning for software defect prediction.** IEEE Trans. Softw. Eng. (TSE), 45(12):1253–1269, 2018.

[105] WHITLEY, E.; BALL, J.. **Statistics review 6: Nonparametric methods.** Critical care, 6(6):509, 2002.

[106] MCDONALD, J. H.. **Handbook of biological statistics**, volumen 2. sparky house publishing Baltimore, MD, 2009.

[107] GRISSOM, R. J.; KIM, J. J.. **Effect sizes for research: A broad practical approach.** Lawrence Erlbaum Associates Publishers, 2005.

[108] ROMANO, J.; KROMREY, J. D.; CORAGGIO, J.; SKOWRONEK, J. ; DEVINE, L.. **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, p. 1–51. Citeseer, 2006.

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

[111] FAN, Y.; XIA, X.; LO, D. ; LI, S.. **Early prediction of merged code changes to prioritize reviewing tasks.** Emp. Softw. Eng. (ESE), 23(6):3346–3393, 2018.

[112] HERZIG, K.; JUST, S. ; ZELLER, A.. **It's not a bug, it's a feature: how misclassification impacts bug prediction.** In: PROCEEDINGS OF THE 35TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE), p. 392–401. IEEE, 2013.

[113] PAIXAO, M.; MAIA, P. H.. **Rebasing in code review considered harmful: A large-scale empirical investigation.** In: PROCEEDINGS OF THE 19TH WORKING CONFERENCE ON SOURCE CODE ANALYSIS AND MANIPULATION (SCAM), p. 45–55. IEEE, 2019.

[114] SHARMA, T.; SINGH, P. ; SPINELLIS, D.. **An empirical investigation on the relationship between design and architecture smells.** Emp. Softw. Eng. (ESE), 2020.

[114] PEREIRA, J. A.; MARTIN, H.; ACHER, M.; JÉZÉQUEL, J.-M.; BOTTERWECK, G. ; VENTRESQUE, A.. **Learning software configuration spaces: A systematic literature review.** arXiv preprint arXiv:1906.03018, 2019.

[115] HUTTER, F.; KOTTHOFF, L. ; VANSCHOREN, J.. **Automated machine learning: methods, systems, challenges.** Springer Nature, 2019.

[116] KOHAVI, R.; OTHERS. **A study of cross-validation and bootstrap for accuracy estimation and model selection**. In: IJCAI, volumen 14, p. 1137–1145. Montreal, Canada, 1995.

[118] IOFFE, S.; SZEGEDY, C.. **Batch normalization: Accelerating deep network training by reducing internal covariate shift**. Proceedings of the 32nd International Conference on Machine Learning (ICML), 2015.

[129] PEDREGOSA, F.; VAROQUAUX, G.; GRAMFORT, A.; MICHEL, V.; THIRION, B.; GRISEL, O.; BLONDEL, M.; PRETTENHOFER, P.; WEISS, R.; DUBOURG, V.; VANDERPLAS, J.; PASSOS, A.; COURNAPEAU, D.; BRUCHER, M.; PERROT, M. ; DUCHESNAY, E.. **Scikit-learn: Machine learning in Python**. Journal of Machine Learning Research, 12:2825–2830, 2011.

[130] FAWCETT, T.. **An introduction to roc analysis**. Pattern Recogn. Lett., 27(8):861–874, 2006.

[131] ECLIPSE. `https://git.eclipse.org/r/#/c/3345/`, 2020. Accessed in: August 2020.

[132] ECLIPSE. `https://git.eclipse.org/r/#/c/825/`, 2020. Accessed in: August 2020.

[133] SOARES, V.; OLIVEIRA, A.; FARAH, P.; BIBIANO, A.; COUTINHO, D.; GARCIA, A.; VERGILIO, S.; SCHOTS, M.; OLIVEIRA, D. ; UCHÔA, A.. **On the relation between complexity, explicitness, effectiveness of refactorings and non-functional concerns**. In: PROCEEDINGS OF THE 34TH BRAZILIAN SYMPOSIUM ON SOFTWARE ENGINEERING (SBES), p. 788–797, 2020.

[134] SURYANARAYANA, G.; SAMARTHYAM, G. ; SHARMA, T.. **Refactoring for software design smells: managing technical debt**. Morgan Kaufmann, 2014.

[135] CAULO, M.; LIN, B.; BAVOTA, G.; SCANNIELLO, G. ; LANZA, M.. **Knowledge transfer in modern code review**. In: PROCEEDINGS OF THE 28TH INTERNATIONAL CONFERENCE ON PROGRAM COMPREHENSION (ICPC), p. 230–240, 2020.

[136] UCHÔA, A.; BARBOSA, C.; OIZUMI, W.; BLENILIO, P.; LIMA, R.; GARCIA, A. ; BEZERRA, C.. **How does modern code review impact software design degradation? an in-depth empirical study**. In: PRO-

CEEDINGS OF THE 36TH INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE AND EVOLUTION (ICSME), p. 511–522. IEEE, 2020.

[145] PALOMBA, F.; BAVOTA, G.; PENTA, M. D.; OLIVETO, R.; POSHYVANYK, D. ; LUCIA, A. D.. **Mining version histories for detecting code smells**. IEEE Trans. Softw. Eng. (TSE), 41(5):462–489, 2015.

[147] ARCELLI FONTANA, F.; MÄNTYLÄ, M. V.; ZANONI, M. ; MARINO, A.. **Comparing and experimenting machine learning techniques for code smell detection**. Emp. Softw. Eng. (ESE), 21:1143 – 1191, 2016.

[148] LAL, H.; PAHWA, G.. **Code review analysis of software system using machine learning techniques**. In: PROCEEDINGS OF THE 11TH INTERNATIONAL CONFERENCE ON INTELLIGENT SYSTEMS AND CONTROL (ISCO), p. 8–13, 2017.

[149] FRIEDMAN, M.. **The use of ranks to avoid the assumption of normality implicit in the analysis of variance**. Journal of the american statistical association, 32(200):675–701, 1937.

[154] ROSEN, C.; GRAWI, B. ; SHIHAB, E.. **Commit guru: analytics and risk prediction of software commits**. In: PROCEEDINGS OF THE 10TH ACM JOINT EUROPEAN SOFTWARE ENGINEERING CONFERENCE AND SYMPOSIUM ON THE FOUNDATIONS OF SOFTWARE ENGINEERING (FSE), p. 966–969, 2015.

[155] HINDLE, A.; GERMAN, D. M. ; HOLT, R.. **What do large commits tell us? a taxonomical study of large commits**. In: PROCEEDINGS OF THE 5TH INTERNATIONAL CONFERENCE ON MINING SOFTWARE REPOSITORIES (MSR), p. 99–108, 2008.

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

[157] MENDEZ, D.; GRAZIOTIN, D.; WAGNER, S. ; SEIBOLD, H.. **Open Science in Software Engineering**, p. 477–501. Springer International Publishing, Cham, 2020.

[158] UCHÔA, A.. **Unveiling multiple facets of design degradation in modern code review**. Proceedings of the 29th ACM Joint European

Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), p. 1–5, 2021.

[159] HAN, X.; TAHIR, A.; LIANG, P.; COUNSELL, S. ; LUO, Y.. **Understanding code smell detection via code review: A study of the openstack community**. Proceedings of the 29th International Conference on Program Comprehension (ICPC), 2021.

[160] MANNAN, U. A.; AHMED, I.; JENSEN, C. ; SARMA, A.. **On the relationship between design discussions and design quality: a case study of apache projects**. In: PROCEEDINGS OF THE 28TH ACM JOINT MEETING ON EUROPEAN SOFTWARE ENGINEERING CONFERENCE AND SYMPOSIUM ON THE FOUNDATIONS OF SOFTWARE ENGINEERING (ESEC/FSE), p. 543–555, 2020.

[162] SOUSA, A.; UCHÔA, A.; FERNANDES, E.; BEZERRA, C. I.; MONTEIRO, J. M. ; ANDRADE, R. M.. **Rem4dspl: A requirements engineering method for dynamic software product lines**. In: PROCEEDINGS OF THE XVIII BRAZILIAN SYMPOSIUM ON SOFTWARE QUALITY (SBQS), p. 129–138, 2019.

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

[164] LIMA, L.; UCHÔA, A.; BEZERRA, C.; COUTINHO, E. ; ROCHA, L.. **Visualizing the maintainability of feature models in spls**. In: ANAIS DO VIII WORKSHOP DE VISUALIZAÇÃO, EVOLUÇÃO E MANUTENÇÃO DE SOFTWARE (VEM), p. 1–8. SBC, 2020.

[165] MARTINS, J.; BEZERRA, C.; UCHÔA, A. ; GARCIA, A.. **How do code smell co-occurrences removal impact internal quality attributes? a developers' perspective**. In: PROCEEDINGS OF THE 35TH BRAZILIAN SYMPOSIUM ON SOFTWARE ENGINEERING (SBES), p. 1–10, 2021.

[166] UCHÔA, A.; ASSUNÇAO, W. K. ; GARCIA, A.. **Do critical components smell bad? an empirical study with component-based software**

product lines. In: PROCEEDINGS OF THE 15TH BRAZILIAN SYMPOSIUM ON SOFTWARE COMPONENTS, ARCHITECTURES, AND REUSE (SBCARS), p. 1–10, 2021.

[167] FAGAN, M. E.. **Design and code inspections to reduce errors in program development**. IBM Systems Journal, 38(2.3):258–287, 1999.

[168] WIEGERS, K. E.. **Peer reviews in software: A practical guide**. Addison-Wesley Boston, 2002.

[169] MARTIN, R. C.; NEWKIRK, J. ; KOSS, R. S.. **Agile software development: principles, patterns, and practices**, volumen 2. Prentice Hall Upper Saddle River, NJ, 2003.

[170] RIGBY, P. C.; GERMAN, D. M.; COWEN, L. ; STOREY, M.-A.. **Peer review on open-source software projects: Parameters, statistical models, and theory**. ACM Transactions on Software Engineering and Methodology (TOSEM), 23(4):1–33, 2014.

[171] ELLIS, C. A.; GIBBS, S. J. ; REIN, G.. **Groupware: some issues and experiences**. Communications of the ACM, 34(1):39–58, 1991.

[172] HOEGL, M.; GEMUENDEN, H. G.. **Teamwork quality and the success of innovative projects: A theoretical concept and empirical evidence**. Organization science, 12(4):435–449, 2001.

[173] DE ALMEIDA, J. R.; CAMARGO, J. B.; BASSETO, B. A. ; PAZ, S. M.. **Best practices in code inspection for safety-critical software**. IEEE software, 20(3):56–63, 2003.

[174] GOMES, A. B.; SILVA, M.; VALADARES, D. C. G.; PERKUSICH, M.; ALBUQUERQUE, D.; ALMEIDA, H. O. ; PERKUSICH, A.. **Evaluating the relationship of personality and teamwork quality in the context of agile software development**. In: SEKE, p. 311–316, 2020.

[175] MICHELL, J.. **Measurement in psychology: A critical history of a methodological concept**, volumen 53. Cambridge University Press, 1999.

[176] FELDT, R.; TORKAR, R.; ANGELIS, L. ; SAMUELSSON, M.. **Towards individualized software engineering: empirical studies should collect psychometrics**. In: PROCEEDINGS OF THE 2008 INTERNATIONAL WORKSHOP ON COOPERATIVE AND HUMAN ASPECTS OF SOFTWARE ENGINEERING, p. 49–52, 2008.

[177] MARSLAND, S.. **Machine learning: an algorithmic perspective**. Chapman and Hall/CRC, 2011.

[178] BISHIP, C. M.. **Pattern recognition and machine learning (information science and statistics)**, 2007.

[179] ZHANG, H.. **Exploring conditions for the optimality of naive bayes**. International Journal of Pattern Recognition and Artificial Intelligence, 19(02):183–198, 2005.

[180] CORTES, C.; VAPNIK, V.. **Support-vector networks**. Machine learning, 20(3):273–297, 1995.

[181] BREIMA, L.. **Random forests. machine learning**. 2010.

[182] HO, T. K.. **Random decision forests**. In: PROCEEDINGS OF 3RD INTERNATIONAL CONFERENCE ON DOCUMENT ANALYSIS AND RECOGNITION, volumen 1, p. 278–282. IEEE, 1995.

[183] QUINLAN, J. R.. **Induction of decision trees**. Machine learning, 1(1):81–106, 1986.

[184] NATEKIN, A.; KNOLL, A.. **Gradient boosting machines, a tutorial**. Frontiers in neurorobotics, 7:21, 2013.

[185] STOREY, M.-A.; TREUDE, C.. **Software engineering dashboards: Types, risks, and future**. In: RETHINKING PRODUCTIVITY IN SOFTWARE ENGINEERING, p. 179–190. Springer, 2019.

[186] IBRAHIM, W. M.; BETTENBURG, N.; SHIHAB, E.; ADAMS, B. ; HASSAN, A. E.. **Should i contribute to this discussion?** In: PROCEEDINGS OF THE 7TH IEEE WORKING CONFERENCE ON MINING SOFTWARE REPOSITORIES (MSR), p. 181–190. IEEE, 2010.

[187] DABBISH, L.; STUART, C.; TSAY, J. ; HERBSLEB, J.. **Social coding in github: transparency and collaboration in an open software repository**. In: PROCEEDINGS OF THE 15TH CONFERENCE ON COMPUTER SUPPORTED COOPERATIVE WORK AND SOCIAL COMPUTING (CSCW), p. 1277–1286, 2012.

[188] GIUFFRIDA, R.; DITTRICH, Y.. **Empirical studies on the use of social software in global software development–a systematic mapping study**. Information and Software Technology, 55(7):1143–1164, 2013.

[189] STOREY, M.-A.; ZAGALSKY, A.; FIGUEIRA FILHO, F.; SINGER, L. ; GERMAN, D. M.. **How social and communication channels shape and challenge a participatory culture in software development**. IEEE Transactions on Software Engineering, 43(2):185–204, 2016.