IMPACT OF CODE REVIEW PROCESS SMELLS ON CODE SMELLS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR
THE DEGREE OF
MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

By
Erdem Tuna
January 2023

Impact of Code Review Process Smells on Code Smells
By Erdem Tuna
January 2023

We certify that we have read this thesis and that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Eray Tüzün (Advisor)

Uğur Doğrusöz

İsmail Sengör Altıngövde

Approved for the Graduate School of Engineering and Science:

Orhan Arıkan
Director of the Graduate School

ABSTRACT

IMPACT OF CODE REVIEW PROCESS SMELLS ON CODE SMELLS

Erdem Tuna
M.S. in Computer Engineering
Advisor: Eray Tüzün
January 2023

The code review process is conducted by software teams with various motivations. Among other goals, code reviews act as a gatekeeper for software quality. Software quality comprises several aspects, maintainability (i.e., code quality) being one of them. In this study, we explore whether code review process quality (as evidenced by the presence of code review process smells) influences software maintainability (as evidenced by the presence of code smells). In other words, we investigate whether smells in the code review process are related to smells in the code that was reviewed by using correlation analysis. We augment our quantitative analysis with a focus group study to learn practitioners' opinions. Contrary to our own intuition and that of the practitioners in our focus groups, we found that code review process smells have little to no correlation with the level of code smells. Further investigations revealed that the level of code smells neither increases nor decreases in 8 out of 10 code reviews, regardless of the quality of the code review. We identified multiple potential reasons behind the counter-intuitive results based on our focus group data. Furthermore, practitioners still believe that code reviews are helpful in improving software quality. Our results imply that the community should update our goals for code review practices and reevaluate those practices to align them with more relevant and modern realities.

Keywords: Code Reviews, Code Review Smells, Process Smells, Code Smells, Focus Group, Empirical Study, Mining Software Repositories.

ÖZET

KOD GÖZDEN GEÇİRME SÜRECİNDEKİ KÖTÜ UYGULAMALARIN KOD KUSURLARI ÜZERİNDEKİ ETKİSİ

Erdem Tuna
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Danışmanı: Eray Tüzün
Ocak 2023

Kod gözden geçirme süreci, yazılım geliştirme takımları tarafından birçok amaçla yürütülmektedir. Bu amaçlardan biri de yazılım kalitesini korumaktır. Yazılım kalitesi, sürdürülebilirliği de ele alan geniş bir kavramdır. Bu çalışmada, kod gözden geçirme süreci kalitesinin, bu süreçteki kötü uygulamalar açısından ele alarak yazılım sürdürülebilirliğine olan etkisini kod kusurları açısından inceliyoruz. Diğer bir deyişle, kod gözden geçirme sürecindeki kötü uygulamalar ile kod kusurları arasında ilişki olup olmadığını korelasyon analizi ile araştırıyoruz. Çalışmada nicel analiz, odak küme çalışması ile desteklenerek yazılım geliştiricilerin konu hakkındaki görüşleri alınmıştır. Beklentilere ters olarak, kod gözden geçirme sürecindeki kötü uygulamaların kod kusurlarının yoğunluğu ile ilişkisinin zayıf olduğu veya ilişkisinin bulunmadığını saptadık. Sonrasında yaptığımız incelemelerde, gözden geçirme sürecinin kalitesinden bağımsız olarak, her 10 gözden geçirmenin 8'inde kod kusurlarının ne arttığı ne de azaldığı görülmüştür. Elde edilen beklenmedik sonuçların ardında, odak küme çalışmamızdaki verileri de göz önünde bulundurarak birden çok neden belirledik. Ek olarak, yazılım geliştiricilerinin hala kod gözden geçirme sürecinin yazılım kalitesini artırmada yardımcı olduğunu düşündüğü saptanmıştır. Elde ettiğimiz sonuçlar, geliştiricilerin kod gözden geçirme süreci yürütme amaçlarını güncellemesi gerektiğine ve süreç içerisindeki uygulamalarını yeniden değerlendirerek modern araçlarla ve güncel gerçekliklerle uyumlu hale getirmeleri gerektiğine işaret ediyor.

Anahtar sözcükler: Kod Gözden Geçirme, Kod Gözden Geçirme Kötü Uygulamaları, Süreç Uygunsuzlukları, Kod Kusurları, Odak Küme, Deneysel Araştırma, Yazılım Deposu Madenciliği.

Acknowledgement

I would like to express my deepest gratitude to my advisor Asst. Prof. Eray Tüzün for his invaluable guidance, support, and encouragement throughout the course of my studies. His knowledge and expertise have been essential for shaping my academic ad professional growth. I would like to extend my appreciation to Prof. Dr. Carolyn Seaman for her invaluable contributions as a co-author on the paper version of my thesis. Her guidance and contributions have been invaluable to the success of this work.

I am grateful to my esteemed jury members, Prof. Dr. Uğur Doğrusöz and Prof. Dr. İsmail Sengör Altıngövde, for their invaluable time and consideration in reviewing my work.

I would also like to extend my heartfelt thanks to my family for their unwavering love and support. Their encouragement and belief in me have greatly motivated me throughout this endeavor.

I would like to express my gratitude to Emre Sülün and other BILSEN members for their ideas and feedback during my research. I extend my gratitude to my friends for their support and companionship.

I have a dream...

Erdem

January 2023, Ankara

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software code reviews have been commonly employed by software development teams [2, 3, 4, 5, 6] for decades to help assure software quality [7]. While processes and definitions vary, code reviews are generally understood to involve visual inspection of source code, by persons different from the author of the code being inspected, for the purpose of finding defects [8, 9, 10, 11, 12], improving code quality [9, 10, 11, 12, 13], and knowledge transfer [11, 9].

Software quality is a broad term encompassing many aspects [14, 15]. Code reviews are meant to address several of them. Most obviously, the code review process is meant to reduce defects [11, 9], which improves reliability, an important quality attribute [16]. Another important quality attribute is maintainability [17, 18], one indicator of which is the presence of code smells [19, 20, 21, 22]. Code smells are violations of good programming practices that can be detected in the source code, that are believed to make the code harder to understand and modify. Thus, code containing code smells is said to have lower maintainability [22, 23, 24, 25, 26]. Code reviews can improve maintainability by detecting and removing code smells and other sub-optimal coding constructs that would hinder future maintenance.

1.1 Research Problem

While the benefits of code reviews are well known, there are common ways in which they can be performed less than optimally. These sub-optimal processes have been characterized into code review process smells (CRPSs), which can be detected by examining data from code reviews [27].

Intuitively, the presence of CRPSs would have a negative impact on code quality (since code reviews are meant to positively impact code quality). However, it is not known whether or to what extent sub-optimal (i.e., smelly) code reviews can impact maintainability.

Our investigation fits well into the literature on technical debt (TD), in particular, the role of process debt, of which CRPSs are one type. We describe this connection to the TD literature in Chapter 2, but generally speaking, we take the perspective that process debt (in the form of CRPSs) can be a precursor to technical debt (in the form of code smells). It is this causal relationship that we investigate in this work. Thus, the aim of the study described here is to explore the hypothesis that the presence of CRPSs leads to the presence of code smells in the code that was reviewed. In particular, we address the following research questions:

RQ1: Do code review process smells influence the existence of code smells?

RQ2: Does overall code review process quality affect code quality?

RQ3: How effective is the code review process on reducing code smells?

RQ4: How do developers view the relationship between code review process smells and code smells?

To address these questions, we conduct an analysis of data from open-source software projects that identify smelly and non-smelly code reviews and the code smells introduced or removed during the review process. We also conduct a focus

group study to query software practitioners specifically about the relationships they see between code review process smells and code smells.

We aim not just to investigate the correlation between CRPSs and code smells but also to provide evidence of a causal relationship. We do this by conducting our correlational analysis with a narrow focus on code smell changes that occurred only in the context of a code review (thus removing significant confounding factors). We also augment our analysis with expert opinion, specifically on the causal nature of the relationship. In the end, we failed to find even a correlational relationship, so causality was not supported.

1.2 Contributions of the Thesis

The main contributions of the thesis are as below.

- We explore whether code review process quality (quantified as the presence of code review process smells) influences software maintainability (from code smells perspective).
- We investigate the change of code smells in code review process.
- We augment our quantitative analysis with a focus group study to learn practitioners' opinions.

Thesis Organization. Chapter 2 discusses the related work. The background information on the main concepts is provided in Chapter 3. We define the terminology and concepts that underlie our study design in Chapter 4, which is followed by Chapter 5 in which we detail our study design. Chapter 6 lists our findings and results. We discuss and elaborate on the potential implications of the results in Chapter 7. Threats to validity of our study are presented in Chapter 8. Chapter 9 includes our concluding remarks.

Chapter 2

Related Work

2.1 Modern Code Review

Since its first formal introduction as a meeting-based inspection process by Fagan in 1976 [28], code review has been a critical part of the software development life cycle. Even though the fundamental rationale has not deviated, the code review process has evolved into a tool-based practice today that we refer to as the *modern code review*.

2.1.1 Motivations

A large body of research exists in the literature that tries to understand the purposes of having a code review process and its benefits. The studies generally utilize qualitative techniques such as interviews [11, 9, 4, 29], surveys [9, 29, 11], and analysis of pull request comments [11]. Quantitative pull request data analysis is also used [29]. The studies list several items as the motivations behind code reviews. They agree that improving the code and increasing code quality is considered one of the primary reasons for conducting code reviews [11, 9, 4, 29]. Finding bugs [11, 9, 4, 29], design and solution evaluation [11, 4], knowledge

transfer [11, 9], educational purposes [29, 4], and code convention checks [29] are the other main driving factors for having a review process.

In our study, we are interested in analyzing the code quality aspect of the code review process from a code smells perspective.

2.1.2 What Do Reviewers Find?

The previous subsection summarizes the works that explore the driving factors for conducting a code review process in a software development life cycle. In this subsection, first, we list studies that detail the developers' perspectives, findings, and detections during the code review process. The literature on this topic mainly revolves around exploring how developers identify defects and code smells during code reviews. Besides, some works conduct more in-depth analysis on what kind of bugs or code smells developers are inclined to find.

Mäntylä and Lassenius [30] conducted an experimental study involving two groups (developers and students) to identify the most frequently found defect types during the code review process. They observed the defect identification of two groups in code reviews. One of the groups comprised developers working in a company, whereas the other consisted of students taking a quality assurance course. The authors classified the identified defects into two categories: evolvability and functional. Their analysis showed that both groups' 75% of the identified defects were of the evolvability category, and the remaining were functional. Parallel to the findings, the authors concluded that code review is a good process for detecting evolvability defects. Beller et al. [31] explored the fixed defects during code reviews of open-source software. They confirmed the findings in [30], almost 75% of the fixed defects were evolvability defects.

AlOmar et al. [1] analyzed the differences between the statistical characteristics of refactoring-related and non-refactoring-related code reviews. Besides, they investigated the theme of refactoring-related reviews on OpenStack projects. Initially, they searched for the reviews concerned with refactoring based on specific

keywords and then further filtered the data with manual methods. Their dataset resulted in 5,505 code reviews (out of 775,657). One of the emerging themes they identified is quality. The theme comprises reviewer suggestions on adherence to coding conventions, avoiding code smells, and correctness of design pattern practices. Code reviews about finding, avoiding, and removing code smells are studied extensively by [32]. Han et al. [32] investigated the code smells that were identified during reviews, the suggestions made by reviewers about those smells, and the actions that were taken based on those suggestions. They manually curated a dataset of 1,539 smell-related code review comments (out of 25,315) from open-source projects. Their analysis revealed that code smell identification was not largely prevalent. Furthermore, the root cause of the smells was a violation of conventions. Furthermore, they found that authors took the required actions to remove the smells most of the time once reviewers suggested a fix to code smells.

2.1.3 What Really Changes?

In the literature, there are efforts to quantitatively assess the changes occurring in code reviews. The studies generally use an analysis tool to measure the subject quantity and investigate its evolution in code reviews.

Panichella et al. [33] investigated whether warnings of the static code analysis tools (Checkstyle and PMD) are remedied or not in the code review process. Here, *warnings* correspond to issues related to software design issues, coding styles, and documentation. Their analysis depends on *warning density*, that is, the number of warnings the static analysis tool finds per line of code. They checked whether warning density distribution changes from initial patch to final patch sets[1]. The analysis consisted of almost 1K code reviews from six projects developed in Java language. The results showed that the warning density change in code reviews is small and not statistically significant. In contrast, their cumulative analysis of all code reviews demonstrated that the number of warnings decreased. They further detailed their analysis to find out the warning types discussed in code reviews.

[1] *Patch set* is a term in the Gerrit platform. *Patch set* in Gerrit is analogous to a *commit* in a GitHub pull request [34]

They found that some specific warnings are resolved more compared to other warning types. Han et al. [35] investigated the coding violation changes in the code review process. Their analysis involved almost 12.7K code reviews, and they detected coding violations using the Checkstyle tool. They utilized the CROP dataset [36] and selected four projects developed in Java primary language. The results showed that only in 25.3% of code reviews did the number of violations change from the initial to the final patch. They reported that the number of coding convention violations increased for 10 out of 11 coding convention categories from the initial patch to the final patch. Also, they found that the number of new coding convention violations increased with the patch size. Their results contradict that of Panichella et al. [33]. Lenarduzzi et al. [37][2] conducted a study to explore whether violations detected by the PMD tool impact the acceptance of pull requests. Their data collection contained 28 projects in Java language and 36K pull requests. They concluded that the pull request acceptance is not affected by the presence of violations detected by the tool. They supported the result with a manual investigation of 1925 accepted and 1705 rejected pull requests. They reported that none of the pull requests were rejected for code quality reasons.

While the studies mentioned above checked the change in all kinds of potential problems (some of which are code smells [38]) raised by the static code analysis tools during the code review process, there are studies analyzing the change in only the code smells. Pascarella et al. [39] analyzed the effect of code review on the code smell severity. Their analysis consisted of 21K code reviews from seven Java projects in the CROP dataset. Similar to the above studies, their projects use Gerrit as the code review platform. So, their analysis is based on checking the code smell severity variation from the initial to the final patch set. They employed heuristic-based code smell detection techniques and considered six code smell types. Their results suggest that the code smell severity decreases only in 4% of the code reviews under consideration. They also found that 95% of decreases are due to side effects of unrelated changes by conducting a manual analysis of code reviews.

[2]This work is not strictly built on the idea of code smell changes during code reviews. However, we believe that it is conceptually related and worth mentioning in this section of the thesis.

We believe that our work is complementary to the studies described above. On the one hand, we use a different static code analysis tool (SonarQube) to detect code smells, which allows us to examine a wider variety of smells in our study. On the other hand, we limit our analysis to code smells, a subset of the world of bad practices addressed in [33], [35], and [37]. Although it is narrower, our approach eliminates potentially irrelevant or out-of-scope issues (for human inspection in code reviews) and lets us focus on potentially more problematic issues related to source code.

2.1.4 Effects on Software Development

Our research community put effort into understanding the cumulative effect of the code review process on software quality and development. In their study, Davila and Nunes [40] divided the software quality concept into two, code quality and product quality. Here, code quality corresponds to following the best practices in design and programming, whereas product quality refers to the reduced number of bugs or security defects [40]. Morales et al. [41] investigated the relationship between code review coverage and code quality. They used the occurrence of design anti-patterns as a proxy for code quality. Their analysis showed that both code review coverage and participation have a negative impact on the occurrence of anti-patterns to some extent. It is suggested that developers have high participation to improve code quality. McIntosh et al. [42] explored the potential relation between post-release defects and code review metrics (similar to [41]). They used code review coverage, participation, and expertise metrics to account for the review quality. The study revealed that the used metrics are associated with post-release defects. However, it is noted that other properties of the code review process are also effective in this context. Shimagaki et al. [43] conducted a complementary study to that of [42] in a proprietary setting at Sony Mobile. They reported similar findings. Uchôa et al. [44] worked on the impact of code review on software design degradation. They found that review comments pointing out design improvements help reduce or avoid design degradation. They also denoted those design discussions on their own are not enough to prevent

design degradation. According to their study, long discussions increase the risk of degradation.

The existing literature indicates a potential relationship between the code review process and software quality. Even though the product quality aspect of this relation is widely studied in our research community, the code quality aspect is scarce [40]. In our study, we are interested in finding the effect of code review practices on code quality.

2.1.5 Code Review Process Smells

Our community established a large body of research on the *smell* concept. Smells generally refer to suboptimal activities or anti-patterns in software development processes. Smells could exist in software artifacts such as code [45, 46], test codes [47, 48], infrastructure as code [49, 50], architecture [51, 52], or processes such as continuous integration [53, 54], bug tracking [55], and code review [27]. As software development is a collaborative work, community smells are also identified in the literature [56, 57, 58]. In this thesis, we are particularly interested in the studies about smells in the code review process.

Chouchen et al. [59] explored the anti-patterns in the code review process. They suggested that the anti-patterns could be an indicator of tense review processes, and thus they may slow down integration and lead to a decrease in software quality. The authors diagnosed five anti-patterns based on the existing literature. The study also listed the symptoms of each anti-pattern and identified the potential consequences on the review process, artifact, and people. They manually probed the prevalence of the anti-patterns by selecting a random sample of 100 code review instances of the OpenStack project. Their results showed that 67% and 21% of the code reviews contain at least one anti-pattern and two anti-patterns, respectively. Doğan and Tüzün [27] categorized the bad practices (smells) in the code review process. They conducted a multivocal literature review in white and gray literature. Then, they obtained the developers' opinions through a survey and interviews. As a result, the authors devised a taxonomy

of seven code review process smells. The taxonomy includes each smell's definition, root causes, and potential side effects. The study further described the smell detection methodologies. After obtaining the taxonomy, they explored the quantitative evidence for the CRPSs in eight open-source projects. Their results suggest that the smells exist with a varying ratio in different projects.

In this study, we explore the association between CRPSs and code smells. For two reasons, we decided to use the CRPS taxonomy of Doğan and Tüzün [27]. First, their taxonomy is based on both a systematic review and a developer survey. Second, their study analyzed more projects and code reviews compared to [59].

2.2 Debt Concept in Software Engineering

Our research community investigated different debt types occurring in the current and evolving context of the software development life cycle. In general, debts provide interim benefits and cause detrimental consequences later. The existing literature explores the technical [60, 21], social [56, 61], and process [62, 63] debts to varying extents. We provide an overview of the relevant parts of existing studies on technical and process debts, which our study primarily relates to.

2.2.1 Technical Debt

Technical debt (TD) has been defined and refined several times in the literature since it was initially introduced by Cunningham [64] in 1992. Technical debt refers to *advantageous* short-term compromises that cause increased complexity and cause in the long run [65]. TD is explored from several perspectives in the literature, such as type, intentionality, time horizon, and degree of focus and strategy [66]. Li et al. [67] identified ten types of TD, one of which is code debt. Code smells are the main component of code debt and thus the evidence of TD [68]. In our study, code smells constitute one of the main concepts.

2.2.2 Process Debt

Rather *earlier* research [69, 67] hinted at the existence of process debt (PD) without mentioning the term. Alves et al. [63, 60] explicitly identified PD as a type of TD in their works. These studies [63, 60] framed PD as a TD type concerning the ineffective processes in software development. Later, two leading studies [70, 62] distinguished the PD as a debt type other than technical and social debts. These papers are of interest to our study, which we explain below.

Martini et al. [62] investigated whether different debt types (technical, social, and process) concern agile teams. They explored the extent that the debts are discussed and handled. The authors designed a case study to observe in-team and inter-team retrospective meetings of software development teams. Before conducting the study, they made the first definition of PD in the literature as "a sub-optimal activity or process that might have short-term benefits but generates a negative impact in the medium-long term." Their results showed that developers PD is widely mentioned in the retrospective meetings, considering the number of issues for each debt type. They discussed that PD requires more coordination between different teams than the other debt types.

Following [62], Martini et al. [70] conducted another study to explore PD systematically, filling the knowledge gap about PD. Their work tried to handle PD from various aspects, such as PD causes, consequences, and types. Lastly, they arrived at a formal PD definition. The authors collected the primary input data over an empirical study. They conducted workshops and interviews with 16 practitioners from four companies. They found that process competencies, value neglection, and organizational issues are some of the PD causes. They identified that late deliveries, low software quality and TD, and impeding other processes are the long-term consequences of PD. Various PD types are also revealed, including documentation debt, mismatched role debt, process synchronization debt, and infrastructure debt. A holistic evaluation of the results led to the following PD definition in the study: "PD is the occurrence of sub-optimal process design, divergence from optimal process or deficiencies in the infrastructure that might

be beneficial in the short term. However, such issues might cause both short-term waste and might create a context in the long term where a high negative impact is suffered by the process stakeholders.".

In our study, we utilize the definition in [70] and consider that code review process smells create debt which in turn becomes a process debt.

Chapter 3

Background

This section provides an overview of the code review process smells and the technical debt metaphor we use in our study.

3.1 Pull Requests

Pull request in GitHub is a proposed set of source code changes to be added to the code repository, usually aligned with user stories, tasks, or bug reports. Pull requests notify other developers working on the same project about the potential changes. Developers can discuss and review the proposed changes in an open pull request.

3.1.1 Roles

Generally speaking, we can classify the developer roles into three in the context of pull requests as below.

- *Participant:* A developer conducting any activity[1] on a pull request.
- *Author:* The developer opening the pull request, i.e., preparing the source code change proposal.
- *Reviewer:* The developers reviewing the code using the review functionality on GitHub.

We note that the pull request *author* and *reviewer* are also the pull request *participants*.

3.1.2 Code Review Process

In a pull request, the author and reviewers participate in the *code review process*. Code review is the inspection of the set of changes proposed in the pull request. Reviewers may ask for clarification or additional modifications upon inspecting source codes. If so, the author is expected to provide explanations or commit changes compliant with the reviewers' expectations. If reviewers are satisfied, they approve the pull request for the merging operation. However, it is also possible that a pull request could be closed without merging, meaning the abandonment of the efforts.

In this thesis, we are specifically interested in the code review process and its actors.

3.2 Code Review Process Smells

In this section, we provide a brief overview of CRPSs based on the work of Doğan and Tüzün [27]. Code review process smells are the classification of bad practices followed in the code review process. Dogan and Tüzün [27] proposed

[1]A developer can conduct many activities on a pull request, such as commenting, assigning reviewers, labels, and others. The variety of these activities is not within the scope of the thesis.

seven CRPSs. The analyses in our study involve the *lack of code review*, *large changesets*, *missing context*, *ping-pong*, and *sleeping reviews* CRPSs. We excluded the *looks good to me reviews* smell because there is no well-established detection mechanism for the smell. Similarly, we excluded the *review buddies* smell because it is detected on aggregate contributions of author-reviewer pairs and so it cannot be evaluated on a pull request basis.

Lack of Code Review. The code review process provides various benefits in a software development project, including assuring readability and consistency [29], serving as a medium for teaching and discussions [2], and reducing code smells [41]. The benefits are subject to degradation or are even lost if reviews are not conducted properly. Pull requests identified with the *lack of code review (lack of review)* process smell are either not reviewed or self-reviewed by the pull request author before merging to the codebase.

Large Changesets. If a pull request consists of more than 500 changed lines of code (LOC), it has the *large changesets* smell. Developers may avoid reviewing a pull request if it has a large number of changed LOCs, or they may not sufficiently review the change coverage.

Looks Good to Me Reviews. The code review process has sound benefits, as mentioned earlier if conducted properly. Reviewers may conduct the code review activity without paying enough attention. Such pull requests suffer from *looks good to me reviews*. This smell breaks the feedback loop between the author and reviewers and indicates a poor review.

Missing Context. To develop high-quality software efficiently, developers should maintain traceability between various software artifacts [71]. In particular, pull requests should include a link to the related issue in the issue tracking system whenever possible. In other cases, context can be provided by a pull request title and description that provide sufficient detail to reviewers. Bacchelli and Bird [11] reported that providing context and direction to reviewers of pull requests leads to better code reviews. Lack of a link to an issue or a sufficient description indicates that a pull request has the *missing context* smell. Pull requests with

this smell deprive reviewers of required details and break the traceability among issue, commit, and pull request artifacts.

Reviewer-Author Ping-Pong. Code reviewers can request further modifications from the author, resulting in an iteration between reviewers and authors until the reviewers are satisfied, and then the pull request can be merged. However, if the loop between the author and reviewers becomes longer than three iterations, then the *reviewer-author ping-pong (ping-pong)* smell is present. The smell could lead to increased review time and other developers being blocked if they are depending on the pull request.

Review Buddies. There are many studies on the reviewer recommendation problem [72, 73] and its relevance to developers and their perceptions of the problem [74]. The review effort should be balanced among team members in order to disperse code knowledge. If the same author(s) and reviewer(s) team up frequently for code reviews, then this creates a *review buddies* smell. Buddy relationships could cause ineffective reviews and decrease shared code ownership.

Sleeping Reviews. Practitioners usually try to complete code reviews of pull requests in a timely manner. The review time in companies such as Google, Microsoft, and AMD is reported to vary from 4 hours [29] up to almost 21 hours [2]. A review time of more than 48 hours indicates the *sleeping reviews* smell. Such pull requests may cause merge conflicts if other developers work on the files under review. Also, remembering the changed content and detail becomes harder for the pull request author as the review takes a longer time.

3.3 The Technical Debt Metaphor

Technical Debt (TD) is a set of design and implementation choices that bring ad hoc benefits but incur costly changes in the future [65]. Code debt is one defined type of TD [75], which "refers to the problems found in the source code (poorly

written code that violates best coding practices or coding rules) that can negatively affect the legibility of the code, making it more difficult to maintain.“ [76]. Code debt is usually incurred by introducing code smells to the code repository and is typically identified by the presence of code smells.

TD is based on a metaphor (based on financial debt) that intuitively fits many software engineering-related phenomena. One such phenomenon is process debt, which is defined as deficiencies in the definition or implementation of some software development process that could have negative impacts on the project that appear later in time. Code review process smells are a type of process debt [27] that applies specifically to the code review process. Process debt has been discussed in the context of TD, but not as a type of TD [70]. Instead, it is seen as a phenomenon that is a precursor to TD. In this thesis, we take the perspective that process debt (in the form of code review process smells) can be a precursor to technical debt (in the form of code smells). It is this causal relationship that we investigate in this work.

Research on TD management often attempts to define the principal and interest associated with the debt being studied, extending the analogy to the principal and interest associated with financial debt. The principal refers to the cost (usually effort, but sometimes other costs) that was avoided by incurring the technical debt. For example, the principal associated with a particular class that has code smells is in theory the effort saved by not taking the time and effort required to develop or maintain that class correctly, i.e. in a way that results in clean, smell-free code. In practice, however, estimating the time saved in situations like this is difficult if not impossible, so the proxy typically used is the cost of remediating the debt. In our example, this would be the cost of refactoring the class to remove the code smells. While it is difficult to estimate this cost a priori, as all software cost estimation is, one can develop estimates that are adequate for planning purposes.

TD interest, on the other hand, is extremely difficult and complicated to capture. Some work by TD researchers has attempted to do this in certain specific conditions, but in general, the estimation of TD interest has been elusive.

Process debt, although not actually a type of technical debt, still is consistent with the debt metaphor and so it makes sense to talk about the principal and interest associated with it. The principal, in this case, would represent the time saved by carrying out a process in a sub-optimal way. In our context, the principal associated with CRPSs would be the time saved by each code review that exhibits a smell, as compared to how long that review would have taken if it had been done properly. As with TD, it makes sense to think about this as the extra time that code reviews will take once the process smell is remedied, i.e. once the development team starts performing code reviews without the smell. But this presents a conundrum, in that the debt metaphor does not quite fit. With TD, as with financial debt, principal is "paid back" just once. The loan is repaid, or the code is refactored, and the debt disappears. With process debt, specifically, the code review smells we are studying, the principal must be paid each time a code review takes place. This complicates any reasoning about the relative costs and benefits of "paying back" process debt, as the cost of paying it back extends forever into the future.

In this study, however, we are addressing the interest associated with process debt, specifically code review process smells. Because process debt is not technical debt per se, but a precursor to it, the interest associated with process debt is not cost, but the technical debt itself, i.e. the technical debt that the process debt causes. In our study, we use code smells as an indicator of the TD that the code review smells lead to. Thus, we are studying the relationship between code review smells and code smells, and investigating whether the former leads to the latter. The implications of our findings inform an understanding of the interest associated with this type of process debt.

Chapter 4

Key Analytical Concepts

We established some definitions and concepts used in our analysis procedures in this section. Each subsection develops direct or related background concepts for the research questions. Section 4.1 and Section 4.2 concern all research questions. Section 4.3 is related to RQ2 and RQ4. Lastly, Section 4.4 relates to RQ3 and RQ4.

4.1 Key Commit Types

Key commits are the beacons of a pull request process for evaluating the change of code smells. We identified three key commit (KC) types in a typical pull request (PR) based on the point in the development process when the commit takes place. These three types, and when they occur, are shown in Figure 4.1. Our analysis depends on measuring the code smell counts and code smell density at the time of each key commit in each pull request. We used c_p^x notation to represent the key commit where x is the key commit type, and p is the pull request containing the commit. In the following, we explain each key commit type.

The first is the *base* commit (c_p^{base}). Developers create a new branch or fork from this commit to change the code files of the repository. This commit is

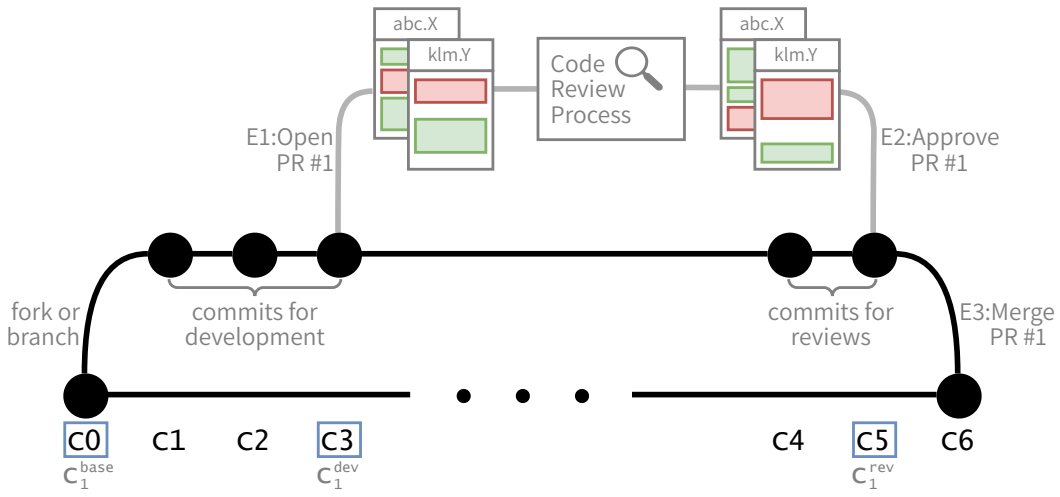


Figure 4.1: Sample development overview. Commit history graph shows the sequential progress of commits, i.e., `c0`, ..., `c6`. Pull request events are represented with E1, E1, and E3.

visualized with `c0` in Figure 4.1.

The second key commit type is the *development* (*dev*) commit (c_p^{dev}). This is the last commit in a pull request[1] before any code review activity on the subject pull request. Reviewers usually review the changes up to and including this commit by checking the additions, deletions, or modifications made in the repository compared to the repository state in c_p^{base}. The *dev* commit of the sample analysis is `c3` in Figure 4.1.

The last key commit type is the *review* (*rev*) commit (c_p^{rev}). This commit contains any modifications requested by reviewers on top of c_p^{dev} and indicates the end of development on the branch or fork. In Figure 4.1, `c5` represents the *rev* commit.

The three key commits allow us to assess and observe the change in the numbers and types of code smells over the life of a pull request. The c_p^{base} and c_p^{dev} commits exist for all pull requests. However, c_p^{rev} commit may not exist in a pull request if there is no change request from the pull request reviewers or the

[1]We refer to commits made on a branch or fork as *pull request commits*.

pull request author decides not to implement the change requests. So the sequence of key commits in a pull request's timeline could either be $c_p^{base} \rightarrow c_p^{dev}$ or $c_p^{base} \rightarrow c_p^{dev} \rightarrow c_p^{rev}$.

4.2 Delta Analysis of Code Smells

We quantified the relative change in code smells using two different *delta analyses* during the lifetime of a pull request. The first one is ***D**elta Analysis of **Co**de **S**mell **C**ounts* (ΔCoS-C analysis). This analysis uses absolute smell counts and measures the change between two KCs. On the other hand, the second delta analysis normalizes the smell counts in KCs by the total number of lines in files affected by the pull request and is called ***D**elta Analysis of **Co**de **S**mell **D**ensities* (ΔCoS-D analysis). We provide a thorough explanation of the analyses below.

4.2.1 Common Concepts

We start by defining the common terms and concepts of the two analyses before describing their details.

In the scope of a pull request, from the code smells perspective, one can compare the states of a repository at two different commits. Accordingly, we defined three comparison types, each of which is defined by the two KC types being compared. More formally, let *comparison* (δ) represent the comparison of a delta analysis parameter (i.e., count or density) calculated on different KCs of a pull request. Considering the sequential nature of KCs in a pull request p, we established three different δ comparisons as below.

- The *development* (*dev*) comparison denotes the change of a delta analysis parameter from commit c_p^{base} to c_p^{dev},

- The *review* (*rev*) comparison denotes the change of a delta analysis parameter from commit c_p^{dev} to c_p^{rev},
- The *total* (*total*) comparison denotes the change of a delta analysis parameter from commit c_p^{base} to commit c_p^{rev} (or c_p^{dev}, if c_p^{rev} does not exist in p).

Also, we defined $\kappa^c(f)$ to represent the total number of code smells in file f in commit c.

4.2.2 ΔCoS-C Analysis

ΔCoS-C analysis expresses the change in the absolute code smell counts between two key commits. Below, we establish the analysis formally.

Let $K(c)$ represent the summation of $\kappa^c(f)$ values, i.e., total number of code smells, over all the files included in the commit c, as in (4.1)

$$K(c) = \sum_{i=1}^{n} \kappa^c(f_i) \tag{4.1}$$

where n is the number of files in the commit.

We adapt the formula and notation in (4.1) to pull request context. In the context of a pull request p, n is the number of files edited by the pull request. Similarly, for pull request p, we can refer to $K(c)$ as $K(c_p^x)$, where x denotes the key commit type, i.e., *base*, *dev*, or *rev*. This representation allows us to compare the $K(c_p^x)$ values of a pull request at the three different key commits. To do so, we define ΔK_p^δ to measure the relative change in K values of a pull request p, where δ indicates the comparison type, i.e., *dev*, *rev*, or *total*. We show the formulae of ΔK_p^{dev}, ΔK_p^{rev}, and ΔK_p^{total} in (4.2), (4.3), and (4.4), respectively.

$$\Delta K_p^{dev} = \left(\frac{K(c_p^{dev}) - K(c_p^{base})}{K(c_p^{base})} \right) * 100 \tag{4.2}$$

$$\Delta K_p^{rev} = \left(\frac{K(c_p^{rev}) - K(c_p^{dev})}{K(c_p^{dev})} \right) * 100 \tag{4.3}$$

$$\Delta K_p^{total} = \left(\frac{K(c_p^{rev}) - K(c_p^{base})}{K(c_p^{base})} \right) * 100 \tag{4.4}$$

Each of these formulae represents the percentage increase (or decrease) in the number of code smells in a pull request from one key commit to the other. We explain the conceptual correspondings of the mathematical values of ΔK_p^{dev}, ΔK_p^{rev}, and ΔK_p^{total} in Appendix A.

4.2.3 ΔCoS-D Analysis

ΔCoS-D analysis measures the change in the code smell density between two key commits. Below, we establish the analysis formally.

We use $\lambda^c(f)$ to represent the number of lines in file f in commit c. Code smell density ($\mu^c(f)$) is the number of code smells per line for the file f in commit c as in (4.5).

$$\mu^c(f) = \frac{\kappa^c(f)}{\lambda^c(f)} \tag{4.5}$$

Then, $M(c)$ is the code smell density over all files in commit c as in (4.6)

$$M(c) = \frac{\sum_{i=1}^{n} \kappa_c(f_i)}{\sum_{i=1}^{n} \lambda_c(f_i)} \tag{4.6}$$

where n is the number of files.

Similar to $K(c_p^x)$, we adapt (4.6) to a pull request context. For pull request p, n accounts for the files edited by p and we can write $M(c)$ as $M(c_p^x)$. Then ΔM_p^{δ} represents the relative change in the M parameter for the key commits of

the pull request p. We present ΔM_p^{dev}, ΔM_p^{rev}, and ΔM_p^{total} in (4.7), (4.8), and (4.9), respectively.

$$\Delta M_p^{dev} = \left(\frac{M(c_p^{dev}) - M(c_p^{base})}{M(c_p^{base})} \right) * 100 \tag{4.7}$$

$$\Delta M_p^{rev} = \left(\frac{M(c_p^{rev}) - M(c_p^{dev})}{M(c_p^{dev})} \right) * 100 \tag{4.8}$$

$$\Delta M_p^{total} = \left(\frac{M(c_p^{rev}) - M(c_p^{base})}{M(c_p^{base})} \right) * 100 \tag{4.9}$$

Each of these formulae represents the percentage increase (or decrease) in the code smell density in a pull request from one key commit to the other. We explain the conceptual correspondings of the mathematical values of ΔM_p^{dev}, ΔM_p^{rev}, and ΔM_p^{total} in Appendix A.

4.2.4 Edge Cases in Delta Analyses

Here, we have handled edge cases in both delta analyses, i.e., ΔCoS-C and ΔCoS-D.

When c_p^{rev} does not exist. As mentioned, c_p^{rev} may not exist in all pull requests. In such cases, (4.3), (4.4), (4.8), and (4.9) are affected. We cannot calculate the values of the ΔK_p^{rev} and ΔM_p^{rev} parameters. So, actually, (4.3) and (4.8) are undefined ($Null$), but, effectively, they are 0. On the other hand, ΔK_p^{total} and ΔM_p^{total} can still be calculated. Conceptually, the *total* comparison expresses the change in code smells from the beginning of a pull request to the end. Thus, it is legitimate to replace c_p^{rev} in (4.4) and (4.9) with c_p^{dev} and calculate ΔK_p^{total} and ΔM_p^{total} parameters accordingly. As a consequence, we obtain the following when c_p^{rev} does not exists:

- $\Delta K_p^{rev} = 0$ and $\Delta K_p^{total} = \Delta K_p^{dev}$.

- $\Delta M_p^{rev} = 0$ and $\Delta M_p^{total} = \Delta M_p^{dev}$.

When the denominator is zero. Division with zero occurs in (4.2), (4.3), and (4.4) when $K(c_p^{base})$ or $K(c_p^{dev})$ is zero, i.e., no code smells exist. As a remedy, we add 1 to terms of interest in (4.2), (4.3), and (4.4). To demonstrate the effect of this modification, let us assume that the files of interest had zero smells initially ($K(c_p^{base}) = 0$). During development, we introduced five code smells ($K(c_p^{base}) = 5$). We can calculate ΔK_p^{dev} with $((5+1) - (0+1))/(0+1) * 100$ as 500% increase. The same edge case exists in ΔCoS-D parameters as well. We employ the same remedy technique for (4.7), (4.8), and (4.9) and demonstrate a numerical example. Suppose that $M(c_p^{dev}) = 0$ and $M(c_p^{rev}) = 0.1$. Then, ΔM_p^{rev} can be calculated with $((0.1+1) - (0+1))/(0+1) * 100$ as 10% increase. The solution prevents discarding pull requests with this edge case because the delta analysis parameter cannot be calculated.

4.3 Accumulation of Code Review Process Smells

ΔCoS-C and ΔCoS-D analyses help us assess the potential effect of CRPSs on code smells in files touched by a single pull request. However, to address RQ2, we need an aggregate analysis of the cumulative effect of CRPSs on code smells in a file over all the pull requests in which the file has ever been involved. Therefore, we look at files that have, over their history, been subject to different numbers of smelly reviews to see if the level of smelliness in the code reviews has a relationship with the level of code smells in the file. Below, we provide a formal explanation.

We know that a file in a code repository can be changed by multiple pull requests. Thus, we can aggregate all the pull requests that ever caused an edit to any particular file f. More formally, let the set $APR(f)$ represent *all pull requests* (APR) that changed file f. Further, let the set $SPR^{crps}(f)$ account for the *smelly pull requests* (SPR) with a particular *crps* that edited file f (where *crps* could

be any one of the five CRPS types we examine in this study, as described in Section 3.2. We note that $SPR^{crps}(f) \subseteq APR(f)$. Using both of these terms, we define the *CRPS density* of a file f, i.e., $\rho^{crps}(f)$, using the cardinality of these sets as in (4.10).

$$\rho^{crps}(f) = \frac{|SPR^{crps}(f)|}{|APR(f)|} \tag{4.10}$$

Intuitively, $\rho^{crps}(f)$ represents the fraction of all pull requests affecting file f in which CRPSs were detected.

The CRPS density $\rho^{crps}(f)$ is an indicator of the cumulative effect of code review process smells on file f.

4.4 Classifying the Code Review Performance

We can view the increases or decreases in code smells after a code review as performance indicators. Thus, we classified the code review performance of a pull request according to the changes in code smells before, during, and after the code review. For the classification, we decided to use ΔCoS-D analysis parameters instead of ΔCoS-C because using a normalized parameter lets us avoid size-related bias in our analysis.

Conceptual Basis. We intuitively argue that the code review process is expected to *decrease*[2] the number of code smells (or the code smell density) in source code. At least, the review process should keep the code smells as is, i.e., *no change*. In the case of an *increase*, the codebase becomes more smelly, and the code review process fails to act as a gatekeeper from the code quality point of view. In this way, we classify the performance of code reviews that result in decreasing code smells as *positive*, no change is *neutral*, and increasing code smells *negative* during the code review process.

Methodology. We use the ΔM_p^{dev}, ΔM_p^{rev}, and ΔM_p^{total} parameters (that quantify the relative change in the code smell density, as explained in Section 4.2)

[2]We use *increase*, *no change*, and *decrease* as explained in Appendix A.

to classify the code review performance. We first evaluate ΔM_p^{rev} to decide the goodness of the review performance from the code smell perspective. Then, by checking both ΔM_p^{dev} and ΔM_p^{total}, we assess the review process in finer detail. Below, we explain our *code review performance classification* rationale, including the *review classes*.

- Better: When the review process removes the increase in code smell density that occurred during the development of the pull request, and further decreases it below what it was even before the beginning of the pull request development.
- Good: When the review process negates some of the increase in code smell density that occurred during the development of the pull request by decreasing it by some amount or in some cases back to the level before the pull request was developed.
- NWRC[3]: When the review process remains neutral on code smells (does not increase or decrease the code smell density), even though there are commits that occur during the review.
- NWoRC[4]: When the review process remains neutral on code smells without any additional commits during the review, i.e., c_p^{rev} does not exist in the subject pull request.
- Bad: When the review process negates the decrease in code smell density that occurred during the development by increasing it by some amount or in some cases back to the level before the pull request was developed.
- Worse: When the review process removes the decrease in the code smell density that occurred during the development of the pull request, and further increases it above what it was even before the beginning of the pull request development.

[3]Neutral with Review Commit (c_p^{rev}).
[4]Neutral without Review Commit (c_p^{rev}).

Table 4.1: Code review classification logic

ΔM_p^{rev}	ΔM_p^{dev}	ΔM_p^{total}	**Class**
< 0	< 0	< 0	Good
< 0	$= 0$	< 0	Better
< 0	> 0	< 0	Better
< 0	> 0	$= 0$	Good
< 0	> 0	> 0	Good
$= 0$	< 0	< 0	NWRC
$= 0$	$= 0$	$= 0$	NWRC
$= 0$	> 0	> 0	NWRC
$Null$	< 0	< 0	NWoRC
$Null$	$= 0$	$= 0$	NWoRC
$Null$	> 0	> 0	NWoRC
> 0	< 0	< 0	Bad
> 0	< 0	$= 0$	Bad
> 0	< 0	> 0	Worse
> 0	$= 0$	> 0	Worse
> 0	> 0	> 0	Bad

Conceptual correspondings of the mathematical values of ΔM_p^{dev}, ΔM_p^{rev}, and ΔM_p^{total} are in Appendix A.

Table 4.1 shows all potential combinations of ΔM_p^{dev}, ΔM_p^{rev}, and ΔM_p^{total} for a pull request together with the review classes.

Chapter 5

Study Design

In this section, we present the structure of our study. First, we detail the analyses for each research question. Then, we list the criteria we used to select analysis projects. Afterward, we report our data collection methodology. Lastly, we present the details of the focus group study. We summarized the study design in Figure 5.1.

5.1 Analysis

5.1.1 Influence of Code Review Process Smells on Code Quality (RQ1)

We conducted this analysis to investigate whether smelly code reviews result in a more smelly codebase. On the one hand, we detected the existence of each CRPS type for each pull request of a project. This identification quantifies developers' conformance with the best practices in the code review process. On the other hand, we evaluated the ΔM_p^{dev} (change in code smell density during pull request development) and ΔM_p^{total} (change in code smell density over the entire pull request lifecycle) parameters for each pull request and project using the code

smell results and employing ΔCoS-D analysis. Then, we checked the correlation between the presence of CRPSs and ΔM_p^{total} value for pull requests with $\Delta M_p^{dev} > 0$. We expect that the code review process should reverse the increase in code smells that occurred during the pull request preparation ($\Delta M_p^{dev} > 0$), i.e., ΔM_p^{total} should be close to 0. We also want to investigate whether or not the presence of a CRPS is related to whether or not the code review process is able to reverse the code smell increase. We conducted the same investigation using ΔCoS-C analysis as well. We present the employed correlation methods in Section 5.1.3.

5.1.2 Accumulated Process Smells (RQ2)

We designed this analysis to quantify the accumulated effect of CRPSs on code smells over the pull request history of files. For each file in a project repository, we calculated the $\rho^{crps}(f)$ (CRPS density) and $\mu^c(f)$ (code smell density) parameters. We determined c in $\mu^c(f)$ to be the merge commit of the latest pull request of each project in our dataset, and we share the used latest commits in our online replication package [77]. We note that while finding $\rho^{crps}(f)$ of every file, we took into account file renamings in the repository history. Lastly, we found the correlation between $\rho^{crps}(f)$ and $\mu^c(f)$ to answer RQ2 with the correlation methods in Section 5.1.3.

5.1.3 Correlation Methods

We employed two complementary correlation methods in the analyses of our research questions. For both methods, we chose the p-value limit as 0.05 for statistical significance and used the interpretations in [78] for the numerical values of the correlation coefficients.

As a classical correlation method, we used *Spearman's Correlation (SC)* as it has no assumptions about the data distribution [79]. The method is also used in

Table 5.1: Characteristics of selected projects

Project	Organization	NCLOC	Number of Files	Number of PRs	Number of Closed PRs	Number of Merged PRs
ANGULAR.JS	ANGULAR	231,652	1,118	7,999	7,922	789
CESIUM	CESIUMGS	687,138	2,947	5,530	5,507	4,876
ESLINT	ESLINT	355,030	1,222	6,314	6,299	5,180
GHOST	TRYGHOST	110,236	1,256	8,585	8,560	6,624
OPENLAYERS	OPENLAYERS	140,170	931	8,242	8,219	7,203
PDF.JS	MOZILLA	130,135	320	6,754	6,738	5,749
REACT	FACEBOOK	356,849	1,807	12,357	12,074	8,358
SHIELDS	BADGES	70,312	1,155	5,893	5,862	4,887
STRAPI	STRAPI	215,186	2,564	6,114	6,048	4,768
WEBPACK	WEBPACK	152,330	5,938	5,784	5,650	4,169
	Total	2,449,038	19,258	73,572	72,879	52,603

many other software engineering papers [80].

A recent study by Chatterjee [81] introduced a new correlation method. Chatterjee listed several properties of the new *Chatterjee's Correlation (CC)*. CC, being an asymmetric correlation method, uses the notation $CC(X, Y)$ to express the relationship between X and Y. We decided to use CC alongside of SC for two reasons. First, unlike SC, CC can capture non-linear relationship between X and Y. Second, like SC, CC is robust against different data distributions.

For both correlation methods, we utilized the implementations in R [82, 83].

5.1.4 Classifying the Effect of the Code Review Process (RQ3)

As explained in Section 5.1.1, RQ1 analysis calculates ΔM_p^{dev}, ΔM_p^{rev}, and ΔM_p^{total}. We used those calculated parameters per pull request and classified each code review to the related review class as in Section 4.4.

5.2 Case Project Selection

We selected 10 projects to conduct the analyses described in Section 5.1. We used the following criteria in the selection of the projects:

1. The project uses GitHub to host their Git version control system and code review platform.
2. The project has 3,000 or more stars. Amantanidis et al. [84] also used this threshold to select subject projects.
3. The project has 5,000 or more closed pull requests.
4. The project's main language is JavaScript, which constitutes at least 90% of the repository code.
5. The project is owned by an organization, i.e., a community or a company.

We wrote a script to retrieve candidate projects and relevant numerical data. The numerical data contains the number of stars, the number of closed pull requests, and language contributions per project. We used the PyGitHub[1] library to obtain the data from GitHub API[2]. After retrieving the top 1000 projects, we sorted the projects with respect to the number of stars and filtered them based on the selection criteria. The filtration process resulted in 11 projects, and we selected the top 10 projects. Table 5.1 shows the projects together with their characteristics. We share the list of selected projects with the numerical details in our online replication package [77].

5.3 Data Collection

We detail the data collection processes for RQ1 and RQ2 in this section. RQ1 depends on three types of data per project: the KCs of pull requests, SonarQube

[1] `https://github.com/PyGithub/PyGithub` (Accessed on 19 Dec 2022)
[2] `https://docs.github.com/en/rest` (Accessed on 19 Dec 2022)

code smell data, and the CRPSs associated with each pull request. On the other hand, to explore RQ2 we used CRPS results aggregated per code file and SonarQube code smell data. We describe the details of how these data were obtained in the following subsections.

5.3.1 Detection of Code Review Process Smells

Both RQ1 and RQ2 depend on the CRPS detection results. CRPSs are detected for each pull request which requires obtaining the pull request data of the projects. To download this data, we used the Perceval [85] tool. We relied on the algorithms from Doğan and Tüzün [27] to detect CRPSs at the pull request granularity. We saved the smell results for each project in a PostgreSQL[3] database. We share the SQL dump of the resulting database in our online replication package [77].

The analysis of RQ2 depends on identifying $APR(f)$ and $SPR^{crps}(f)$ for any file f. Thus we created a dataset that contains $APR(f)$ and $SPR^{crps}(f)$ information for each CRPS and file of the project repositories by relating pull requests with the files edited in them. We share the resulting dataset as JSON files in our online replication package [77]. We note that a file could be removed at any time in the history of a repository, so the shared JSON files may include files not existing in the latest commit of the project repositories. We used files existing in the latest state[4] of the repositories in the RQ2 analysis.

5.3.2 Identifying Key Commits

To answer RQ1, we conducted delta analyses. Running a delta analysis requires identifying each pull request's key commits, i.e., c_p^{base}, c_p^{dev}, and c_p^{rev}. This process depends on commit and review data of pull requests.

[3] `https://www.postgresql.org/` (Accessed on 19 Dec 2022)

[4] As mentioned in Section 5.1.2, *latest state* refers to the merge commit of the latest pull request in our dataset.

Algorithm 1 Identifying *dev* Commit

```
1: function GetDevCommit(pr : GitHubPullRequest)
2:     commits ← pr.getCommits()                                  ▷ list
3:     commitDates ← pr.getCommitDates()                          ▷ list
4:     reviewDates ← pr.getReviewDates()                          ▷ list
5:                          ▷ all lists are sorted w.r.t. activity timestamps
6:     devCommit ← Null
7:     if reviews.length() = 0 then
8:         devCommit ← commits.lastItem()
9:     else
10:        firstReviewDate ← reviewDates[0]
11:        for i ← 0, commitDates.length() − 1 do
12:            if commitDates[i] > firstReviewDate then
13:                devCommit ← commits[i − 1]
14:                break
15:            else if i = commitDates.length() − 1 then
16:                devCommit ← commits.lastItem()
17:    return devCommit
```

Identifying *base* Commit. The parent of the first commit in a pull request corresponds to the *base* commit. If all pull request commits are available in a project repository's Git history, c_p^{base} can be easily retrieved with Git commands, and we used the GitPython[5] library to retrieve them. However, this is not always the case, as we found that some pull request commits are orphaned (not referenced), and orphaned commits are deleted by the Git garbage collector [86]. But even in this case, the data is preserved on GitHub, so we were able to use the GitHub API to remedy this problem.

Identifying *dev* Commit. The last commit before the first review activity in the pull request is the *dev* commit, c_p^{dev}. We identify this commit by analyzing the commit and review dates of a pull request p. If there is no review activity or no commits after review, the c_p^{dev} is the last commit of a pull request. Algorithm 1 summarizes this process. We note that a few pull requests do not contain any commit information in either the GitHub web interface or the API. We could not detect c_p^{dev} of such pull requests and discarded them from the analysis[6].

[5] `https://github.com/gitpython-developers/GitPython` (Accessed on 19 Dec 2022)

[6] Discarded PRs constitute $<$ %0.05 of the total merged PRs.

Table 5.2: Analytics of pull requests with key commit identifiers.

Project	**Number of PRs with KCs**	**Number of PRs with c_p^{rev}**	**Ratio (%)**
ANGULAR.JS	703	88	12.5
CESIUM	4,314	1,266	29.3
ESLINT	5,167	833	16.1
GHOST	6,159	374	6.1
OPENLAYERS	7,083	562	7.9
PDF.JS	5,724	240	4.2
REACT	8,168	1,551	19.0
SHIELDS	4,861	2,970	61.1
STRAPI	2,943	1,404	47.7
WEBPACK	3,582	550	15.4
All	48,704	9,838	20.2

Readers may notice that the merged PR numbers in Table 5.1 and the number of PRs with KCs in Table 5.2 differ. The reason for the difference is due to the file eligibility criteria we employed for the edited files in pull requests. We explain the file filtration process in Section 5.3.3.

Identifying *rev* Commit. This commit type can be identified in a pull request only if review activities and commits exist after the *dev* commit. When this condition does not hold, we considered the c_p^{rev} to be $Null$. Otherwise, the last commit of the pull request after c_p^{dev} corresponds to c_p^{rev}.

Key Commits of Pull Requests. For each project, we identified key commits of pull requests. Those pull requests with identified key commits are called *pull requests with key commits*. For convenience, we further classify the pull requests with a valid c_p^{rev}, i.e., not $Null$ value, and call them as *pull requests with* c_p^{rev}. Table 5.2 shows the number PRs with KCs and PRs with c_p^{rev}. We observe that PRs with c_p^{rev} constitute the minority among all PRs with KCs for all projects except SHIELDS and STRAPI. We share the key commit data of pull requests in JSON format for each project in our online replication package [77].

5.3.3 Finding Code Smells

We detected code smells for RQ1 and RQ2 analyses using SonarQube. For RQ1, we found code smells in files modified in the pull requests of each project. On the other hand, for RQ2, we detected the code smells of files in the latest states of the project repositories. We provide a brief overview of the SonarQube tool and explain how code smells are detected below.

Detection Tool. We relied on SonarQube[7] [87] to find code smells for our analyses. It is a widely used tool in industry [88] and in academic research [89, 38, 90]. The tool is used in practice to monitor software quality and maintainability. SonarQube scans source code components (files and directories) and computes various metrics, such as number of non-commented lines of code, as well as code smells [91] based on defined rule sets [92]. It can also classify the severity of a code smell according to a 5-level scale [93, 38]. We excluded from our analysis code smells with the lowest severity level, "Info".

Finding Code Smells in a Pull Request. We found the code smells in a pull request at the identified KCs. For each KC, we gathered the files (by preserving the directory structure) affected by the pull request. The gathering process was not straightforward. As discussed in Section 5.3.2, the Git garbage collector removes the data of orphaned commits (branches or forks), which means the file states are also deleted. Consequently, we could not obtain the file states for some pull requests for *dev* or *rev* key commits. Again, however, we found that a file state in any commit of a repository can be obtained from GitHub's raw file service.[8] So similarly, we employed a two-way strategy to collect files in the KCs, using the local Git repository if the file is available there, otherwise sending a `HTTP GET` request to GitHub's raw file service to retrieve the file.

We further filtered the files in pull request KCs based on eligibility criteria. The eligibility criteria serve to develop a consistent code smell evaluation strategy,

[7] `https://www.sonarqube.org/` (Accessed on 19 Dec 2022).
We used community edition, version 9.5 (build 56709).

[8] `https://raw.githubusercontent.com/owner/repo/commit/filePath` provides the file in a repository in the specified commit.

eliminate irrelevant files from the analysis, and shorten the analysis duration. The criteria are as follows:

1. The file is modified by the pull request. We considered only the files edited in the pull request.
2. The file exists among all key commits. This criterion implies that if a file is newly added, deleted, or renamed[9] in one of the key commits, it is ignored. This enables us to evaluate files with respect to a reference state.
3. The file should have a JavaScript extension. We used SonarQube's default JavaScript extensions `.js`, `.jsx`, `.mjs`, and `.vue`.

As we noted earlier, some pull requests are eliminated due to the filtering above because no files of the pull request satisfied the eligibility criteria. We ran SonarQube on the eligible files of each pull request of all the projects. SonarQube identifies the code smells and provides the results over an API[10]. We used this API to get the code smell results per file and key commit. We saved the query results to a PostgreSQL database table. This data is used in the analysis for RQ1. In our online replication package [77], we share the SQL dump of the resulting database.

Finding Code Smells in a Project Repository. To answer RQ2, we ran SonarQube on the latest state in our dataset of each project repository. By utilizing the SonarQube API, we extracted the code smell data of each file and saved the results in JSON format. We share the resulting JSON files for each selected project in our online replication package [77].

[9]The pull request data contains the new path and name of the file but not the old path and name.

[10]`https://next.sonarqube.com/sonarqube/web_api/api/issues/search` (Accessed on 19 Dec 2022)

5.4 Focus Group Study

In order to better interpret the quantitative results from RQ1, RQ2, and RQ3 and increase our data diversity [94], we carried out a focus group study with practitioners from the software industry. Using qualitative data from focus groups helps us increase the depth of our study [95] and also helps triangulate our quantitative results with practitioners' views and discussions.

5.4.1 Design

Here, we describe the design of a preliminary survey and the outline of the focus group sessions.

Preliminary Survey. Prior to the focus group sessions, we sent a short online survey to the participants to gather their unbiased views (i.e., their views before seeing our quantitative results) on the relationships between code review process smells and code smells based on their perceptions and experiences. Mainly, the survey consisted of two focus group questions (FQs) for each CRPS, as presented below.

FQ1: Do you agree that this process smell could hinder benefits of the code review process?

FQ2: In your opinion, how likely is it that code reviews with this process smell would lead to an increase in code smells?

We used the survey results to initiate discussion during the focus group sessions.

Session Outline. We followed the guidelines of Krueger and Casey [96] to develop our focus group session outline. We started the sessions with *opening questions*, asking the participants to describe the code review process in their workplaces. Afterward, with *transition questions*, we asked about their general

Table 5.3: Demographic information of the focus group participants

ID	Session	Current Role	Years of Experience		Current Company		
			Total	Current Company	ID	Company Size	Team Size
P1	Virtual	Architect	16+	1-2	C1	151-500	2-3
P2	Virtual	Engineering Manager	16+	<1	C2	501+	4-7
P3	Virtual	Software Engineer	11-16	6-10	C3	501+	21+
P4	Virtual	Engineering Manager	11-16	11-16	C4	51-150	21+
P5	Virtual	Software Engineer	3-5	1-2	C5	501+	2-3
P6	In-Person	Architect	16+	16+	C3	501+	8-10
P7	In-Person	Architect	16+	1-2	C6	501+	21+
P8	In-Person	Software Engineer	16+	11-16	C3	501+	8-10
P9	In-Person	Team Lead	6-10	3-5	C8	151-500	11-20
P10	In-Person	Software Engineer	1-2	1-2	C7	51-150	4-7

SD = Software Development

impressions and thoughts about the CRPSs presented in the survey. Then we moved on to the *key questions*. We presented the survey results for each CRPS, asking participants to expand on their survey responses on FQ1 and FQ2. Finally, we presented our quantitative results from analyzing open-source projects and asked for their opinions.

We provide the preliminary survey and the focus group session outline in our online replication package [77].

5.4.2 Methods

After deciding to conduct a focus group study, we obtained approval from the Ethics Committee of Bilkent University. We employed a purposeful sampling strategy [95], in which we tried to diversify the participants' years of experience and company domain. We contacted developers in the authors' professional circle and selected 10 participants based on availability. Table 5.3 shows the characteristics and demographics of the focus group participants.

We conducted two focus group sessions, each lasting two hours. One session

was virtual, and the other was an in-person meeting. The sessions involved only the co-moderators (first and third authors) and study participants, similar to [97]. Topic coverage was the first author's responsibility whereas the third author assured the session followed the determined outline. We allowed the participants to engage and comment until we reached data saturation, i.e., no new insights were captured. We recorded each session in video and audio formats. The moderators also wrote down field notes during the sessions.

5.4.3 Data Analysis

We transcribed the audio recordings to text using Microsoft Word's transcription feature [98]. The moderator authors proofread the transcriptions and then translated them into English. Even though we video-recorded the sessions, we did not utilize the recordings because we could identify the participants just by using the audio recording.

We analyzed the transcripts using a fairly lightweight coding approach based on our research questions, particularly RQ4. We started with a short list of preformed codes (namely CRPS, Code Smells, and the relationship between them). While reviewing and coding the transcripts, these codes were expanded and refined as themes emerged. Finally, the text tagged with each code was reviewed as a whole to identify emerging themes. We used the Taguette[11] tool to manage our coding.

[11] `https://www.taguette.org/` (Accessed on 19 Dec 2022)

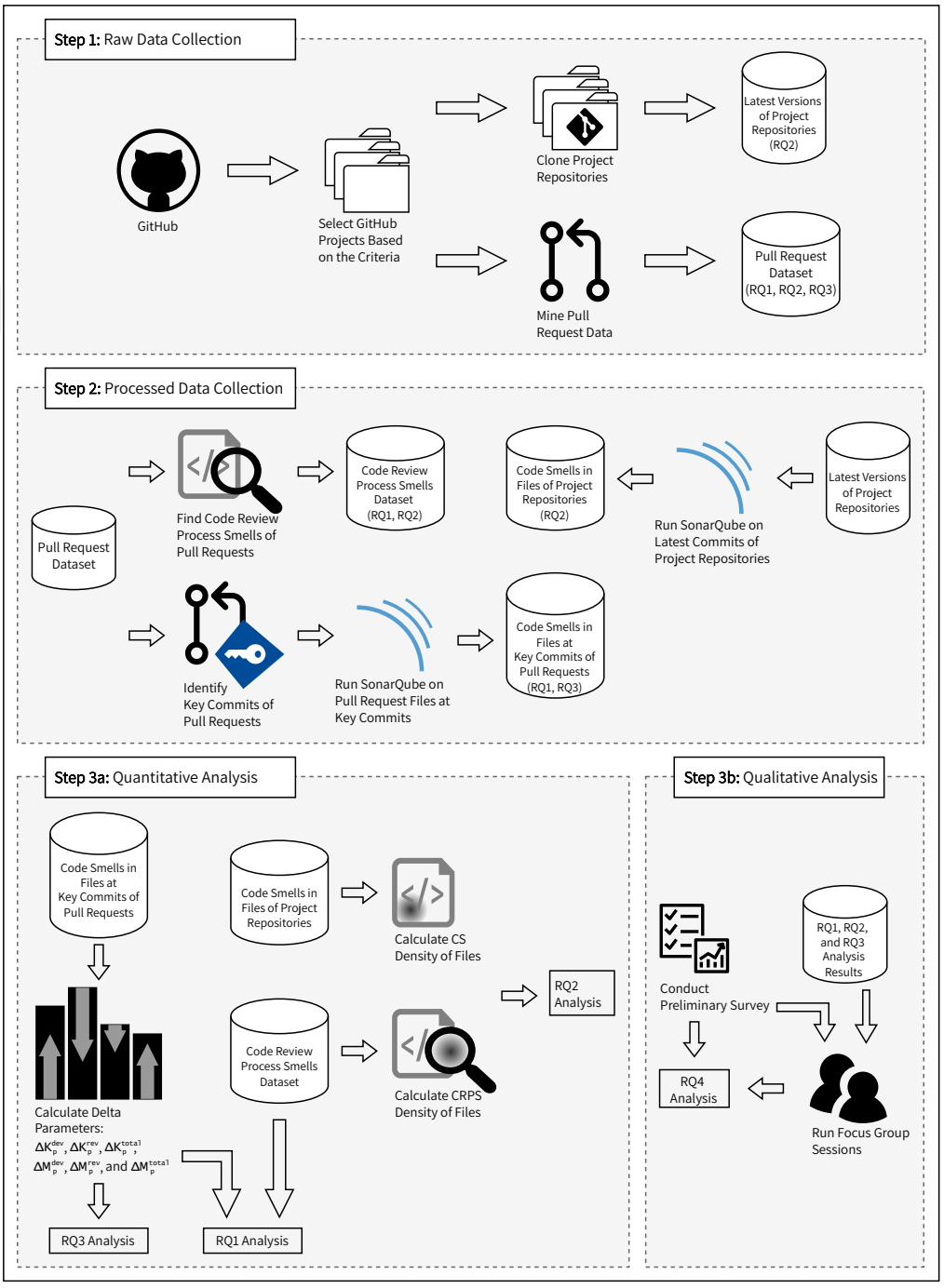


Figure 5.1: Simplified overview of our study design. Some connections and steps are omitted for clarity. Inspired from [1].

Chapter 6

Results

This section presents the results of the analysis we conducted for all research questions.

6.1 Prevalence of Code Review Process Smells

We detected the CRPSs in all projects in order to address both RQ1 and RQ2. Complete data on the occurrence of code review process smells in the projects is presented in Table 6.1. The table contains the results from two perspectives: for all pull requests and pull requests with KCs. We provide two views to show that CRPS prevalence does not change considerably between them. In our study, we focus on the pull requests with KCs because our analysis depends on the KCs of pull requests. We look at the results for each process smell more closely below.

The *lack of review* smell occurs in as few as 7% of pull requests in the SHIELDS project and up to 77% in GHOST. This represents the largest gap between the lowest and highest ratios of all the CRPSs. The *large changesets* smell prevalence varies from 2.6% to 16.0%. SHIELDS has the lowest occurrence again, whereas CESIUM's pull requests are more likely to contain this smell than any of the others. *Missing context* smell occurrence values are similar and range from 1.3%

Table 6.1: Prevalence of code review process smells in all pull requests (*All*) and pull requests with KC. *Count* column shows the number of pull requests identified with the smell. *Ratio* column presents the percentage of pull requests with the smell to the total number of pull requests, excluding NA ones. *NA*[1] column shows the number of pull requests excluded from the smell detection. Project names are abbreviated for spacing purposes.

Prj.	PR Data[2]	Lack of Review			Large Changesets			Missing Context			Ping-Pong			Sleeping Reviews		
		Count	Rat.	NA	Count	Rat.	NA	Count	Rat.	NA	Count	Rat.	NA	Count	Rat.	NA
ANG.	All	439	55.6%	7,210	481	6.0%	0	147	18.6%	7,210	22	0.3%	0	4,222	53.3%	77
	KC	374	53.2%	0	77	11.0%	0	119	16.9%	0	3	0.4%	0	297	42.2%	0
CES.	All	3,104	63.7%	654	1,003	18.1%	0	303	6.2%	654	246	4.4%	0	2,005	36.4%	23
	KC	2,830	65.6%	0	691	16.0%	0	274	6.4%	0	189	4.4%	0	1,414	32.8%	0
ESL.	All	1,689	32.6%	1,134	371	5.9%	0	795	15.3%	1,134	110	1.7%	0	2,811	44.6%	15
	KC	1,683	32.6%	0	299	5.8%	0	790	15.3%	0	96	1.9%	0	2,213	42.8%	0
GHO.	All	5,125	77.4%	1,961	629	7.3%	0	136	2.1%	1,961	31	0.4%	0	2,943	34.4%	25
	KC	4,781	77.6%	0	399	6.5%	0	116	1.9%	0	24	0.4%	0	1,842	29.9%	0
OPE.	All	3,064	42.5%	1,039	484	5.9%	0	547	7.6%	1,039	34	0.4%	0	1,947	23.7%	23
	KC	2,973	42.0%	0	366	5.2%	0	541	7.6%	0	23	0.3%	0	1,335	18.8%	0
PDF.	All	2,565	44.6%	1,005	424	6.3%	0	853	14.8%	1,005	8	0.1%	0	2,271	33.7%	16
	KC	2,549	44.5%	0	311	5.4%	0	842	14.7%	0	4	0.1%	0	1,774	31.0%	0
REA.	All	2,993	35.8%	3,999	1,122	9.1%	0	674	8.1%	3,999	181	1.5%	0	4,285	35.5%	283
	KC	2,920	35.7%	0	706	8.6%	0	667	8.2%	0	140	1.7%	0	2,193	26.8%	0
SHI.	All	354	7.2%	1,006	156	2.6%	0	305	6.2%	1,006	128	2.2%	0	1,489	25.4%	31
	KC	341	7.0%	0	127	2.6%	0	304	6.3%	0	124	2.6%	0	1,090	22.4%	0
STR.	All	598	12.5%	1,346	841	13.8%	0	346	7.3%	1,346	125	2.0%	0	2,522	41.7%	66
	KC	343	11.7%	0	197	6.7%	0	202	6.9%	0	68	2.3%	0	1,277	43.4%	0
WEB.	All	2,253	54.0%	1,615	413	7.1%	0	49	1.2%	1,615	36	0.6%	0	2,239	39.6%	134
	KC	1,981	55.3%	0	210	5.9%	0	48	1.3%	0	26	0.7%	0	1,156	32.3%	0
Tot.	All	22,184	42.2%	20,969	5,924	8.1%	0	4,155	7.9%	20,969	921	1.3%	0	26,734	36.7%	693
	KC	20,775	42.7%	0	3,383	6.9%	0	3,903	8.0%	0	697	1.4%	0	14,591	30.0%	0

[1] Due to algorithm precondition checks [27], some PRs are not evaluated for this particular smell. Those PRs are labeled with NA.
[2] Readers may refer to Tables 5.1 and 5.2 for the total number of PRs and the number of PRs with KCI, respectively.

(WEBPACK) to 16.9% (ANGULAR.JS). The *sleeping reviews* smell is generally more prevalent over all the projects, compared to other smells. *sleeping reviews* are lowest (almost 19%) in the OPENLAYERS project, and range up to 43.4% in STRAPI. The number and ratio of code reviews with the *ping-pong* smell is comparatively low for all projects, never rising above 5%.

Finding 0. Code review process smells are observable in all projects. The prevalence of process smells varies according to the smell type and project. The *lack of review* smell varies the most across projects. The *sleeping reviews* smell is in general the most prevalent, while the *ping-pong* smell is the least prevalent.

6.2 Relationship between Code Review Process Smells and Code Smells (RQ1)

We analyzed the relationship between the existence of CRPSs with code smells (ΔM_p^{total} and ΔK_p^{total}). Appendix B shows the detailed correlation results. Contrary to our intuition, our findings suggest that CRPSs have at best a weak correlation with ΔM_p^{total} and ΔK_p^{total}, and in many cases no correlation at all. In some projects, our correlation methods, i.e., CC and SC, yield different results. For instance, CC indicates no correlation between *lack of review* and ΔM_p^{total}, whereas SC indicates a weak correlation in REACT project.

Finding 1. The existence of CRPSs has at best a weak correlation with ΔM_p^{total} and ΔK_p^{total}. Thus we can conclude that the code review process smells in a pull request have little to no influence on code smells in the code in that pull request.

6.3 Aggregate Effect of Code Review Process Smells on Code Smells (RQ2)

We assessed whether the CRPS density ($\rho^{crps}(f)$) of a file is correlated with its code smell density ($\mu^c(f)$). Our expectation was that the higher the CRPS density of a file, the more smelly the file will be. However, the results show that, in general, CRPS density does not correlate with code smell density, as most of the correlations we found were statistically insignificant and/or of negligible strength. Appendix C shows the detailed values of the correlation coefficients.

The correlation between *lack of review* density and code smell density is statistically significant for ANGULAR.JS, CESIUM, ESLINT, and OPENLAYERS projects. The former two projects' SC and CC coefficient values indicate a positive and

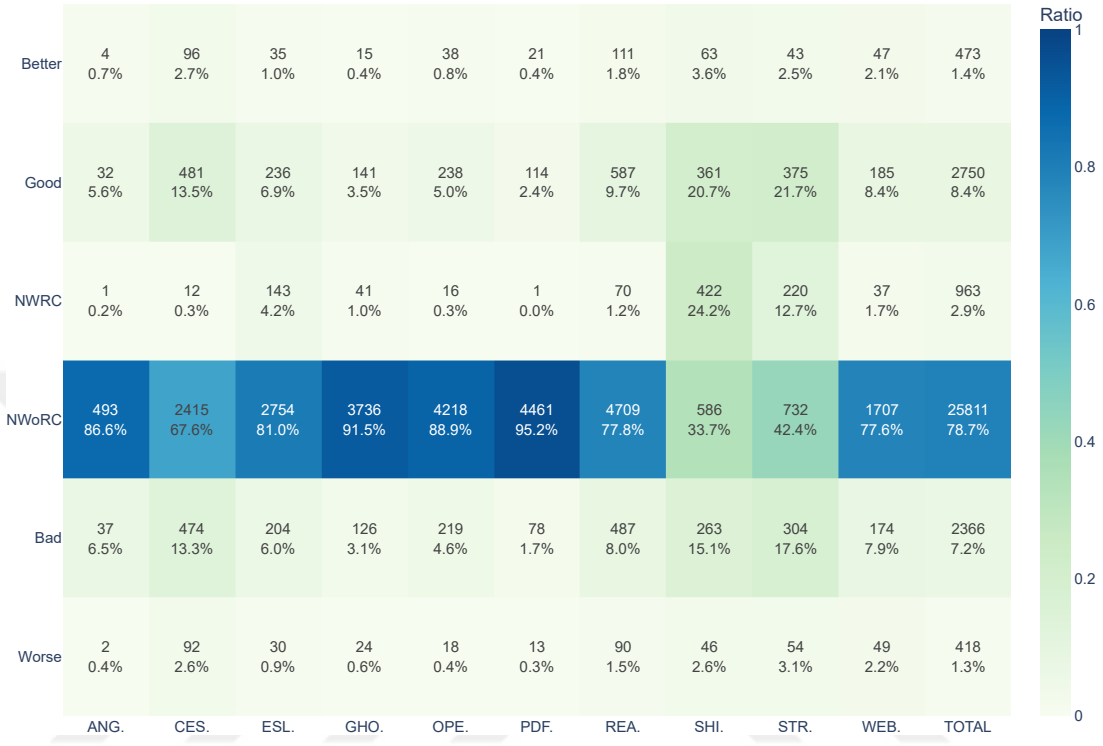


Figure 6.1: Overall code review classification results.

weak association. On the other hand, SC values from the last two projects demonstrate a weak association, in contrast to CC values which indicate a negligible correlation.

Some results are in contrast to our expectations. For the *large changesets* smell in ANGULAR.JS, CESIUM, OPENLAYERS projects, SC results show a weak negative correlation. The higher the CRPS density for *large changesets* smell, the less code smell density the file contains, although the association is weak.

Finding 2. The CRPS density and code smell density are, at best, weakly correlated. Thus, the code smell density of a file is not influenced by smelly code reviews.

6.4 Code Review Classifications (RQ3)

We obtained results that were counter to our expectations for RQ1 and RQ2. Our analyses viewed a code review in terms of its contribution to the code smell density, and then tested whether those code reviews that had unfavorable code smell density results were smelly or not. We decided to analyze the data from a different point of view in an effort to gain some insight into why our results were so counterintuitive. As introduced in Section 4.4, we classified the code review performance from the code smell perspective using the ΔM_p^{dev}, ΔM_p^{rev}, and ΔM_p^{total} parameters of ΔCoS-D analysis. This classification ignored whether the code reviews were smelly or not. Figure 6.1 shows the classification results.

It is evident that NWoRC reviews constitute the majority, varying between 67.6% and 95.2%, in all projects except SHIELDS and STRAPI. In other words, additional changes are usually not requested in code reviews. Looking at the other review quality classes, when there exists a review commit, the proportion of Bad and Worse code reviews (8.5%, in total) are similar to the proportions of Good and Better code reviews (9.8%, in total). This result indicates that code reviews, in general, do not often affect code smell densities either positively or negatively.

We further investigated the distribution of the smelly code reviews ratio (similar to Table 6.1) among the review classes. However, we did not obtain a notable insight.

Finding 3. Code reviews usually (81.8% of the time) remain neutral on code smells. When the code smell density changes as a result of the review, they increase the code smell density as often as they decrease it.

6.5 Focus Group with Practitioners (RQ4)

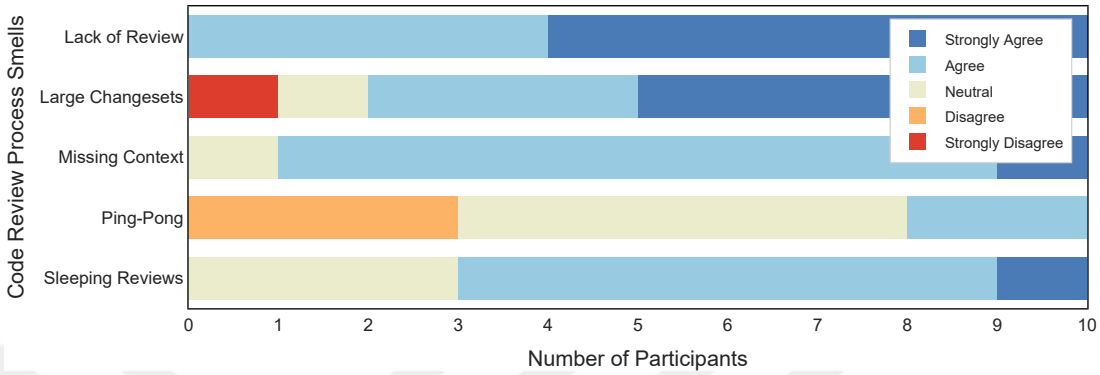
In this subsection, we present the focus group participants' ideas and emerging themes in the focus group sessions.

6.5.1 On FQ1 and FQ2

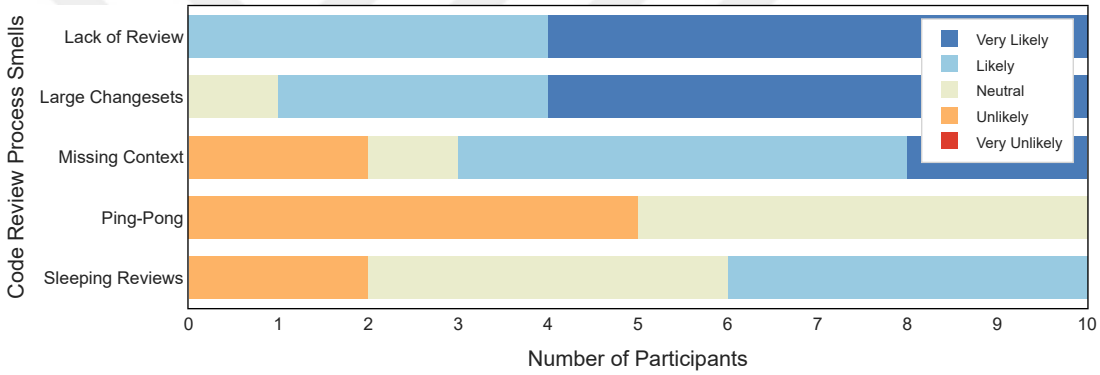
Figure 6.2 summarizes the focus group participants' responses to FQ1 and FQ2 from the preliminary survey (i.e., before they saw our quantitative results presented above). The results show that they generally agree that *lack of review*, *large changesets*, and *missing context* smells may hinder the benefits of the code review process, but are more neutral or disagreed for *ping-pong* and *sleeping reviews* smell. On the other hand, for FQ2, the responses were a bit more mixed. Almost all participants think that *lack of review* and *large changesets* smells are likely to lead to an increase in code smells. All but three of the participants agreed for the *missing context* smell as well. Participants are primarily neutral or find it unlikely that the *ping-pong* smell could lead to increased code smells, while views were quite mixed on the *sleeping reviews* smell.

Below, for each CRPS, we provide representative quotes from participants to exemplify emerging themes and interesting views we captured in our focus group sessions.

Lack of Review. Participants agree on the side effects of the *lack of review* smell and its relationship with code smells. As P10 said, "It is unacceptable to approve a code that was not inspected by someone else.". They also discussed situations in which the lack of review smell may lose its importance. For example, P4 shared that their team does not conduct a proper code review process for legacy projects because those products are mature and do not require extensive modifications. Another example is explained by P3 who said, "When we edit configuration files, e.g., YAML files, a review is not required as a few developers are knowledgeable of them anyways.". Lastly, P2 argued that code review does



(a) FQ1: Do you agree that this process smell could hinder benefits of the code review process?



(b) FQ2: In your opinion, how likely is it that code reviews with this process smell would lead to an increase in code smells?

Figure 6.2: Focus group participants' view on FQ1 and FQ2.

not make sense in a pair programming paradigm because the code is reviewed during development.

Large Changesets. Most attendees again agree with the drawbacks of *large changesets* in the code review process, as P3 indicated, "If a pull request is too large with many changes, some details are definitely missed.". We identified two main opposing views. The first one is to differentiate between *large changesets* with rather *straightforward* modifications and more complex changes. P1 explains, "To edit a common data model, you need to change many files. At the end, it seems like 100 files are modified, but in reality, it is not complicated.". P1 adds, "Sometimes, developers change the business logic. In that case, it is a real *large changesets*.". The second opposing view described a successful application of code reviews with *large changesets*: "To comply with the DO-200 standard,

we determined a code review inspection plan. In this plan, we grouped the code repository according to specific domains, which usually yielded *large changesets*. However, we successfully completed this process and complied with the standard." (P8).

Missing Context. There is a consensus among the participants that this smell does hinder a project from benefitting from the code review. P5 denoted, "If you want to assess the architecture and design in code reviews, this is impossible without context. For instance, how can you know which is better, UDP or TCP, with a missing context?". Several participants underlined the significance of context awareness in code reviews. "I can write a class that complies with all coding styles and rules and passes SonarQube checks, but if the class does not fit in the context, it may smash the software.", said P6.

The focus group participants had contrasting opinions regarding the *missing context's* potential effect on code smells. P10 shared that "There is no apparent connection between code changes and code smells. We can observe the same code smell in another context as well.". On the other hand, P8 indicated, "Code reviews without a context are pointless. Maybe you can catch a few things that tools could not...".

Ping-Pong. The majority of the participants shared that they face the *ping-pong* smell occasionally. P2 said, "It [ping-pong smell] happens, for instance, when a developer's implementation is in a wrong direction...". Similarly, P8 expressed that this smell is inevitable in remote working cultures. However, their *ping-pong* experiences did not yield clear ideas about its effect on code review benefits and code smells. They mostly remained neutral on these points. P2 indicated, "It is not necessarily a bad thing, but it should not repeat as well. Necessary actions should be taken.". P7 denoted, "The *ping-pong* existence could be an indicator of serious review culture in a company.".

Sleeping Reviews. Some participants declared that they face this smell in their code review process due to their work environment practices. However, we observed disagreement between the participants on the effects of the smell on code

reviews. On the one hand, they indicated that they could live with the smell. P10 said, "I am working in a distributed team, so we have an async working culture. If I open a PR today, I will see the review tomorrow. Besides, for some reason, the review may have gotten delayed, and so is the process. However, that is our work environment.". On the other hand, some participants were suspicious about the potential effects of the smell as P2 explains, "Perhaps, the smell does not take away the benefits. The review process is still in action. However, the branch gets outdated. So, in effect, it causes refactoring, which leads to code smells.". The survey results were quite mixed on this CRPS, and the discussion revealed that the presence and effects of *sleeping reviews* highly depend on the work context.

Finding 4. The focus group participants mostly agree that CRPSs could hinder the benefits of the code review process, except the *ping-pong* smell. The participants' views are somewhat mixed in FQ2. They generally find it likely that CRPSs other than *ping-pong* and *sleeping reviews* lead to an increase in code smells.

6.5.2 Reflection on the Quantitative Results

After discussing the CRPSs, as described above, we gathered the opinions of focus group participants on the quantitative results of our study. First, we presented the summary results of RQ1 and RQ2. Then, we asked exploratory questions to initiate conversations. Lastly, we showed the RQ3's results and gathered the participants' ideas. Based on these conversations, we identified common emerging themes that shed further light on the RQs. The themes try to identify the potential reasons behind the quantitative results of our study.

Developer profiles. Generally speaking, the participants underlined the potential behavior difference between open-source and commercial developers[1]. The participants indicated both positive and negative aspects related to this theme that could have had an influence on the results. One view states that

[1]We note that all survey participants are currently working in a company.

open-source contributions are the portfolios of open-source developers. Hence, they care about what they commit and try to avoid erroneous code. Another view argues that the open-source community usually comprises skillful developers, so their contributions are of high quality. The participants think that these *superior* cases and characteristics could be the reason behind the unintuitive results of RQ1, RQ2, and RQ3. On the other hand, there was one contradictory view on the same matter. P3 stated, "Maybe they [open-source developers] are satisfied with changes that resolve some bugs or introduce a feature that serves the community but includes code smells. Code smells may not be their concern, but the functionality.".

Finding 5. Most participants think that open-source developers try to make high-quality code contributions to their projects.

Different nature of open-source and company projects. This theme, although similar to the previous one, talks about the general practices followed in the two different project paradigms. Overall, we can group the participants' responses into two views. On the one hand, some participants think that open-source projects pay more attention to their software quality. P1 stated that "In our company, we try to have 70%-80% code coverage. However, I know many open-source projects with code coverage above 90%, even 100%.". Similarly, P6 expressed that "Especially if a project is used in a community or commercial environment, they [open-source developers] must be conducting serious code reviews." On the other hand, some participants think that company projects have higher quality standards, as P3 said "I think open-source communities are more concerned with increasing the prevalence of their projects. In our cases, we are more concerned with healthy software because we could lose customers if the code is smelly and buggy.". A similar view underlines that companies must meet specific standards and rules in a code review process, which may not be the case in open-source projects. Additionally, some participants mentioned that the results could be different, and more intuitive, when a wider variety of projects is analyzed, e.g. commercial projects and projects with different languages (e.g., Java, C#).

Finding 6. The participants shared conflicting views on the nature and adopted practices in company and open-source projects. Nevertheless, they identify the difference as a factor that could have influenced the results.

Tool utilization. The effectiveness and prevalence of linters and other tools were other emerging themes that were revealed in an effort to explore potential reasons behind the results. Participants emphasized the accuracy and robustness of linters. The main rationale behind this emphasis in the sessions was that as developers rely on linters during development, they may not seriously check or care about code smells in code reviews. P8 said "Linters reveal many problems.". Another participant mentioned the built-in capabilities of modern IDEs by saying "IDEs can detect most of what SonarQube finds." (P1). Besides, P4 argued that tool reliance might be encouraging negligence with respect to code smells in code reviews, as developers know that linters and SonarQube are used in their projects. Another aspect of the topic was tool configurations. P9 expressed that its company uses a specific set of SonarQube rules, which could be the case in our subject projects, i.e., the analyzed projects in our study could have adapted different or smaller set of code-smell checks than the ones we had in SonarQube.

Finding 7. The participants consider that linters and software quality tools are effective in removing code quality problems in code before code review.

Chapter 7

Discussion

Here, we first discuss and elaborate on the results presented in the previous section. Then, we introduce the potential implications of our research.

7.1 Prevalence and Effect of Code Review Process Smells

Our analyses showed that CRPSs occur in practice. The prevalence of each process smell type is different and differs among projects, as Table 6.1 shows. This is in line with the findings in [27], where the authors also used open-source projects. The result may be expected in this regard. However, we observed some differences, which we note here[1]. We found a lower prevalence of the *missing context* smell (highest rate of 18.2%, vs. 44.2% in [27] and the *ping-pong* (4.4% vs. 11.5% in [27]. We found the opposite for the *sleeping reviews* smell (23.7% vs. 10.8% in [27]). In both studies, however, CRPSs were found to occur in the code review process of all projects studied.

For RQ1 and RQ2, we investigated the potential relationship between the

[1]We compare the occurrence ratio of CRPSs in [27] with *All* rows in Table 6.1.

occurrence of CRPSs and code smell changes, both at the level of individual pull requests and at the aggregate level over the life of a file. We did not find a notable association between smelly code reviews and code smell increases (or decreases) in either case. This investigation sheds light on one aspect of CRPSs' potential effects, namely on code quality. However, we think the potential effect of the process smells on other outcomes of the code review process is worth exploring.

Table 7.1: The existence of recommendations and mentions on code quality and code review process smells in contribution guidelines.

Project	**Code Quality**	**Code Review Smells**				
		Lack of Review	**Large Changesets**	**Missing Context**	**Ping-Pong**	**Sleeping Reviews**
ANGULAR.JS [99]	Yes	Yes	No	Yes	No	No
CESIUM [100]	Yes	Yes	Yes	Yes	No	No
ESLINT [101]	Yes	Yes	No	Yes	No	No
GHOST [102]	Yes	Yes	No	Yes	No	No
OPENLAYERS [103]	No	No	No	Yes	No	No
PDF.JS [104]	Yes	Yes	No	No	No	No
REACT [105]	Yes	Yes	No	Yes	No	Yes
SHIELDS [106]	No	Yes	Yes	Yes	No	No
STRAPI [107]	Yes	Yes	No	Yes	No	No
WEBPACK [108]	Yes	Yes	No	Yes	No	No

7.2 Code Smell Density Changes in Code Reviews

In RQ3, we found that roughly in 8 out of 10 code reviews, code reviews remain neutral on code smells, regardless of how smelly the code reviews are. This finding implies that code changes are generally approved independently of whether the author increased or decreased the code smell density during pull request development. Therefore, we can conclude that the gatekeeping function of the code review process in terms of code quality is not achieved in the subject projects. We identify two potential reasons behind this result and observation, as presented below.

Developers rely on linters. To understand potential reasons behind the failure of code reviews to manage code smells, we decided to analyze the coding guidelines of the project repositories that we studied. Similar to [90], we investigated suggestions or practices aiming to achieve code quality by visiting the repository guidelines[2]. Some guidelines included external links to document their practices, e.g., REACT, and we also investigated the linked content. We visually inspected the guidelines for keywords (such as *code quality* or *guideline*) following [90]. The result of our investigation in Table 7.1, show that 8 out of 10 projects contain suggestions for achieving code quality. Those eight projects explicitly suggest using code linters, e.g., ESLint[3], which are static code analysis tools that identify potential issues such as bugs, bad practices, coding convention violations, and style problems. Developers utilize linters for several purposes, such as maintaining consistency, preventing errors, saving discussion time, and avoiding complex code [109]. We found evidence of this in our focus group sessions, where participants indicated the effectiveness of linters in their development environments. Thus, we contemplate that the reliance on linters could lead reviewers to the conclusion that they do not need to look for or report code smells in code reviews. Knowing that linters are suggested in the project guidelines, they could spend

[2]Repositories in GitHub usually contain `CONTRIBUTING.md` files for documenting the adapted practices.

[3]`https://eslint.org/` (Accessed on 19 Dec 2022)

their time and attention on higher-level issues in code reviews, such as solution rationale, architecture, and overall design.

Developers do not focus on code smells. Researchers have explored the perception and knowledge of code smells among developers. For example, Palomba et al. [110] found that developers perceive some code smell types as not problematic at all and that their perceptions vary according to the intensity of the smell type. Mello et al. [111] found that code smell detection is associated with developers' individual skills. Another study by Palomba et al. [112] found that developers find structural code smells more difficult to understand than textual smells. The existing literature indicates that manual code smell detection is a subjective task [111], and developers may not be aware of some code smells. Thus, we posit that developers being unaware of code smells or dismissing them could be the reason behind not identifying code smells during reviews.

We also note that our study found that roughly 8 out of 10 code reviews remain neutral (did not result in an increase or a decrease) on code smells. This is consistent with the finding of Han et al. [35], who reported that coding violations change only in 25.3% of code reviews. Furthermore, even if code smells change, it is equally likely (approximately 9% for each) that the change will negatively or positively affect code smells. The results hint at two possibilities. Either code smells are not a concept that developers pay sufficient attention to in code reviews, or the manual methods are less efficient or effective than current tools at spotting and fixing code smells. In fact, these two possibilities may be related, as developers may perceive that tools do a better job of managing code smells, and this leads them to minimize their attention on code smells in code reviews.

7.2.1 Implications For Researchers

Our results cue some new research potentials.

Call for investigating the impact of CRPSs on other aspects. We did not find a significant relationship between the code review process and code smells. Code

smells have been of interest of our community for a long time and have been well studied in both depth and breadth. However, the literature on code review process smells is recent. The potential effect and relationship of CRPSs with other benefits of the code review process could be a new and fruitful direction for future work.

Call for expanding the scope of investigation. Our quantitative results are based on data gathered from open-source projects written primarily in JavaScript. We acknowledge that the findings may vary in commercial projects and for other programming languages. This was also mentioned in our focus group sessions. Thus, we anticipate potential benefits for exploring the relationship between CRPSs and code smells in *industrial contexts*. Further, sufficient support for a programming language is required to utilize a software quality tool effectively. Tools do not (and cannot) support every programming language to the same extent. Current tools tend to support Java more extensively than other languages [45] [84] but this is likely to change over time. In our focus group sessions, some participants shared that our results are worth investigating in projects with main languages other than JavaScript. Thus, we believe that *analyzing projects written in other languages* is worth giving attention.

Call for exploring open-source and commercial software development paradigms. In our focus group sessions, practitioners had differing views on the developer profiles and development practices in open-source and commercial projects. The literature on this topic is quite limited to the best of our knowledge. A few studies [113] [114] [115] have done direct comparisons of development and management practices in commercial and open-source projects, but these studies were conducted in the mid-2000s. It is worth exploring the differences between open-source and commercial software development again, considering modern development practices.

7.2.2 Implications For Practitioners

We discuss the key take-aways of our study for practitioners below.

Improving and adhering to development guidelines. Table 7.1 shows that 8 out of 10 projects contain recommendations on code quality. However, our analyses showed that the code quality is not preserved considering the code smells. Besides, the guidelines mostly do not mention preventive measures for code review process smells. We recommend that development teams should *better adhere* to their established guidelines. The practitioners should also *evaluate and improve* the existing guidelines to improve their processes and avoid any kinds of smells.

Benefiting from linters while continuing code reviews. Tómasdóttir et al. [109] listed several advantages and use cases of linters for developers in JavaScript projects. Our focus group participants also mentioned the usefulness of linters in their development environments. The increased use of linters is encouraged further by including crucial checks [109]. We consider that linter usage frees up time to *focus more on the other goals* of the code review process. However, we want to highlight that use of linters in the development phase cannot replace the code review process, even from the code smells perspective, as our results show that code smells are still prevalent.

Call for a new set of guidelines in code reviews. Previous research has identified the motivations for conducting code reviews in detail, as explained in Section 2. Our results have shown that one of the key motivations (ensuring code quality) is not being fulfilled and is potentially no longer a valid concern for code reviewers in practice, because of the recent progress in the development tools used in different phases of the software development life cycle. We conjecture that incorporating modern tools could have changed both the understanding and practice of code reviews for practitioners. We believe that further investigation is required to understand this better. But if it is true that code review time and effort could be diverted from ensuring code quality without compromising that quality in the resulting code, then this calls for the development of a *a new set of guidelines for code reviews* to ensure that the process is focused on those goals and outcomes that code reviews are best suited for.

Chapter 8

Threats to Validity

We discuss the threats to validity of our study and results by following the guidelines of Wohlin et al. [116].

8.1 Internal Validity

The way that developers use GitHub could have influenced some of our results. Detecting the *lack of code review* smell and identifying key commits of pull requests depends on the *review* data of pull requests on GitHub. *Review* data is generated as long as developers use the review functionality of GitHub. But in some cases, developers might express their suggestions and findings on the changes introduced in a pull request in other ways, such as in *comments*. In this case, our analysis would have missed this data, and we would have incorrectly detected the *lack of code review* smell as well as key commits (c_p^{dev} and c_p^{rev}) of pull requests. We did not account for *comments* as part of *reviews* because we could not identify a methodological and reliable way to extract review-related data from comments. Further, manual extraction was infeasible due to the volume of comment data.

We evaluated the code smells at the key commits of pull requests. There is a possibility that, in prolonged development, developers may merge other branches

(usually the main branch) onto the branch they are working on to synchronize with the latest changes in the repository [117]. If this merge operation modifies the pull request's edited files, there exists a threat that the values of ΔK_p^{dev}, ΔK_p^{rev}, ΔK_p^{total}, ΔM_p^{dev}, ΔM_p^{rev}, and ΔM_p^{total} could be incorrect. Unfortunately, we could not identify a methodological way of detecting the described scenario in pull requests. Further, a manual check was not feasible (and not reliable as well) due to the number of data points we dealt with.

There are two assumptions we have made about the code smell evaluation and the comparison between c_p^{dev} and c_p^{rev}. One is that the pull request author implements the modifications requested by reviewers after c_p^{dev}. However, there is a possibility that commits after c_p^{dev} are not directly tied to reviewers' comments. Similarly, reviewers' comments could concern topics other than improving code smells, such as suggesting added functionality, improving the existing functionality, or improving testing [118]. Even though these two concerns are valid, code review is still expected to improve future maintenance of the repository. Thus we believe our strategy of catching code smell changes in pull requests is valid.

8.2 Construct Validity

Our code smell detection relied on the use of SonarQube's rules and heuristics. So, there is a possibility that the detection could produce wrong results in some cases. However, we can quantitatively express the code smell detection quality of SonarQube and, by design, it returns zero false-positive instances [92]. Further, SonarQube is supported by both a company[1] and an open-source community and has operated since 2007. So, we can conclude that the tool is quite mature. Finally, it is the most popular tool in the software community [120] to identify and remediate technical debt.

We used the algorithms in [27] to detect the code review process smells. The detection algorithms rely on configurable thresholds. We used the thresholds as

[1]SonarQube is a product of SonarSource S.A. [119].

they appear in the original paper [27] because they are based on a literature search and a survey of developers.

8.3 Conclusion Validity

Our quantitative data analysis depends on the theoretical fundamentals of our study, detailed in Chapter 4. We have expressed many of our constructs in formal mathematical notation in order to both check and demonstrate their validity. We note that there was no manual interception in any stage of quantitative data collection, which helps us avoid researcher bias. To further mitigate this threat, we utilized statistical tests (e.g., Spearman and Chatterjee correlations). We refrained from drawing further conclusions on the results that did not yield statistically significant values. We have also documented our data collection procedures carefully, for both the quantitative and qualitative portions of our work, and used widely accepted techniques for qualitative data analysis. We provided the resulting data artifacts in our online replication package [77] for replication and validation purposes. Results are always potentially prone to subjective evaluation. We not only have tried to avoid this possibility, but we have also been transparent about our methods so that any bias could be detected by others.

8.4 External Validity

There are three aspects to consider for the generalizability of our results. First, we used only open-source projects to explore the potential effects of code review process smells on code smells. However, our focus group participants were all commercial developers. Case studies on closed-source projects are required to improve generalizability. Secondly, we sampled only ten projects to get a broader view of the analysis results and diminish related concerns. Thirdly, all our projects used JavaScript as the primary language, which is a limiting factor for generalizing the results to projects using other programming languages. This choice was driven

by the limitations of code smell detection capabilities. SonarQube can detect code smells in various programming languages, but for some languages (e.g., Java) it requires compilation of the entire repository which requires too much computation power and time. Also, detection in some languages (e.g., Python) depends on specific versions of the language, and there is no standard way of finding the language version of projects. JavaScript does not have these limitations, which made it suitable for our analysis.

We used the code smells that can be detected by SonarQube. Based on the fact that SonarQube is backed by a company and has a large community, and it is widely used in academia and industry, it can be considered a reputable tool. Further, the tool covers a variety of code smells, which helps mitigate the threat of code smell types. However, replication of this study with other code smell types, such as the ones introduced by Fowler [121], could still be helpful.

Chapter 9

Conclusion

Our study showed that code review process smells have little to no correlation with code smells. That is to say, the process smells in the code review process do not result in higher (or lower) levels of code smells, which is contrary to our intuition. These results led us to investigate the code smell density changes after code reviews in general, with or without considering code review process smells. We found that code smell density was unaffected in 8 out of 10 code reviews. We also conducted two focus group sessions to understand practitioners' views on code review process smells, code smells, and their relationship. The focus group participants provided some explanations for the lack of an expected relationship between code review process smells and code smells, but also reinforced the common wisdom that well-executed code reviews should have a positive influence on code quality.

Improving code quality has historically been cited as one of the goals of code reviews [11, 9, 4, 29]. However, our results indicate that this particular benefit of code reviews no longer holds. While code reviews might still provide advantages such as removing defects and educating developers, they appear to be benign when it comes to code quality improvement, at least in terms of code smells. Even bad (i.e. smelly) code reviews appear to have little effect. Our focus group results provide some reasons for this, including the effectiveness (and increased use) of

modern static analysis tools that find and help remove code quality problems automatically before code is even submitted for review.

However, our focus group results also show that developers still believe that code reviews have benefits with respect to code quality. In many cases, our participants expressed the opinion that code review process smells have a negative impact on the level of code smells in the code being reviewed, but found their opinions refuted by our quantitative analysis results.

The combined message from our quantitative and qualitative results could be that we need to rethink the goals and expected benefits of code reviews and possibly let go of the idea that they are necessarily an efficient way to address code quality. This provides an opportunity for the community to reimagine the code review process, aligning it more closely with goals that the process can still achieve, such as improving product quality and educational goals such as spreading product knowledge around a development team, and onboarding new team members. Such a realignment could mean more efficient code review processes.

Bibliography

[1] E. A. Alomar, M. Chouchen, M. W. Mkaouer, and A. Ouni, "Code Review Practices for Refactoring Changes: An Empirical Study on OpenStack; Code Review Practices for Refactoring Changes: An Empirical Study on OpenStack," *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*, p. 13, 2022.

[2] P. C. Rigby and C. Bird, "Convergent contemporary software peer review practices," *2013 9th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE 2013 - Proceedings*, pp. 202–212, 2013.

[3] V. Stray, N. B. Moe, M. Mikalsen, and E. Hagen, "An Empirical Investigation of Pull Requests in Partially Distributed BizDevOps Teams," *Proceedings - 2021 IEEE/ACM Joint 15th International Conference on Software and System Processes and 16th ACM/IEEE International Conference on Global Software Engineering, ICSSP/ICGSE 2021*, pp. 110–119, 5 2021.

[4] T. Baum, O. Liskin, K. Niklas, and K. Schneider, "Factors influencing code review processes in industry," *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, vol. 13-18-November-2016, pp. 85–96, 11 2016.

[5] T. Baum, H. Leßmann, and K. Schneider, "The choice of code review process: A survey on the state of the practice," *Lecture Notes in Computer Science*, vol. 10611 LNCS, pp. 111–127, 2017.

[6] P. C. Rigby, D. M. German, L. Cowen, and M. A. Storey, "Peer Review on Open-Source Software Projects," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 23, 9 2014.

[7] A. Krutauz, T. Dey, P. C. Rigby, and A. Mockus, "Do code review measures explain the incidence of post-release defects?: Case study replications and bayesian networks," *Empirical Software Engineering*, vol. 25, pp. 3323–3356, 9 2020.

[8] P. Thongtanunam, S. McIntosh, A. E. Hassan, and H. Iida, "Investigating code review practices in defective files: An empirical study of the Qt system," *IEEE International Working Conference on Mining Software Repositories*, vol. 2015-August, pp. 168–179, 8 2015.

[9] L. MacLeod, M. Greiler, M. A. Storey, C. Bird, and J. Czerwonka, "Code Reviewing in the Trenches: Challenges and Best Practices," *IEEE Software*, vol. 35, pp. 34–42, 7 2018.

[10] G. Bavota and B. Russo, "Four eyes are better than two: On the impact of code reviews on software quality," *2015 IEEE 31st International Conference on Software Maintenance and Evolution, ICSME 2015 - Proceedings*, pp. 81–90, 11 2015.

[11] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," *Proceedings - International Conference on Software Engineering*, pp. 712–721, 2013.

[12] P. Rigby, B. Cleary, F. Painchaud, M. A. Storey, and D. German, "Contemporary peer review in action: Lessons from open source development," *IEEE Software*, vol. 29, no. 6, pp. 56–61, 2012.

[13] A. Bosu, M. Greiler, and C. Bird, "Characteristics of useful code reviews: An empirical study at Microsoft," *IEEE International Working Conference on Mining Software Repositories*, vol. 2015-August, pp. 146–156, 8 2015.

[14] P. J. Denning, "Editorial: what is software quality?," *Communications of the ACM*, vol. 35, pp. 13–15, 1 1992.

[15] M. Jørgensen, "Software quality measurement," *Advances in Engineering Software*, vol. 30, pp. 907–912, 12 1999.

[16] M. Galster, D. Weyns, D. Tofan, B. Michalik, and P. Avgeriou, "Variability in software systems-A systematic literature review," *IEEE Transactions on Software Engineering*, vol. 40, no. 3, pp. 282–306, 2014.

[17] M. Broy, F. Deissenboeck, and M. Pizka, "Demystifying Maintainability," *Proceedings of the 2006 International Workshop on Software Quality - WoSQ '06*, pp. 21–26, 2006.

[18] R. Baggen, J. P. Correia, K. Schill, and J. Visser, "Standardized code quality benchmarking for improving software maintainability," *Software Quality Journal*, vol. 20, pp. 287–307, 5 2012.

[19] A. Yamashita and S. Counsell, "Code smells as system-level indicators of maintainability: An empirical study," *Journal of Systems and Software*, vol. 86, pp. 2639–2653, 10 2013.

[20] F. Khomh, M. D. Penta, Y. G. Guéhéneuc, and G. Antoniol, "An exploratory study of the impact of antipatterns on class change- and fault-proneness," *Empirical Software Engineering*, vol. 17, pp. 243–275, 6 2012.

[21] P. Kruchten, R. L. Nord, and I. Ozkaya, "Technical debt: From metaphor to theory and practice," *IEEE Software*, vol. 29, no. 6, pp. 18–21, 2012.

[22] F. Palomba, G. Bavota, M. D. Penta, F. Fasano, R. Oliveto, and A. D. Lucia, "On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation," *Empirical Software Engineering*, vol. 23, pp. 1188–1221, 6 2018.

[23] Z. Soh, A. Yamashita, F. Khomh, and Y. G. Guéhéneuc, "Do code smells impact the effort of different maintenance programming activities?," *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016*, vol. 1, pp. 393–402, 5 2016.

[24] M. D'Ambros, A. Bacchelli, and M. Lanza, "On the impact of design flaws on software defects," *Proceedings - International Conference on Quality Software*, pp. 23–31, 2010.

[25] A. Yamashita and L. Moonen, "Exploring the impact of inter-smell relations on software maintainability: An empirical study," *Proceedings - International Conference on Software Engineering*, pp. 682–691, 2013.

[26] N. Zazworka, M. A. Shaw, F. Shull, and C. Seaman, "Investigating the impact of design debt on software quality," *Proceedings - International Conference on Software Engineering*, pp. 17–23, 2011.

[27] E. Doğan and E. Tüzün, "Towards a taxonomy of code review smells," *Information and Software Technology*, vol. 142, p. 106737, 2 2022.

[28] M. E. Fagan, "Design and code inspection to reduce errors in program development," *IBM Systems Journal*, vol. 15, no. 3, pp. 182–211, 1976.

[29] C. Sadowski, E. Söderberg, L. Church, M. Sipko, and A. Bacchelli, "Modern code review: A case study at google," *Proceedings - International Conference on Software Engineering*, pp. 181–190, 5 2018.

[30] M. V. Mäntylä and C. Lassenius, "What types of defects are really discovered in code reviews?," *IEEE Transactions on Software Engineering*, vol. 35, no. 3, pp. 430–448, 2009.

[31] M. Beller, A. Bacchelli, A. Zaidman, and E. Juergens, "Modern code reviews in open-source projects: Which problems do they fix?," *11th Working Conference on Mining Software Repositories, MSR 2014 - Proceedings*, pp. 202–211, 5 2014.

[32] X. Han, A. Tahir, P. Liang, S. Counsell, K. Blincoe, B. Li, and Y. Luo, "Code smells detection via modern code review: a study of the OpenStack and Qt communities," *Empirical Software Engineering*, vol. 27, pp. 1–42, 7 2022.

[33] S. Panichella, V. Arnaoudova, M. Di Penta, and G. Antoniol, "Would static analysis tools help developers with code reviews?," *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015 - Proceedings*, pp. 161–170, 4 2015.

[34] "Patch sets in gerrit." `https://gerrit-documentation.storage.googleapis.com/Documentation/3.6.1/concept-patch-sets.html`. Accessed on 2022-09-05.

[35] D. G. Han, C. Ragkhitwetsagul, J. Krinke, M. Paixao, and G. Rosa, "Does code review really remove coding convention violations?," *Proceedings - 20th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2020*, pp. 43–53, 9 2020.

[36] M. Paixao, J. Krinke, D. Han, and M. Harman, "CROP: Linking Code Reviews to Source Code Changes; CROP: Linking Code Reviews to Source Code Changes," *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, 2018.

[37] V. Lenarduzzi, V. Nikkola, N. Saarimäki, and D. Taibi, "Does code quality affect pull request acceptance? An empirical study," *Journal of Systems and Software*, vol. 171, p. 110806, 1 2021.

[38] V. Lenarduzzi, N. Saarimäki, and D. Taibi, "Some SonarQube issues have a significant but small effect on faults and changes. A large-scale empirical study," *Journal of Systems and Software*, vol. 170, p. 110750, 12 2020.

[39] L. Pascarella, D. Spadini, F. Palomba, and A. Bacchelli, "On The Effect Of Code Review On Code Smells," 12 2019.

[40] N. Davila and I. Nunes, "A systematic literature review and taxonomy of modern code review," *Journal of Systems and Software*, vol. 177, p. 110951, 7 2021.

[41] R. Morales, S. Mcintosh, and F. Khomh, "Do code review practices impact design quality? A case study of the Qt, VTK, and ITK projects," *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015 - Proceedings*, pp. 171–180, 4 2015.

[42] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan, "An empirical study of the impact of modern code review practices on software quality," *Empirical Software Engineering*, vol. 21, pp. 2146–2189, 10 2016.

[43] J. Shimagaki, Y. Kamei, S. McIntosh, A. E. Hassan, and N. Ubayashi, "A study of the quality-impacting practices of modern code review at Sony mobile," *Proceedings - International Conference on Software Engineering*, pp. 212–221, 5 2016.

[44] A. Uchoa, C. Barbosa, W. Oizumi, P. Blenilio, R. Lima, A. Garcia, and C. Bezerra, "How Does Modern Code Review Impact Software Design Degradation? An In-depth Empirical Study," *Proceedings - 2020 IEEE International Conference on Software Maintenance and Evolution, ICSME 2020*, pp. 511–522, 9 2020.

[45] E. V. D. P. Sobrinho, A. De Lucia, and M. D. A. Maia, "A Systematic Literature Review on Bad Smells-5 W's: Which, When, What, Who, Where," *IEEE Transactions on Software Engineering*, vol. 47, pp. 17–66, 1 2021.

[46] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. D. Penta, A. De Lucia, and D. Poshyvanyk, "When and Why Your Code Starts to Smell Bad (and Whether the Smells Go Away)," *IEEE Transactions on Software Engineering*, vol. 43, pp. 1063–1088, 11 2017.

[47] G. Bavota, A. Qusef, R. Oliveto, A. De Lucia, and D. Binkley, "Are test smells really harmful? An empirical study," *Empirical Software Engineering*, vol. 20, pp. 1052–1094, 8 2015.

[48] L. Pascarella, D. Spadini, F. Palomba, M. Bruntink, and A. Bacchelli, "Information Needs in Contemporary Code Review," *Proceedings of the ACM on Human-Computer Interaction*, vol. 2, p. 27, 11 2018.

[49] A. Rahman, C. Parnin, and L. Williams, "The Seven Sins: Security Smells in Infrastructure as Code Scripts," *Proceedings - International Conference on Software Engineering*, vol. 2019-May, pp. 164–175, 5 2019.

[50] T. Sharma, M. Fragkoulis, and D. Spinellis, "Does Your Configuration Code Smell?," *Proceedings of the 13th International Conference on Mining Software Repositories*, 2016.

[51] F. A. Fontana, I. Pigazzini, R. Roveda, and M. Zanoni, "Automatic detection of instability architectural smells," *Proceedings - 2016 IEEE International Conference on Software Maintenance and Evolution, ICSME 2016*, pp. 433–437, 1 2017.

[52] D. Sas, P. Avgeriou, and U. Uyumaz, "On the evolution and impact of architectural smells—an industrial case study," *Empirical Software Engineering*, vol. 27, pp. 1–45, 7 2022.

[53] F. Zampetti, C. Vassallo, S. Panichella, G. Canfora, H. Gall, and M. Di Penta, "An empirical characterization of bad practices in continuous integration," *Empirical Software Engineering*, vol. 25, pp. 1095–1135, 3 2020.

[54] C. Vassallo, S. Proksch, A. Jancso, H. C. Gall, and M. D. Penta, "Configuration Smells in Continuous Delivery Pipelines: A Linter and a Six-Month Study on GitLab ACM Reference Format," *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020.

[55] K. A. Qamar, E. Sülün, and E. Tüzün, "Taxonomy of bug tracking process smells: Perceptions of practitioners and an empirical analysis," *Information and Software Technology*, vol. 150, p. 106972, 10 2022.

[56] D. A. Tamburri, P. Kruchten, P. Lago, and H. v. Vliet, "Social debt in software engineering: insights from industry," *Journal of Internet Services and Applications*, vol. 6, pp. 1–17, 12 2015.

[57] D. A. Tamburri, F. Palomba, and R. Kazman, "Exploring Community Smells in Open-Source: An Automated Approach," *IEEE Transactions on Software Engineering*, vol. 47, pp. 630–652, 3 2021.

[58] E. Caballero-Espinosa, J. C. Carver, and K. Stowers, "Community smells—The sources of social debt: A systematic literature review," *Information and Software Technology*, vol. 153, p. 107078, 1 2023.

[59] M. Chouchen, A. Ouni, R. G. Kula, D. Wang, P. Thongtanunam, M. W. Mkaouer, and K. Matsumoto, "Anti-patterns in Modern Code Review: Symptoms and Prevalence," *Proceedings - 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2021*, pp. 531–535, 3 2021.

[60] N. S. Alves, T. S. Mendes, M. G. De Mendonça, R. O. Spinola, F. Shull, and C. Seaman, "Identification and management of technical debt: A systematic mapping study," *Information and Software Technology*, vol. 70, pp. 100–121, 2 2016.

[61] D. A. Tamburri, P. Kruchten, P. Lago, and H. Van Vliet, "What is social debt in software engineering?," *2013 6th International Workshop on Cooperative and Human Aspects of Software Engineering, CHASE 2013 - Proceedings*, pp. 93–96, 2013.

[62] A. Martini, V. Stray, and N. B. Moe, "Technical-, Social- and Process Debt in Large-Scale Agile: An Exploratory Case-Study," *Lecture Notes in Business Information Processing*, vol. 364, pp. 112–119, 2019.

[63] N. S. Alves, L. F. Ribeiro, V. Caires, T. S. Mendes, and R. O. Spínola, "Towards an ontology of terms on technical debt," *Proceedings - 2014 6th IEEE International Workshop on Managing Technical Debt, MTD 2014*, pp. 1–7, 12 2014.

[64] W. Cunningham, "The WyCash portfolio management system," *ACM SIGPLAN OOPS Messenger*, vol. 4, pp. 29–30, 12 1992.

[65] P. Avgeriou, P. Kruchten, I. Ozkaya, and C. Seaman, "Managing Technical Debt in Software Engineering (Dagstuhl Seminar 16162)," *DROPS-IDN/6693*, vol. 6, no. 4, 2016.

[66] E. Lim, N. Taksande, and C. Seaman, "A balancing act: What software practitioners have to say about technical debt," *IEEE Software*, vol. 29, no. 6, pp. 22–27, 2012.

[67] Z. Li, P. Avgeriou, and P. Liang, "A systematic mapping study on technical debt and its management," *Journal of Systems and Software*, vol. 101, pp. 193–220, 3 2015.

[68] C. Izurieta, A. Vetrò, N. Zazworka, Y. Cai, C. Seaman, and F. Shull, "Organizing the technical debt landscape," *2012 3rd International Workshop on Managing Technical Debt, MTD 2012 - Proceedings*, pp. 23–26, 2012.

[69] Z. Codabux and B. Williams, "Managing technical debt: An industrial case study," *2013 4th International Workshop on Managing Technical Debt, MTD 2013 - Proceedings*, pp. 8–15, 2013.

[70] A. Martini, T. Besker, and J. Bosch, "Process debt: A first exploration," *Proceedings - Asia-Pacific Software Engineering Conference, APSEC*, vol. 2020-December, pp. 316–325, 12 2020.

[71] J. Cleland-Huang, O. C. Gotel, J. H. Hayes, P. Mäder, and A. Zisman, "Software traceability: Trends and future directions," *Future of Software Engineering, FOSE 2014 - Proceedings*, pp. 55–69, 5 2014.

[72] E. Sülün, E. Tüzün, and U. Doğrusöz, "RSTrace+: Reviewer suggestion using software artifact traceability graphs," *Information and Software Technology*, vol. 130, p. 106455, 2 2021.

[73] Y. Yu, H. Wang, G. Yin, and T. Wang, "Reviewer recommendation for pull-requests in GitHub: What can we learn from code review and bug assignment?," *Information and Software Technology*, vol. 74, pp. 204–218, 6 2016.

[74] V. Kovalenko, N. Tintarev, E. Pasynkov, C. Bird, and A. Bacchelli, "Does Reviewer Recommendation Help Developers?," *IEEE Transactions on Software Engineering*, vol. 46, pp. 710–731, 7 2020.

[75] E. Tom, A. Aurum, and R. Vidgen, "An exploration of technical debt," *Journal of Systems and Software*, vol. 86, pp. 1498–1516, 6 2013.

[76] N. Rios, M. G. d. Mendonça Neto, and R. O. Spínola, "A tertiary study on technical debt: Types, management strategies, research trends, and base information for practitioners," *Information and Software Technology*, vol. 102, pp. 117–145, 10 2018.

[77] E. Tuna, C. Seaman, and E. Tüzün, "Replication package of the thesis." `https://doi.org/10.17632/2zknb6r4kx.2`.

[78] P. Schober and L. A. Schwarte, "Correlation coefficients: Appropriate use and interpretation," *Anesthesia and Analgesia*, vol. 126, pp. 1763–1768, 5 2018.

[79] R. L. Fowler, "Power and Robustness in Product-Moment Correlation:," *http://dx.doi.org/10.1177/014662168701100407*, vol. 11, pp. 419–428, 7 1987.

[80] D. Falessi and R. Kazman, "Worst Smells and Their Worst Reasons," *Proceedings - 2021 IEEE/ACM International Conference on Technical Debt, TechDebt 2021*, pp. 45–54, 5 2021.

[81] S. Chatterjee, "A New Coefficient of Correlation," *Journal of the American Statistical Association*, vol. 116, no. 536, 2021.

[82] "R stats package." `https://www.rdocumentation.org/packages/stats/versions/3.6.2/topics/cor.test`. Accessed on 2022-07-29.

[83] "Xicor cran package." `https://cran.r-project.org/web/packages/XICOR/index.html`. Accessed on 2022-07-29.

[84] T. Amanatidis, N. Mittas, A. Moschou, A. Chatzigeorgiou, A. Ampatzoglou, and L. Angelis, "Evaluating the agreement among technical debt measurement tools: building an empirical benchmark of technical debt liabilities," *Empirical Software Engineering*, vol. 25, pp. 4161–4204, 9 2020.

[85] S. Dueñas, V. Cosentino, G. Robles, and J. M. Gonzalez-Barahona, "Perceval: Software project data at your will," *Proceedings - International Conference on Software Engineering*, pp. 1–4, 5 2018.

[86] "Git garbage collection documentation." `https://git-scm.com/docs/git-gc`. Accessed on 2022-07-10.

[87] G. A. Campbell and P. P. Papapetrou, *Sonar in Action*. Manning Publications Co, 1 ed., 11 2013.

[88] "Sonarsource customer list." `https://www.sonarsource.com/company/customers`. Accessed on 2022-12-19.

[89] J. M. Conejero, R. Rodríguez-Echeverría, J. Hernández, P. J. Clemente, C. Ortiz-Caraballo, E. Jurado, and F. Sánchez-Figueroa, "Early evaluation of technical debt impact on maintainability," *Journal of Systems and Software*, vol. 142, pp. 92–114, 8 2018.

[90] G. Digkas, A. Chatzigeorgiou, A. Ampatzoglou, and P. Avgeriou, "Can Clean New Code Reduce Technical Debt Density?," *IEEE Transactions on Software Engineering*, vol. 48, pp. 1705–1721, 5 2022.

[91] "Sonarqube metric definitions." `https://docs.sonarqube.org/9.5/user-guide/metric-definitions`. Accessed on 2022-07-10.

[92] "Sonarqube rules." `https://docs.sonarqube.org/9.5/user-guide/rules`. Accessed on 2022-07-10.

[93] "Sonarqube issue severity levels." `https://docs.sonarqube.org/9.5/user-guide/issues`. Accessed on 2022-07-10.

[94] C. B. Seaman, "Qualitative methods in empirical studies of software engineering," *IEEE Transactions on Software Engineering*, vol. 25, no. 4, pp. 557–572, 1999.

[95] L. A. Palinkas, S. M. Horwitz, C. A. Green, J. P. Wisdom, N. Duan, and K. Hoagwood, "Purposeful sampling for qualitative data collection and analysis in mixed method implementation research," *Administration and policy in mental health*, vol. 42, p. 533, 9 2015.

[96] R. A. Krueger and M. A. Casey, *Focus Groups: A Practical Guide for Applied Research*. SAGE Publication, Inc, 8 2014.

[97] M. Soliman, P. Avgeriou, and Y. Li, "Architectural design decisions that incur technical debt — An industrial case study," *Information and Software Technology*, vol. 139, p. 106669, 11 2021.

[98] "Microsoft word transcription." `https://support.microsoft.com/en-us/office`/transcribe-your-recordings-7fc2efec-245e-45f0-b053-2a97531ecf57. Accessed on 2022-10-09.

[99] "Angular.js contribution guideline." `https://github.com/angular/angular.js/blob/master/CONTRIBUTING.md`. Accessed on 2022-07-22.

[100] "Cesium contribution guideline." `https://github.com/CesiumGS/cesium/blob/main/CONTRIBUTING.md`. Accessed on 2022-07-22.

[101] "Eslint contribution guideline." `https://github.com/eslint/eslint/blob/main/CONTRIBUTING.md`. Accessed on 2022-07-22.

[102] "Ghost contribution guideline." `https://github.com/TryGhost/Ghost/blob/main/.github/CONTRIBUTING.md`. Accessed on 2022-07-22.

[103] "Openlayers contribution guideline." `https://github.com/openlayers/openlayers/blob/main/CONTRIBUTING.md`. Accessed on 2022-07-22.

[104] "Pdf.js contribution guideline." `https://github.com/mozilla/pdf.js/wiki/Contributing`. Accessed on 2022-07-22.

[105] "React contribution guideline." `https://reactjs.org/docs/how-to-contribute.html`. Accessed on 2022-07-22.

[106] "Shields contribution guideline." `https://github.com/badges/shields/blob/master/CONTRIBUTING.md`. Accessed on 2022-07-22.

[107] "Strapi contribution guideline." `https://github.com/strapi/strapi/blob/master/CONTRIBUTING.md`. Accessed on 2022-07-22.

[108] "Webpack contribution guideline." `https://github.com/webpack/webpack/blob/main/CONTRIBUTING.md`. Accessed on 2022-07-22.

[109] K. F. Tomasdottir, M. Aniche, and A. Van Deursen, "The Adoption of JavaScript Linters in Practice: A Case Study on ESLint," *IEEE Transactions on Software Engineering*, vol. 46, pp. 863–891, 8 2020.

[110] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, and A. De Lucia, "Do they really smell bad? A study on developers' perception of bad code smells," *Proceedings - 30th International Conference on Software Maintenance and Evolution, ICSME 2014*, pp. 101–110, 12 2014.

[111] R. De Mello, A. Uchoa, R. Oliveira, W. Oizumi, J. Souza, K. Mendes, D. Oliveira, B. Fonseca, and A. Garcia, "Do Research and Practice of Code Smell Identification Walk Together? A Social Representations Analysis," *International Symposium on Empirical Software Engineering and Measurement*, 9 2019.

[112] F. Palomba, A. Panichella, A. Zaidman, R. Oliveto, and A. De Lucia, "The Scent of a Smell: An Extensive Comparison between Textual and Structural Smells," *IEEE Transactions on Software Engineering*, vol. 44, pp. 977–1000, 10 2018.

[113] S. Lussier, "New Tricks: How Open Source Changed the Way My Team Works," *IEEE Software*, vol. 21, pp. 68–72, 1 2004.

[114] N. Serrano, S. Calzada, J. M. Sarriegui, and I. Ciordia, "From Proprietary to Open Source Tools in Information Systems Development," *IEEE Software*, vol. 21, pp. 56–58, 1 2004.

[115] A. I. Wasserman and E. Capra, "Evaluating software engineering processes in commercial and community open source projects," *First International Workshop on Emerging Trends in FLOSS Research and Development, FLOSS'07*, 2007.

[116] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, "Planning," *Experimentation in Software Engineering*, pp. 89–116, 2012.

[117] M. Paixao and P. H. Maia, "Rebasing in code review considered harmful: A large-scale empirical investigation," *Proceedings - 19th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2019*, pp. 45–55, 9 2019.

[118] J. Jiang, J. Lv, J. Zheng, and L. Zhang, "How Developers Modify Pull Requests in Code Review," *IEEE Transactions on Reliability*, 2021.

[119] "Sonarsource customer list." `https://www.sonarsource.com/`. Accessed on 2022-07-10.

[120] P. C. Avgeriou, D. Taibi, A. Ampatzoglou, F. Arcelli Fontana, T. Besker, A. Chatzigeorgiou, V. Lenarduzzi, A. Martini, A. Moschou, I. Pigazzini, N. Saarimaki, D. D. Sas, S. S. De Toledo, and A. A. Tsintzira, "An Overview and Comparison of Technical Debt Measurement Tools," *IEEE Software*, vol. 38, pp. 61–71, 5 2021.

[121] M. Fowler, *Refactoring: Improving the Design of Existing Code*. 1999.

Appendix A

Interpreting Delta Analysis Parameters

In Section 4, we explained the theoretical and conceptual development of the delta analyses. However, we did not go into details of the intuitive interpretations of the delta analysis parameters, i.e., ΔK_p^{dev}, ΔK_p^{rev}, ΔK_p^{total}, ΔM_p^{dev}, ΔM_p^{rev}, and ΔM_p^{total} parameters. Here, we provide explanations for those parameters and the mathematical values they can take. We utilize the sample scenario in Figure 4.1 for clarity.

As delta analyses express the percent change in code smells mathematically, they have a corresponding meaning in real life. In this subsection, we explain those meanings. A *change* could be an *increase*, *no change*, or a *decrease*. In numbers, *increase*, *no change*, or *decrease* corresponds to the value being > 0, $= 0$, or < 0, respectively. The interpretation of the *change* is different for the parameters of ΔCoS-C and ΔCoS-D analyses.

For ΔK_p^{dev}, ΔK_p^{rev}, and ΔK_p^{total}, *increase* indicates an increase in the code smell count. *Decrease* and *no change* are similarly explained. On the other hand, for ΔM_p^{dev}, ΔM_p^{rev}, and ΔM_p^{total} in ΔCoS-D analysis, *increase* corresponds to boosting the code smell count per NCLOC, i.e. the code smell density. One

can achieve this by decreasing the NCLOC, increasing the code smell count, or by some combination that yields the same result. *Decrease* can be explained similarly. In this case, *no change* indicates keeping the ratio of code smell count to NCLOC the same. To exemplify, one can keep this ratio by doubling the code smell count and NCLOC or changing them with the same multiplier. So, *no change* does not mean keeping the total code smell count the same for ΔCoS-D analysis parameters.

Changes for ΔK_p^{dev} and ΔM_p^{dev}. An *increase*, *decrease*, or *no change* for ΔK_p^{dev} and ΔM_p^{dev} realized by changes made by the PR author during the development of the pull request, comparing commits `c3` to `c0` in Figure 4.1.

Changes for ΔK_p^{rev} and ΔM_p^{rev}. An *increase*, *decrease*, or *no change* for ΔK_p^{rev} and ΔM_p^{rev} realized by changes made by the PR author after receiving review comments from the reviewers, comparing `c5` to `c3` in Figure 4.1.

Changes for ΔK_p^{total} and ΔM_p^{total}. An *increase*, *decrease*, or *no change* for ΔK_p^{rev} and ΔM_p^{rev} realized by changes made by the PR author as the result of merging new source code changes, including changes made during the development of the PR and changes made in response to review comments, comparing `c5` to `c0` in Figure 4.1.

Appendix B

Detailed Results of RQ1

We analyzed whether the existence of the code review process smells is related to undesired code smell results. We statistically explored this correlation using two techniques, the Chatterjee correlation (CC) and Spearman correlation (SC). We used two correlation methods to cover non-linear and linear potential relationships between two concepts. Table B.1 shows the details of the obtained correlation coefficients for both CC and SC methods concerning the RQ1.

Each project consists of four rows in the table. The first two rows of a project express the CC value of the ΔK_p^{rev} and ΔM_p^{rev} parameters with each CRPS. The last two rows document the same values for SC. Looking at CESIUM project, we can see that the relationship between ΔK_p^{rev} and lack of code review smell is not statistically significant, and the CC coefficient value is 0.00. In contrast, when SC is used, the same correlation is statistically significant with -0.08 coefficient value. The result indicates a *negligible* correlation..

Table B.1: Correlation of the CRPS presence with ΔK_p^{total} and ΔM_p^{total}. Statistically significant coefficient values ($p < 0.05$) are expressed in boldface.

Prj.	CM	Δ Parameter	Lack of Review	Large Changesets	Missing Context	Ping-Pong	Sleeping Reviews
ANG.	CC	ΔK_p^{total}	-0.03	0.02	-0.02	-0.01	-0.04
		ΔM_p^{total}	0.03	0.02	0.00	0.02	-0.01
	SC	ΔK_p^{total}	0.10	**0.16**	0.11	-0.03	-0.07
		ΔM_p^{total}	0.09	0.04	0.12	-0.03	-0.02
CES.	CC	ΔK_p^{total}	0.00	**0.03**	-0.03	-0.01	**0.03**
		ΔM_p^{total}	0.02	0.01	0.02	0.02	0.00
	SC	ΔK_p^{total}	**-0.08**	**0.2**	-0.03	**0.15**	**0.12**
		ΔM_p^{total}	-0.04	**0.12**	**-0.07**	-0.02	0.01
ESL.	CC	ΔK_p^{total}	0.00	-0.01	0.02	-0.01	-0.01
		ΔM_p^{total}	0.01	-0.01	-0.01	-0.01	-0.01
	SC	ΔK_p^{total}	**-0.19**	0.02	**-0.07**	-0.04	**0.16**
		ΔM_p^{total}	**-0.24**	0.04	**-0.16**	0.04	**0.21**
GHO.	CC	ΔK_p^{total}	-0.01	0.00	-0.03	-0.02	0.00
		ΔM_p^{total}	-0.02	-0.03	-0.03	-0.04	-0.03
	SC	ΔK_p^{total}	0.03	-0.05	0.05	0.01	**-0.09**
		ΔM_p^{total}	**-0.08**	**0.08**	0.01	0.00	0.01
OPE.	CC	ΔK_p^{total}	-0.02	0.01	0.00	0.02	0.00
		ΔM_p^{total}	0.01	-0.02	-0.01	-0.01	0.00
	SC	ΔK_p^{total}	-0.05	-0.02	-0.05	0.03	0.01
		ΔM_p^{total}	**-0.17**	0.04	**-0.07**	0.00	**0.07**
PDF.	CC	ΔK_p^{total}	0.01	**0.04**	0.00	0.02	0.02
		ΔM_p^{total}	**0.04**	-0.01	0.01	-0.01	-0.01
	SC	ΔK_p^{total}	**-0.16**	**0.14**	0.00	0.00	-0.03
		ΔM_p^{total}	**-0.3**	**0.13**	-0.02	**-0.06**	**-0.05**
REA.	CC	ΔK_p^{total}	**0.03**	0.01	0.01	0.00	0.01
		ΔM_p^{total}	0.01	0.00	0.02	0.01	-0.02
	SC	ΔK_p^{total}	-0.01	0.03	-0.01	-0.01	**-0.05**
		ΔM_p^{total}	**-0.14**	**0.11**	**-0.1**	-0.02	**-0.04**
SHI.	CC	ΔK_p^{total}	-0.05	-0.01	-0.03	-0.01	-0.04
		ΔM_p^{total}	**0.05**	**0.06**	0.04	0.03	0.02
	SC	ΔK_p^{total}	-0.06	**0.15**	**0.17**	-0.05	-0.11
		ΔM_p^{total}	**-0.23**	**0.19**	**0.09**	-0.05	**-0.21**
STR.	CC	ΔK_p^{total}	-0.01	0.00	0.01	0.00	-0.06
		ΔM_p^{total}	0.03	0.00	0.00	0.03	0.00
	SC	ΔK_p^{total}	-0.12	**-0.23**	-0.03	-0.01	-0.03
		ΔM_p^{total}	0.05	**0.16**	0.05	0.07	0.02
WEB.	CC	ΔK_p^{total}	**0.05**	0.03	0.03	0.03	0.04
		ΔM_p^{total}	-0.06	0.00	0.01	0.01	-0.05
	SC	ΔK_p^{total}	-0.01	**-0.17**	-0.07	-0.01	-0.01
		ΔM_p^{total}	0.04	0.04	**-0.08**	**-0.09**	0.00

Appendix C

Detailed Results of RQ2

In RQ2, we investigated the cumulative effect of smelly code reviews on the code smell density of files. We used two different correlation methods similar to RQ1. Table C.1 shows the details of the obtained correlation coefficients for both CC and SC methods concerning the RQ2.

In the table, each project consists of two rows where the rows indicate the CC and SC coefficient values, respectively. CESIUM project's coefficient values concerning lack of review smell are statistically significant and are 0.20 and 0.34 for CC and SC coefficients, respectively. Both values correspond to *weak* correlation. The conceptual interpretation is that the CRPS and CS densities of files have a weak relationship in CESIUM project.

Table C.1: Correlation of CRPS Density and CS Density per file in projects. Statistically significant coefficient values ($p < 0.05$) are expressed in boldface.

Project	CM	Lack of Review	Large Changesets	Missing Context	Ping-Pong	Sleeping Reviews
ANG.	CC	**0.17**	**0.21**	**0.06**	0.02	**0.20**
	SC	**0.10**	**-0.20**	**0.11**	**0.23**	**0.14**
CES.	CC	**0.20**	**0.20**	**0.05**	**0.18**	**0.20**
	SC	**0.34**	**-0.23**	0.02	**-0.29**	**-0.12**
ESL.	CC	**0.05**	**0.07**	**0.04**	-0.01	**0.15**
	SC	**-0.15**	0.02	**-0.16**	0.06	**0.24**
GHO.	CC	-0.02	-0.02	-0.02	-0.02	-0.03
	SC	**0.10**	**-0.06**	**0.14**	NA[1]	0.04
OPE.	CC	**0.06**	**0.08**	**0.07**	**0.07**	**0.05**
	SC	**0.11**	**-0.21**	**0.18**	**0.18**	**0.17**
PDF.	CC	-0.01	0.06	-0.05	0.03	**0.08**
	SC	**0.21**	**-0.15**	**0.12**	0.09	-0.02
REA.	CC	-0.03	0.02	-0.03	-0.02	0.03
	SC	0.00	**-0.06**	0.05	**0.10**	**-0.07**
SHI.	CC	-0.04	-0.03	0.00	0.00	-0.06
	SC	0.07	0.11	-0.07	**0.19**	-0.11
STR.	CC	**0.04**	0.02	0.00	0.01	0.02
	SC	**0.06**	**-0.07**	**0.05**	-0.03	**0.06**
WEB.	CC	0.01	**0.02**	0.01	0.01	0.02
	SC	-0.01	0.02	**0.08**	**0.10**	**0.06**

[1] *Ping-pong* smell result of SC analysis in GHOST project is marked with NA because none of the files in the latest commit of the project contains this smell. So, the Spearman correlation cannot be calculated as the standard deviation is zero in R-package.