The Journal of Systems and Software 215 (2024) 112101

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Do code reviews lead to fewer code smells?☆

Erdem Tuna [a,*], Carolyn Seaman [b], Eray Tüzün [a]

[a] Bilkent University, Ankara, 06800, Turkey
[b] University of Maryland, Baltimore County, 1000 Hilltop Circle, Baltimore, 21250, MD, United States

ARTICLE INFO

Keywords:
Code reviews
Code review smells
Process smells, Code smells
Focus group
Empirical study
Mining software repositories

ABSTRACT

Context: The code review process is conducted by software teams with various motivations. Among other goals, code reviews act as a gatekeeper for software quality.
Objective: In this study, we explore whether code reviews have an impact on one specific aspect of software quality, software maintainability. We further extend our investigation by analyzing 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).
Method: 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.
Results: Our 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. 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. 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 maintainability.
Conclusion: 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.

1. Introduction

Software code reviews have been commonly employed by software development teams (Rigby and Bird, 2013; Stray et al., 2021; Baum et al., 2016, 2017; Rigby et al., 2014) for decades to help assure software quality (Krutauz et al., 2020). 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 (Thongtanunam et al., 2015; MacLeod et al., 2018; Bavota and Russo, 2015; Bacchelli and Bird, 2013; Rigby et al., 2012), improving code quality (MacLeod et al., 2018; Bavota and Russo, 2015; Bacchelli and Bird, 2013; Rigby et al., 2012; Bosu et al., 2015), and knowledge transfer (Bacchelli and Bird, 2013; MacLeod et al., 2018).

Software quality is a broad term encompassing many aspects (Denning, 1992; Jørgensen, 1999). Code reviews are meant to address several of them. Most obviously, the code review process is meant to reduce defects (Bacchelli and Bird, 2013; MacLeod et al., 2018), which improves reliability, an important quality attribute (Galster et al., 2014). Another important quality attribute is maintainability (Broy et al., 2006; Baggen et al., 2012), one indicator of which is the presence of code smells (Yamashita and Counsell, 2013; Khomh et al., 2012; Kruchten et al., 2012; Palomba et al., 2018a). 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 internal code quality, an important component of maintainability (Palomba et al., 2018a; Soh et al., 2016; D'Ambros et al., 2010; Yamashita and Moonen, 2013; Zazworka et al., 2011). Code reviews can improve maintainability by detecting and removing code smells and other sub-optimal coding constructs that would hinder future maintenance.

While the benefits of code reviews are well known, there are common ways in which they can be performed less than optimally. These

☆ Editor: Prof. Neil Ernst.
* Corresponding author.
E-mail address: erdem.tuna@bilkent.edu.tr (E. Tuna).

https://doi.org/10.1016/j.jss.2024.112101
Received 3 February 2024; Received in revised form 21 April 2024; Accepted 16 May 2024
Available online 20 May 2024
0164-1212/© 2024 Published by Elsevier Inc.

sub-optimal processes have been characterized in prior work into code review process smells (CRPSs), which can be detected by examining data from code reviews (Doğan and Tüzün, 2022).

Intuitively, the presence of CRPSs is likely to have a negative impact on internal code quality (since code reviews are meant to positively impact internal 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), particularly the role of process debt, of which CRPSs are one type. We describe this connection to the TD literature in Section 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), in the sense that smelly code reviews fail to prevent or remove code smells. It is this causal relationship that we investigate in this work. Thus, the aim of the study described here is to understand the impact of code reviews on software maintainability.

To carry out this investigation, we conduct an analysis of data mined from open-source software projects that identify the code smells in pull request code that was introduced or removed during code reviews, as well as smelly and non-smelly code reviews. We also conduct a focus group study to query software practitioners specifically about the relationships they see between code smells and code review process smells. We address the following research questions:

RQ1: Do code reviews eliminate or reduce the number of code smells in pull request code?

RQ2: Do higher-quality code reviews lead to greater reduction in code smells than lower-quality code reviews, as measured by the presence of CRPS?

RQ3: Does code with a history of low-quality reviews exhibit higher levels of code smells?

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

We aim not just to investigate code smell changes during code reviews but also investigate the relationship between code smells and CRPSs by providing 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.

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

2. Background and related work

Our study brings together several bodies of software engineering literature. Code reviews have a long history of work that both defines review practices and studies their effects. In Section 2.1, we review this background and introduce the specifics of the code review process in the context we studied it (i.e., in open-source projects). Technical debt is another long-standing thread of work in software engineering research (although not as long as code reviews). We review, in Section 2.2, some basic concepts related to the debt metaphor, in particular, the idea of process smells. Finally, in Section 2.3, we introduce Code Review Process Smells (CRPSs), which bring the code review and technical debt threads of work together.

2.1. Modern code reviews

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

2.1.1. Motivations

Studies that try to understand the purposes of a code review process and its benefits generally utilize qualitative techniques such as interviews (Bacchelli and Bird, 2013; MacLeod et al., 2018; Baum et al., 2016; Sadowski et al., 2018), surveys (MacLeod et al., 2018; Sadowski et al., 2018; Bacchelli and Bird, 2013; Turzo and Bosu, 2024; Cunha et al., 2021), and analysis of pull request comments (Bacchelli and Bird, 2013; Turzo and Bosu, 2024). Quantitative pull request data analysis is also used (Sadowski et al., 2018). These studies agree that increasing internal code quality is considered one of the primary reasons for conducting code reviews (Bacchelli and Bird, 2013; MacLeod et al., 2018; Baum et al., 2016; Sadowski et al., 2018; Turzo and Bosu, 2024). Finding bugs (Bacchelli and Bird, 2013; MacLeod et al., 2018; Baum et al., 2016; Sadowski et al., 2018; Turzo and Bosu, 2024), design and solution evaluation (Bacchelli and Bird, 2013; Baum et al., 2016), knowledge transfer (Bacchelli and Bird, 2013; MacLeod et al., 2018; Cunha et al., 2021), educational purposes (Sadowski et al., 2018; Baum et al., 2016; Cunha et al., 2021), and code convention checks (Sadowski et al., 2018) are the other important driving factors.

In our study, we are interested in analyzing the internal code quality aspect of the code review process, in particular from the perspective of improving maintainability as evidenced by code smells.

2.1.2. What do reviewers find?

Given our focus on the relationship between code reviews and maintainability, we focus on literature that investigates similar themes. Two studies, one by Mäntylä and Lassenius (2009) and the other by Beller et al. (2014), each examined a corpus of code review results and both found that 75% of the identified defects were categorized as evolvability defects. The first of these studies involved commercial developers in a company and students, while the second one studied OSS code review data.

Alomar et al. (2022) took another approach to investigate maintainability by focusing on 5505 refactoring-related code reviews on OpenStack projects. One of the emerging themes they identified among these reviews is quality. The theme comprises reviewer suggestions on adherence to coding conventions, avoiding code smells, and correctness of design pattern practices, all of which are related to maintainability.

Han et al. (2022) 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 1539 smell-related code review comments (out of 25,315) from open-source projects, showing that code smell identification was not a concern of the vast majority of reviews. The root cause of most of the smells was violations of conventions, and the authors took the required actions to remove the smells most of the time when reviewers suggested a fix.

In summary, there is support in the literature that maintainability issues are a subject of concern in code reviews in a variety of contexts, although studies differ as to how prevalent these concerns are expressed.

2.1.3. What really changes?

The quantitative part of our analysis focuses on the question of whether or not code smells are resolved as a result of code reviews.

Several other studies have also quantitatively studied the effects of code reviews in terms of resulting code changes.

Two such studies (Panichella et al., 2015; Han et al., 2020) addressed this question by examining changes in the numbers of warnings and violations from static analysis tools before and after code reviews in Java projects. Like our study, both of these studies analyzed data at the individual code review level as well as at the cumulative level over time. Both studies found little to no change in the number or density of static analysis violations due to individual code reviews. However, the two studies found different effects over time. Panichella et al. (2015) found that the number of warnings decreased over time, while Han et al. (2020) found an increase.

Lenarduzzi et al. (2021) took a slightly different approach to investigating the effects of code reviews on the code. Based on an analysis of 28 Java projects and 36K pull requests, they investigated the relationship between violations detected by the PMD tool and the acceptance of the pull requests. Their analysis, supported by a manual investigation of a large sample of accepted and rejected pull requests, concluded that no such relationship exists. In fact, they reported that none of the pull requests they inspected manually were rejected for code quality reasons.

Even more closely related to the goals of our study, Pascarella et al. (2019) analyzed the effects of code reviews on code smell severity which is a measure of how far away the code smell instance is from the thresholds that define the smell. Their analysis of 21K code reviews from seven Java projects in the CROP dataset was based on comparing the code smell severity in the initial patch set to that of the final one after the review process ended. They employed heuristic-based code smell detection techniques and considered six code smell types. Their results suggest that the code smell severity decreases in only 4% of the code reviews under consideration. They also conducted a manual analysis of a sample of the code reviews and found that 95% of decreases are due to side effects of unrelated changes, not by an intentional effort to reduce the code smells.

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 Panichella et al. (2015), Han et al. (2020), and Lenarduzzi et al. (2021). Although our scope is narrower, it eliminates potentially irrelevant or unrelated 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

Many researchers have put effort into understanding the cumulative effect of the code review process on software quality and development. Davila and Nunes (2021) divided software quality concerns into the categories of code quality and product quality. Here, code quality corresponds to following the best practices in design and programming (what we and others refer to as internal code quality), whereas product quality refers to the reduced number of bugs or security defects (Davila and Nunes, 2021). Studies of code reviews and code quality include investigation of design anti-patterns (Morales et al., 2015), which showed that both code review coverage and participation are somewhat negatively associated with the occurrence of anti-patterns. Uchoa et al. (2020) also focused on code quality as well as software design degradation. They found that review comments pointing out design improvements help reduce or avoid design degradation, but discussions of such issues did not.

Examples of work focusing on product quality include the work of McIntosh et al. (2016) and Shimagaki et al. (2016), who found a relationship between post-release defects and code review metrics (review coverage, participation, and expertise).

The existing literature indicates a potential relationship between the code review process and several aspects of software quality. In our study, we are interested in finding the effect of code review practices on internal code quality.

2.2. Debt concept in software engineering

In general, debt provides interim benefits and causes detrimental consequences later. Financial debt is familiar to most people and serves as the basis of the debt metaphor as it is applied to software engineering. The existing literature explores the technical (Alves et al., 2016; Kruchten et al., 2012), social (Tamburri et al., 2015, 2013), and process (Martini et al., 2019; Alves et al., 2014) debt concepts to varying extents.

2.2.1. The technical debt metaphor

Technical debt (TD) has been defined and refined several times in the literature since it was initially introduced by Cunningham (1992). It is a set of design and implementation choices that bring ad hoc benefits but incur costly changes in the future (Avgeriou et al., 2016). Li et al. (2015) identified ten types of TD, one of which is code debt. Code debt is further defined (Tom et al., 2013), as “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” (Rios et al., 2018). Code smells are a primary indicator of code debt (Izurieta et al., 2012). In our study, we use code smells as an indicator of code debt and, by extension, internal code quality.

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. 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 difficult and complicated to capture. Some work by TD researchers has attempted to do this in limited ways (e.g., Nugroho et al., 2011; Ampatzoglou et al., 2016), but in general, the estimation of TD interest has been elusive.

2.2.2. Process debt

TD is based on a metaphor that intuitively fits many software engineering-related phenomena. One such phenomenon is process debt, 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. Process debt has been discussed in the context of TD, but not as a type of TD (Martini et al., 2020). Instead, it is seen as a phenomenon that is a precursor to TD. In this paper, 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.

Earlier research (Codabux and Williams, 2013; Li et al., 2015) hinted at the existence of process debt (PD) without mentioning the term. Alves et al. (2014) and Alves et al. (2016) explicitly identified PD, framing it as a type of TD concerned with ineffective software development processes. Later, two leading studies (Martini et al., 2019, 2020) distinguished PD as a debt type other than technical or social debt.

Martini et al. (2019) investigated different debt types (technical, social, and process) in agile teams via a case study to observe in-team and inter-team retrospective meetings. They first defined PD as “a sub-optimal activity or process that might have short-term benefits but

generates a negative impact in the medium-long term." They found that PD is widely discussed in retrospective meetings, based on the number of issues recorded. They also found that managing PD requires more coordination between different teams than the other debt types.

Following Martini et al. (2019, 2020) conducted another study to explore PD systematically, addressing PD causes, consequences, and types, in the end, arriving at a formal PD definition. Based on workshops and interviews with 16 practitioners from four companies, they identified process competencies, value neglect, and organizational issues as 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 practical example of the infrastructure debt was given on the use of version control tools. A company opted to use a version control tool and it had worked well within their processes for several years. However, after adopting CI/CD processes, the tool caused a slow deployment process and affected the delivery schedule. A holistic evaluation of the results led to the following PD definition: "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 a 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 this definition from Martini et al. (2020) and posit that code review process smells (CRPSs, described in Section 2.3) constitute process debt.

PD, although not actually a type of TD, 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, or more conveniently, 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. With TD, as with financial debt, the principal is "paid back" just once. The loan is repaid, or the code is refactored, and the debt disappears. With PD, specifically the CRPSs 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" PD, as the cost of paying it back extends forever into the future.

In this study, however, we are addressing the interest associated with PD, specifically CRPSs. Because PD is not TD per se, but a precursor to it, the interest associated with PD is not cost, but the TD itself, i.e., the TD that the PD causes. In our study, we use code smells as an indicator of the TD that the code review process smells lead to. Thus, we are studying the relationship between code review process 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 PD.

2.3. Code review process smells

A large body of research has been established 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 (Sobrinho et al., 2021; Tufano et al., 2017), test code (Bavota et al., 2015; Pascarella et al., 2018), infrastructure as code (Rahman et al., 2019; Sharma et al., 2016), architecture (Fontana et al., 2017; Sas et al., 2022), or processes such as continuous integration (Zampetti et al., 2020; Vassallo et al., 2020), bug tracking (Qamar et al., 2022; Tuna et al., 2022), and code review (Doğan and Tüzün, 2022). As software development is collaborative, community smells are also identified in the literature (Tamburri et al., 2015, 2021; Caballero-Espinosa et al., 2023). In this paper, we are particularly interested in the studies about smells in the code review process.

Chouchen et al. (2021) explored anti-patterns (a neighbor concept to smells) in the code review process. They suggested that the anti-patterns could be an indicator of tense review culture, and thus they may slow down integration and lead to a decrease in software quality. The authors identified five anti-patterns based on the existing literature, then probed for them in a random sample of 100 code review instances from the OpenStack project. Their results showed that 67% and 21% of the code reviews contain at least one or two anti-patterns, respectively.

Doğan and Tüzün (2022) categorized bad practices in the code review process and identified them as smells, based on a multivocal literature review augmented by a survey and interviews with developers. The result is a taxonomy of seven code review process smells (CRPSs), specifying each smell's definition, root causes, potential side effects, and detection method. They then explored the quantitative evidence for the presence of CRPSs in eight open-source projects, and found them to varying degrees in different projects.

For our study, we decided to use the CRPS taxonomy of Doğan and Tüzün (2022), rather than that of Chouchen et al. (2021), because it is empirically based on both a review of the literature and developer opinion, and because it has been tested on more projects and code reviews. Also, our study design requires automated detection of CRPSs from code review data, which is not possible with some of Chouchen et al. (2021)'s anti-patterns.

Below, we provide a brief overview of the CRPSs identified in the work of Doğan and Tüzün (2022). The analyses in our study involve only 5 of these 7 CRPSs. We excluded the *looks good to me reviews* smell because there is no defined detection mechanism for the smell. Similarly, we excluded the *review buddies* smell because it is detected based on aggregate contributions of author–reviewer pairs and so it cannot be evaluated on a pull request basis.

Lack of Code Review. The benefits of code reviews cannot be realized if no code review is conducted. Pull requests identified with this smell are either not reviewed or are self-reviewed by the pull request author before merging into 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 not be able to devote enough review time to sufficiently cover the entire change.

Looks Good to Me Reviews. The code review process has sound benefits 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. However, like smells of all kinds, there are situations in which this smell indicates normal activity and is not a cause for concern. For example, in some projects, "Looks Good to Me" or "LGTM" is a common standard term to indicate that the code has in fact been reviewed and found to be in good shape.

Missing Context. To develop high-quality software efficiently, developers should maintain traceability between various software artifacts (Cleland-Huang et al., 2014). 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 (2013) 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

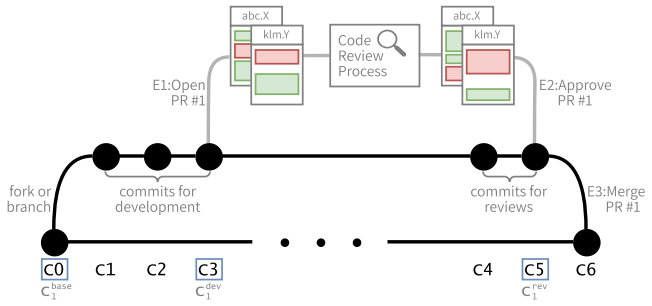


Fig. 1. Sample development overview. Commit history graph shows the sequential progress of commits, i.e., c0, ..., c6. Pull request events are represented with E1, E2, and E3.

increased review time and other developers being blocked if they are depending on the pull request. However, as with the "Looks Good to Me" CRPS, such a back-and-forth might not always be a bad thing, as in some cases it can indicate a teaching and learning exchange.

Review Buddies. There are many studies on the reviewer recommendation problem (Sülün et al., 2021; Yu et al., 2016) and its relevance to developers and their perceptions of the problem (Kovalenko et al., 2020). 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 significantly more often than they pair with other team members for code reviews, then this creates a *review buddies* smell. Buddy relationships could cause ineffective reviews and decrease shared code ownership, although in some cases they are standard practice in projects, especially if one team member (e.g. the project lead) is designated to perform most reviews.

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 (Sadowski et al., 2018) up to almost 21 hours (Rigby and Bird, 2013). A review time of more than 48 hours (Doğan and Tüzün, 2022) 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. Key analytical concepts

We establish some definitions and concepts used in our analysis procedures in this section. Each subsection develops direct or related background concepts for the research questions. Sections 3.1 and 3.2 describe some concepts that were used in the analyses that address all research questions. Section 3.3 relates to RQ1 and RQ4. Lastly, Section 3.4 is related to RQ3 and RQ4.

3.1. Key commit types

Key commits are the beacons of a pull request process for evaluating changes in 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 Fig. 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 visualized with c0 in Fig. 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 Fig. 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 Fig. 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 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 $c_p^{base} \rightarrow c_p^{dev} \rightarrow c_p^{rev}$.

3.2. 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 ***Delta Analysis of Code Smell Counts*** (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 ***Delta Analysis of Code Smell Densities*** (ΔCoS-D analysis). We provide a thorough explanation of the analyses below.

3.2.1. Common concepts

We start by defining the common terms and concepts of the two delta 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 define 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 between different KCs of a pull request. Considering the sequential nature of KCs in a pull request p, we establish 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 define $\kappa^c(f)$ to represent the total number of code smells in file f in commit c.

3.2.2. 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 (1)

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

where n is the number of files in the commit.

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

We adapt the formula and notation in (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 (2), (3), and (4), respectively.

$$\Delta K_p^{dev} = \left(\frac{K(c_p^{dev}) - K(c_p^{base})}{K(c_p^{base})} \right) * 100 \tag{2}$$

$$\Delta K_p^{rev} = \left(\frac{K(c_p^{rev}) - K(c_p^{dev})}{K(c_p^{dev})} \right) * 100 \tag{3}$$

$$\Delta K_p^{total} = \left(\frac{K(c_p^{rev}) - K(c_p^{base})}{K(c_p^{base})} \right) * 100 \tag{4}$$

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 our online supplementary appendix (Tuna et al., 2023b).

3.2.3. 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 (5).

$$\mu^c(f) = \frac{\kappa^c(f)}{\lambda^c(f)} \tag{5}$$

Then, $M(c)$ is the code smell density over all files in commit c as in (6)

$$M(c) = \frac{\sum_{i=1}^{n} \kappa_c(f_i)}{\sum_{i=1}^{n} \lambda_c(f_i)} \tag{6}$$

where n is the number of files.

Similar to $K(c_p^x)$, we adapt (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 (7), (8), and (9), respectively.

$$\Delta M_p^{dev} = \left(\frac{M(c_p^{dev}) - M(c_p^{base})}{M(c_p^{base})} \right) * 100 \tag{7}$$

$$\Delta M_p^{rev} = \left(\frac{M(c_p^{rev}) - M(c_p^{dev})}{M(c_p^{dev})} \right) * 100 \tag{8}$$

$$\Delta M_p^{total} = \left(\frac{M(c_p^{rev}) - M(c_p^{base})}{M(c_p^{base})} \right) * 100 \tag{9}$$

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 our online supplementary appendix (Tuna et al., 2023b).

3.2.4. 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, (3), (4), (8), and (9) are affected. We cannot calculate the values of the ΔK_p^{rev} and ΔM_p^{rev} parameters. So, actually, (3) and (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) and (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 (2), (3), and (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 (2), (3), and (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 (7), (8), and (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.

3.3. Classifying the code review outcomes on code smells

Code smells may increase or decrease as a result of the code review process. We classified the code review outcomes 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 internal code quality point of view. In this way, we classify the 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 3.2) to classify the code review outcome. We first evaluate ΔM_p^{rev} to decide the goodness of the review 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 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.

[2] We use *increase*, *no change*, and *decrease* as explained in our online supplementary appendix (Tuna et al., 2023b).

[3] Neutral with Review Commit (c_p^{rev}).

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

Table 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 our online supplementary appendix (Tuna et al., 2023b).

- 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.

Table 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.

3.4. 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 RQ3, 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 2.3. 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 (10).

$$\rho^{crps}(f) = \frac{|SPR^{crps}(f)|}{|APR(f)|} \tag{10}$$

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. 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 Fig. 2.

4.1. Case project selection

We selected 10 projects to conduct the analyses described in Section 4.3. 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. GitHub is a widely used platform by companies and researchers (MacLeod et al., 2018).
2. The project has 3000 or more stars. Amanatidis et al. (2020) also used this threshold to select subject projects.
3. The project has 5000 or more closed pull requests.
4. The project's main language is JavaScript, which constitutes at least 90% of the repository code. This choice was driven by the limitations of code smell detection capabilities. SonarQube can detect code smells in various programming languages, but for compiled languages (e.g., Java) it requires compilation of the entire repository which requires custom configurations and more computation power and time than we had available. 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 is widely supported in SonarQubeand does not have these limitations.
5. The project is owned by an organization, i.e., an open-source community or a company. The rationale behind this criterion is that such entities would ensure that software development processes are followed as much as possible (Li et al., 2022; Munaiah et al., 2017).

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[5] library to obtain the data from GitHub API.[6] 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 2 shows the projects together with their characteristics. We share the list of selected projects with the numerical details in our online replication package (Tuna et al., 2023a).

4.2. Data collection

We detail the data collection processes for RQ2 and RQ3 in this section. RQ2 depends on three types of data per project: the KCs of pull requests, SonarQube code smell data, and the CRPSs associated with each pull request. On the other hand, to explore RQ3 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.

4.2.1. Detection of code review process smells

Both RQ2 and RQ3 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 (Dueñas et al., 2018) tool. We relied on the algorithms from Doğan and Tüzün (2022) to detect CRPSs at the pull request granularity. We saved the smell results for each project in a PostgreSQL[7] database. We share the SQL dump of the resulting database in our online replication package (Tuna et al., 2023a).

[5] https://github.com/PyGithub/PyGithub (Accessed on 19 Dec 2022).
[6] https://docs.github.com/en/rest (Accessed on 19 Dec 2022).
[7] https://www.postgresql.org/ (Accessed on 19 Dec 2022).

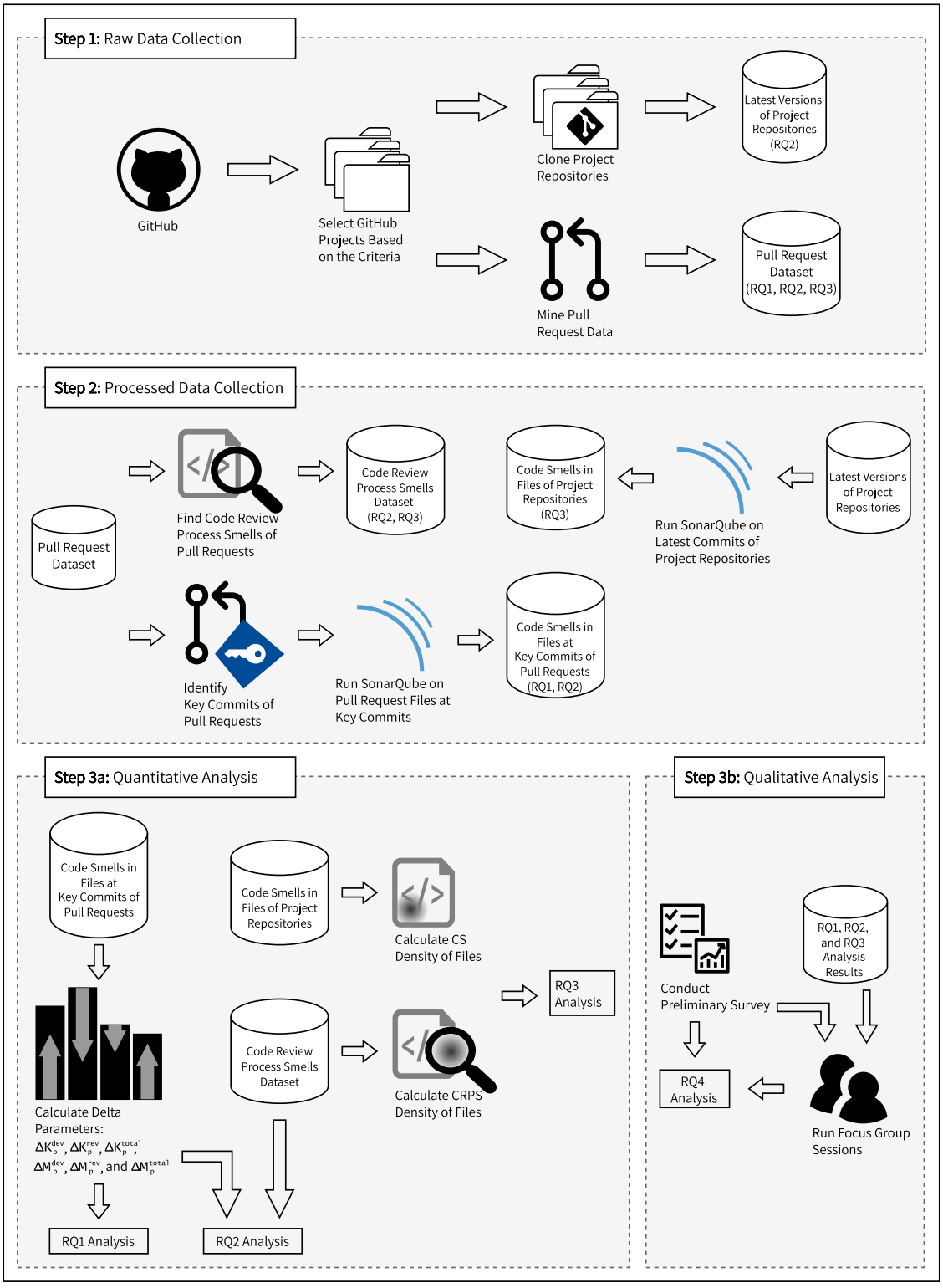


Fig. 2. Simplified overview of our study design. Some connections and steps are omitted for clarity. Inspired from Alomar et al. (2022).

The analysis of RQ3 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 (Tuna et al., 2023a). 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

Table 2
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	1118	7999	7922	789
CESIUM	CESIUMGS	687,138	2947	5530	5507	4876
ESLINT	ESLINT	355,030	1222	6314	6299	5180
GHOST	TRYGHOST	110,236	1256	8585	8560	6624
OPENLAYERS	OPENLAYERS	140,170	931	8242	8219	7203
PDF.JS	MOZILLA	130,135	320	6754	6738	5749
REACT	FACEBOOK	356,849	1807	12,357	12,074	8358
SHIELDS	BADGES	70,312	1155	5893	5862	4887
STRAPI	STRAPI	215,186	2564	6114	6048	4768
WEBPACK	WEBPACK	152,330	5938	5784	5650	4169
	Total	2,449,038	19,258	73,572	72,879	52,603

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
```

used files existing in the latest state[8] of the repositories in the RQ3 analysis.

4.2.2. Identifying key commits

To answer RQ2, 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.

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[9] 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 (Git Garbage Collection Documentation, 2022). 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.[10]

Table 3
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	4314	1266	29.3
ESLINT	5167	833	16.1
GHOST	6159	374	6.1
OPENLAYERS	7083	562	7.9
PDF.JS	5724	240	4.2
REACT	8168	1551	19.0
SHIELDS	4861	2970	61.1
STRAPI	2943	1404	47.7
WEBPACK	3582	550	15.4
All	48,704	9838	20.2

Readers may notice that the merged PR numbers in Table 2 and the number of PRs with KCs in Table 3 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 4.2.3.

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}.

Pull Requests with and without *rev* Key Commits. For each project, we identified all the key commits of each pull request. Those pull requests with any identified key commits are called *pull requests with key commits*. For convenience, we further identify the pull requests with a valid c_p^{rev}, i.e., not *Null* value, and call them as *pull requests with* c_p^{rev}. These pull requests not only have the c_p^{base} and c_p^{dev} commits but also have c_p^{rev} commits, meaning that changes were made during the review process. Table 3 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. However, we conducted our analysis using all PRs with KCs. We share the key commit data of pull requests in JSON format for each project in our online replication package (Tuna et al., 2023a).

4.2.3. Finding code smells

We detected code smells for RQ2 and RQ3 analyses using SonarQube. For RQ2, we found code smells in files modified in the pull requests of each project. On the other hand, for RQ3, 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[11] Campbell and Papapetrou (2013) to find code smells for our analyses. It is a widely used

[8] As mentioned in Section 4.3.3, *latest state* refers to the merge commit of the latest pull request in our dataset.

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

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

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

Table 4
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.

tool in industry (SonarSource Customer List, 2022a) and in academic research (Conejero et al., 2018; Lenarduzzi et al., 2020; Digkas et al., 2022). 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 (SonarQube Metric Definitions, 2022) based on defined rule sets (SonarQube Rules, 2022). It can also classify the severity of a code smell according to a 5-level scale (SonarQube Issue Severity Levels, 2022; Lenarduzzi et al., 2020). We excluded from our analysis code smells with the lowest severity level, "Info", and used "Minor", "Major", "Critical", and "Blocker" levels.

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 4.2.2, 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.[12] So similarly, we employed a two-pronged 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, 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[13] 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.[14] 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 RQ2. In our online replication package, we share the SQL dump of the resulting database (Tuna et al., 2023a).

Finding Code Smells in a Project Repository. To answer RQ3, 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 (Tuna et al., 2023a).

4.3. Analysis

In this section, we describe the process we used to analyze the data to address each of the first three research questions. We also provide an explanation of how we used correlations throughout our analysis.

4.3.1. Classifying the effect of each code review on code smells (RQ1)

We calculated ΔM_p^{dev}, ΔM_p^{rev}, and ΔM_p^{total} parameters per pull request. Then, we classified each code review according to its "goodness", using the review classes defined in Section 3.3.

4.3.2. Influence of the presence of CRPSs on internal code quality (RQ2)

We conducted this analysis to investigate whether smelly code reviews result in a more smelly codebase. On the one hand, with RQ1, 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 entire pull request lifecycle) parameters for each pull request and project using the code smell results and employing ΔCoS-D analysis. On the other 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. 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 4.3.4.

4.3.3. Effect of accumulated code review process smells (RQ3)

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 (Tuna et al., 2023a). 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 RQ3 with the correlation methods in Section 4.3.4.

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

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

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

4.3.4. 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 Schober and Schwarte (2018) 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 (Fowler, 1987). The method is also used in many other software engineering papers (Falessi and Kazman, 2021).

A recent study by Chatterjee (2021) 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 relationships between X and Y. Second, like SC, CC is robust against different data distributions.

For both correlation methods, we utilized the implementations in R (R Stats Package, 2022; XICOR CRAN package, 2022).

4.4. Focus group study

In order to better interpret the quantitative results from RQ1, RQ2, and RQ3 and increase our data diversity (Seaman, 1999), 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 (Palinkas et al., 2015) and also helps triangulate our quantitative results with practitioners' views and discussions.

4.4.1. 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. The survey instrument first presented an explanation of each CRPS we included in our study. Then, two focus group questions (FQs) were asked for each CRPS, as presented below.

FQ1: Do you think 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?

While there is a small risk of confirmatory bias introduced by the wording of the focus group questions above, we attempted to word these questions in as natural a way possible, to elicit natural responses. Attempts to reword the questions to remove bias resulted in questions that were more difficult to understand. Further, we used the survey responses only as prompts to initiate discussion during the focus group sessions. The more relevant data and results came from the subsequent discussion during the focus group sessions.

Session Outline. We followed the guidelines of Krueger and Casey (2014) 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 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 (Tuna et al., 2023a).

4.4.2. 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 (Palinkas et al., 2015), in which we tried to diversify the participants' years of experience and company domain. We contacted software practitioners in the authors' professional circle and selected 10 participants based on availability. Table 4 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 Soliman et al. (2021). Topic coverage was the first author's responsibility whereas the third author ensured 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 by obtaining the participants' verbal consent. The moderators also wrote down field notes during the sessions.

4.4.3. Data analysis

We transcribed the audio recordings to text using Microsoft Word's transcription feature (Microsoft Word Transcription, 2022). The moderator authors proofread the transcriptions. 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. For instance, "(Pull request descriptions matter because) I can write classes that are suitable for all styles, and comply with all the rules, but would render the app inoperable." (P6) reflects on why the *missing context* smell could hinder the code review process and lead to an increase in code smells. On the other hand, for the same smell, "There is no significant connection between code smell and the pull request context. We can see the same code smell in any code piece." (P10) expresses disagreement for FQ1 and FQ2. Both reviews are concerned with the *missing context* smell. The former view contributes to the *"hinderCodeReviewBenefits"* and *"increaseCodeSmells"* themes whereas the latter contributes to the *"notHinderCodeReviewBenefits"* and *"notIncreaseCodeSmells"* themes in our coding scheme. We used the Taguette[15] tool to manage our coding.

5. Results

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

5.1. Code review classifications (RQ1)

As introduced in Section 3.3, we classified the code review outcome from the code smell perspective using the ΔM_p^{dev}, ΔM_p^{rev}, and ΔM_p^{total} parameters of ΔCoS-D analysis. Fig. 3 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, when code reviews

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

Table 5
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*[a] column shows the number of pull requests excluded from the smell detection. Project names are abbreviated for spacing purposes.

Project	PR data[b]	Lack of review			Large changesets			Missing context			Ping-Pong			Sleeping reviews		
		Count	Ratio	NA	Count	Ratio	NA	Count	Ratio	NA	Count	Ratio	NA	Count	Ratio	NA
ANG.	All	439	55.6%	7210	481	6.0%	0	147	18.6%	7210	22	0.3%	0	4222	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	3104	63.7%	654	1003	18.1%	0	303	6.2%	654	246	4.4%	0	2005	36.4%	23
	KC	2830	65.6%	0	691	16.0%	0	274	6.4%	0	189	4.4%	0	1414	32.8%	0
ESL.	All	1689	32.6%	1134	371	5.9%	0	795	15.3%	1134	110	1.7%	0	2811	44.6%	15
	KC	1683	32.6%	0	299	5.8%	0	790	15.3%	0	96	1.9%	0	2213	42.8%	0
GHO.	All	5125	77.4%	1961	629	7.3%	0	136	2.1%	1961	31	0.4%	0	2943	34.4%	25
	KC	4781	77.6%	0	399	6.5%	0	116	1.9%	0	24	0.4%	0	1842	29.9%	0
OPE.	All	3064	42.5%	1039	484	5.9%	0	547	7.6%	1039	34	0.4%	0	1947	23.7%	23
	KC	2973	42.0%	0	366	5.2%	0	541	7.6%	0	23	0.3%	0	1335	18.8%	0
PDF.	All	2565	44.6%	1005	424	6.3%	0	853	14.8%	1005	8	0.1%	0	2271	33.7%	16
	KC	2549	44.5%	0	311	5.4%	0	842	14.7%	0	4	0.1%	0	1774	31.0%	0
REA.	All	2993	35.8%	3999	1122	9.1%	0	674	8.1%	3999	181	1.5%	0	4285	35.5%	283
	KC	2920	35.7%	0	706	8.6%	0	667	8.2%	0	140	1.7%	0	2193	26.8%	0
SHI.	All	354	7.2%	1006	156	2.6%	0	305	6.2%	1006	128	2.2%	0	1489	25.4%	31
	KC	341	7.0%	0	127	2.6%	0	304	6.3%	0	124	2.6%	0	1090	22.4%	0
STR.	All	598	12.5%	1346	841	13.8%	0	346	7.3%	1346	125	2.0%	0	2522	41.7%	66
	KC	343	11.7%	0	197	6.7%	0	202	6.9%	0	68	2.3%	0	1277	43.4%	0
WEB.	All	2253	54.0%	1615	413	7.1%	0	49	1.2%	1615	36	0.6%	0	2239	39.6%	134
	KC	1981	55.3%	0	210	5.9%	0	48	1.3%	0	26	0.7%	0	1156	32.3%	0
Total	All	22,184	42.2%	20,969	5924	8.1%	0	4155	7.9%	20,969	921	1.3%	0	26,734	36.7%	693
	KC	20,775	42.7%	0	3383	6.9%	0	3903	8.0%	0	697	1.4%	0	14,591	30.0%	0

[a] Due to algorithm precondition checks (Doğan and Tüzün, 2022), some PRs are not evaluated for this particular smell. Those PRs are labeled with NA.
[b] Readers may refer to Tables 2 and 3 for the total number of PRs and the number of PRs with KCI, respectively.

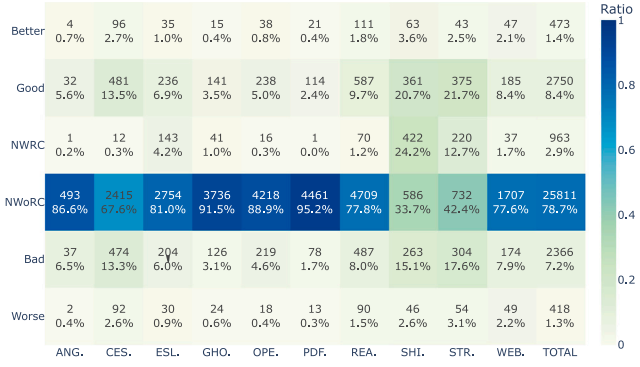


Fig. 3. Overall code review classification results. For each review class (y-axis) per project (x-axis), the number of pull requests and their ratios are shown.

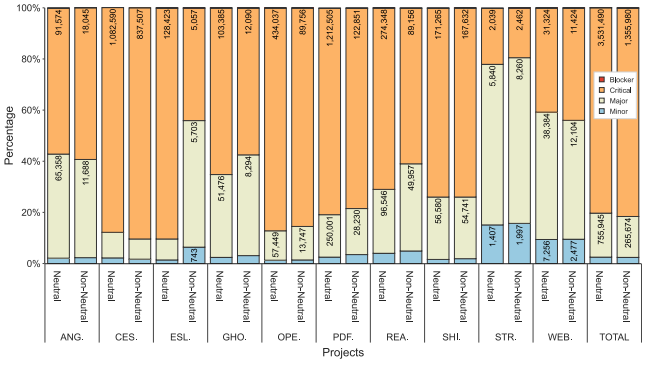


Fig. 4. Code smell severity distribution per project. The "Neutral" group involves NWoRC and NWRC reviews, whereas the "Non-Neutral" includes Better, Good, Bad, and Worse reviews. The numbers inside the bars indicate the count of code smells for each severity level. "Blocker" severities are present but may be difficult to discern visually due to their low frequency (less than 0.5%).

do affect code smell densities, they are equally likely to increase them as decrease them.

We also analyzed how code smell severity differs among neutral (i.e., NWoRC and NWRC) and non-neutral (i.e., Better, Good, Bad, and Worse) reviews. Per each review, we aggregated the number of code smells for each code smell severity. As shown in Fig. 4, we did not observe a notable difference, suggesting a similar code smell distribution between the two groups.

Finding 1. 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, it increases the code smell density as often as it decreases it.

5.2. Code smells and code review process smells

After finding that, on average, code reviews seem to have little effect on the number or density of code smells, we analyzed whether either smelly or non-smelly reviews as a group might have a different effect.

5.2.1. Prevalence of code review process smells

We analyzed all projects to find CRPSs in order to address both RQ2 and RQ3. Complete data on the occurrence of CRPSs in the projects is presented in Table 5. 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 projects. *Missing context* smell occurrence values are similar and range from 1.3% (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 2. 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.

5.2.2. Relationship between code review process smells and code smells (RQ2)

We analyzed the relationship between the existence of CRPSs and the prevalence of code smells (ΔM_p^{total} and ΔK_p^{total}). The detailed correlation results are presented in our online supplementary appendix (Tuna et al., 2023b). 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.

To observe whether the RQ1 and RQ2are related or not, We further investigated the distribution of the smelly code reviews ratio (similar to Table 5) with respect to the code review classifications (as in Fig. 3). However, we did not obtain a notable insight.

Finding 3. 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.

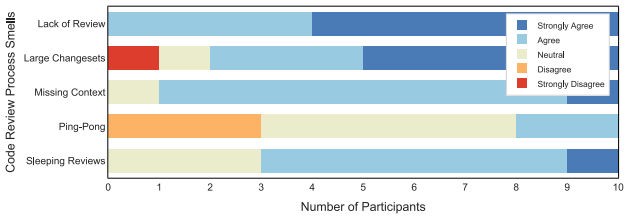
5.2.3. Aggregate effect of code review process smells on code smells (RQ3)

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 (i.e. the lower the quality of the code reviews conducted on that 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. We provide the detailed values of the correlation coefficients in our online supplementary appendix (Tuna et al., 2023b).

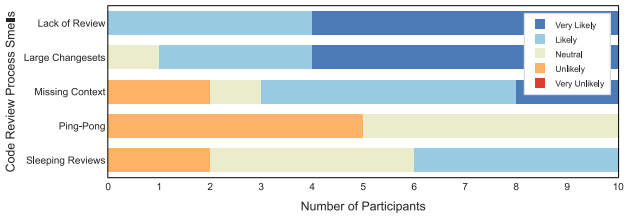
The correlation between *lack of review* density and code smell density is statistically significant for ANGULAR.JS, CESIUM, ESLINT, and OPENLAYERS projects. The former two projects' SC and CC coefficient values indicate a positive and weak association. On the other hand, SC values from the last two 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 4. 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.



(a) FQ1: Do you think 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?

Fig. 5. Focus group participants' view on FQ1 and FQ2.

5.3. Focus group with practitioners (RQ4)

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

5.3.1. On FQ1 and FQ2

Fig. 5 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. Three out of ten participants indicated the same for *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 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 5. 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 on whether CRPSs could lead to increased code smells in the code, but they generally find it likely that CRPSs other than *ping-pong* and *sleeping reviews* lead to such an increase.

5.3.2. 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 RQ2 and RQ3. Then, we asked exploratory questions to initiate conversations. Lastly, we showed the RQ1'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.[16] 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 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 6. Almost all participants think that open-source developers try to make high-quality code contributions to their projects, that is reducing the need for code reviews to address internal code quality.

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 7. 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, which may lead developers to not seriously check or care about code smells in code reviews, if they rely on linters during development 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

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

Table 6
The existence of recommendations and mentions on internal 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 (Angular.js Contribution Guideline, 2022)	Yes	Yes	No	Yes	No	No
CESIUM (Cesium Contribution Guideline, 2022)	Yes	Yes	Yes	Yes	No	No
ESLINT (Eslint Contribution Guideline, 2022)	Yes	Yes	No	Yes	No	No
GHOST (Ghost Contribution Guideline, 2022)	Yes	Yes	No	Yes	No	No
OPENLAYERS (Openlayers Contribution Guideline, 2022)	No	No	No	Yes	No	No
PDF.JS (Pdf.js Contribution Guideline, 2022)	Yes	Yes	No	No	No	No
REACT (React Contribution Guideline, 2022)	Yes	Yes	No	Yes	No	Yes
SHIELDS (Shields Contribution Guideline, 2022)	No	Yes	Yes	Yes	No	No
STRAPI (Strapi Contribution Guideline, 2022)	Yes	Yes	No	Yes	No	No
WEBPACK (Webpack Contribution Guideline, 2022)	Yes	Yes	No	Yes	No	No

know that linters and SonarQube are used in their project. 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 8. The participants consider that linters and software quality tools are effective in removing internal code quality problems in code before code review.

5.3.3. Summary

Overall, the focus groups let us obtain the views of practitioners in a discussion medium. The participants not only expressed their own opinions but also built upon and challenged others' views. As mentioned previously, the questions posed in the preliminary survey were potentially biasing, and in fact the survey responses were tilted towards confirming the propositions that code reviews process smells have an overall negative effect on internal code quality, for most smells. While not intentional, this trend in the survey responses provided focus group participants with an opportunity to think more deeply about the subject, and provide deeper explanations, when presented with quantitative results that contradicted their opinions as expressed in the survey. This revealed a wider range of interpretations for the RQs, including a more nuanced and context-dependent understanding of when and where CRPS's are likely to have an effect on code smells, particularly related to open source contexts. We believe that combining large-scale analysis with targeted discussions with developers increased the depth of our understanding of the quantitative results.

6. Discussion

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

6.1. Code smell density changes in code reviews

Concerning RQ1, we found that roughly 8 out of 10 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 internal 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 Digkas et al. (2022), we investigated suggestions or practices aiming to achieve internal code quality by visiting the repository guidelines.[17] 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 Digkas et al. (2022). The results of our investigation, in Table 6, show that 8 out of 10 projects contain suggestions for achieving code quality. Those eight projects explicitly suggest using code linters, e.g., ESLint,[18] 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 (Tomasdottir et al., 2020). 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 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. Our study found that roughly 8 out of 10 code reviews remain neutral (did not result in an increase or a decrease) in code smells. This is consistent with the finding of Han et al. (2020), 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 focus 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 to code smells in code reviews.

The code smell perception and knowledge of developers have been subjects of interest in literature. Researchers have explored how developers identify and understand these indicators of potential code issues. For example, Palomba et al. (2014) 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. De Mello et al. (2019) found that code smell detection is associated with developers' individual skills. Another study by Palomba et al. (2018b) 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 (De Mello et al., 2019), and developers may not be aware of some code smells. Thus, we posit that developers being unaware of code smells or not focusing on their detection could be the reason behind not identifying code smells during reviews.

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

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

6.2. 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 5 shows. This is in line with the findings in Doğan and Tüzün (2022), 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.[19] We found a lower prevalence of the *missing context* smell (highest rate of 18.2%, vs. 44.2% in Doğan and Tüzün (2022)) and the *ping-pong* (4.4% vs. 11.5% in Doğan and Tüzün (2022)). We found the opposite for the *sleeping reviews* smell (23.7% vs. 10.8% in Doğan and Tüzün (2022)). In both studies, however, CRPSs were found to occur in the code review process of all projects studied.

For RQ2 and RQ3, we investigated the potential relationship between the 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 internal code quality. However, we think the potential effect of the process smells on other outcomes of the code review process are worth exploring.

6.3. 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 in our community for a long time and have been well studied in both depth and breadth. However, the literature on the 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 (Sobrinho et al., 2021; Amanatidis et al., 2020) 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 (Lussier, 2004; Serrano et al., 2004; Wasserman and Capra, 2007) 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.

[19] We compare the occurrence ratio of CRPSs in Doğan and Tüzün (2022) with *All* rows in Table 5.

6.4. Implications for practitioners

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

Improving and adhering to development guidelines. Table 6 shows that 8 out of 10 projects contain recommendations on code quality without explicitly referencing internal code quality. Our study showed that the internal code quality is not improved in 90.2% of code reviews (as in Fig. 3) considering the code smells. The analyzed 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 internal code quality. Such an improvement could encompass the augmentation of project guidelines to explicitly articulate the project's approach towards addressing instances or types of code smells.

Benefiting from linters while continuing code reviews. Tomasdottir et al. (2020) 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 (Tomasdottir et al., 2020). 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 internal 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 internal 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.

7. Threats to validity

We discuss the threats to the validity of our study and results by following the guidelines of Wohlin et al. (2012).

7.1. 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 systematic and reliable way to extract review-related data from comments. Further, manual extraction was infeasible due to the volume of comment data. We project that future studies may consider incorporating comments in reviews systematically in code review-related analyses.

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 (Paixao and Maia, 2019). 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 distorted. Unfortunately, we could not identify a systematic 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 (Jiang et al., 2021). 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.

7.2. 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, there is an expectation of zero false-positive instances (SonarQube Rules, 2022). Further, SonarQube is supported by both a company[20] 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 (Avgeriou et al., 2021) to identify and remediate technical debt.

We used the algorithms in Doğan and Tüzün (2022) to detect the code review process smells. The detection algorithms rely on configurable thresholds. We used the thresholds as they appear in the original paper (Doğan and Tüzün, 2022) because they are based on a literature search and a survey of developers.

7.3. Conclusion validity

Our quantitative data analysis depends on the theoretical fundamentals of our study, detailed in Section 3. 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 (Tuna et al., 2023a) 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.

7.4. External validity

There are five 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. To increase the generalizability, we selected our focus group participants as commercial developers. However, case studies on closed-source projects are still required to improve the generalizability of our conclusions. Secondly, we selected the focus group participants using the purposeful sampling method, to ensure a variety of perspectives. This involved contacting a broad range of developers with different levels of expertise. However, we note that the findings may have been different with a different set of focus group participants. Thirdly, we analyzed ten projects based on the mentioned selection criteria to get a broader view of the analysis results and diminish related concerns. We believe the sample size is sufficient to draw insights, given the high number of GitHub stars and merged pull requests in the analyzed projects, as per our selection criteria. However, to generalize our findings across all scenarios, further research with a larger and more diverse set of projects would be beneficial and could reinforce the validity of our results. Fourth, all our projects used JavaScript as the primary language, which is a limiting factor for generalizing the results to projects using other programming languages. JavaScript does not have these limitations, which made it suitable for our analysis. Finally, 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 (1999), could still be helpful.

8. Conclusion

We found that code smell density is unaffected by the code review process in 8 out of 10 code reviews. Further, our results show that the quality of the code review process does not change this. Code review process smells had 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.

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 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 internal code quality.

Improving internal code quality has historically been cited as one of the goals of code reviews (Bacchelli and Bird, 2013; MacLeod et al., 2018; Baum et al., 2016; Sadowski et al., 2018). 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 internal code quality improvement, at least in terms of code smells. Even bad (i.e., smelly) and good (i.e., smell-free) 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 internal 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 internal 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 internal code quality. This provides an opportunity for the community to reimagine the code review process, aligning

[20] SonarQube is a product of SonarSource Customer List (2022b).

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.

CRediT authorship contribution statement

Erdem Tuna: Writing – original draft, Visualization, Validation, Software, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Carolyn Seaman:** Writing – review & editing, Validation, Supervision, Project administration, Methodology, Conceptualization. **Eray Tüzün:** Writing – review & editing, Validation, Supervision, Project administration, Methodology, Investigation, Formal analysis, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data links are provided in the paper as Mendeley links (please refer to the “References" section for the figshare data links).

Acknowledgment

We thank anonymous reviewers who helped us improve our manuscript.

Appendix A. Supplementary data

Supplementary material related to this article can be found online at https://doi.org/10.1016/j.jss.2024.112101.

References

Alomar, E.A., Chouchen, M., Mkaouer, M.W., Ouni, A., 2022. Code review practices for refactoring changes: An empirical study on OpenStack; code review practices for refactoring changes: An empirical study on OpenStack. In: 2022 IEEE/ACM 19th International Conference on Mining Software Repositories. MSR, ACM, p. 13. http://dx.doi.org/10.1145/3524842.3527932.

Alves, N.S., Mendes, T.S., De Mendonça, M.G., Spinola, R.O., Shull, F., Seaman, C., 2016. Identification and management of technical debt: A systematic mapping study. Inf. Softw. Technol. 70, 100–121. http://dx.doi.org/10.1016/J.INFSOF.2015.10.008.

Alves, N.S., Ribeiro, L.F., Caires, V., Mendes, T.S., Spínola, R.O., 2014. Towards an ontology of terms on technical debt. In: Proceedings - 2014 6th IEEE International Workshop on Managing Technical Debt. MTD 2014, Institute of Electrical and Electronics Engineers Inc., pp. 1–7. http://dx.doi.org/10.1109/MTD.2014.9.

Amanatidis, T., Mittas, N., Moschou, A., Chatzigeorgiou, A., Ampatzoglou, A., Angelis, L., 2020. Evaluating the agreement among technical debt measurement tools: building an empirical benchmark of technical debt liabilities. Empir. Softw. Eng. 25 (5), 4161–4204. http://dx.doi.org/10.1007/S10664-020-09869-W/TABLES/8, URL: https://link.springer.com/article/10.1007/s10664-020-09869-w.

Ampatzoglou, A., Ampatzoglou, A., Avgeriou, P., Chatzigeorgiou, A., 2016. A financial approach for managing interest in technical debt. Lect. Not. Bus. Inf. Process. 257, 117–133. http://dx.doi.org/10.1007/978-3-319-40512-4{_}7/FIGURES/5, URL: https://link.springer.com/chapter/10.1007/978-3-319-40512-4_7.

Angular.js Contribution Guideline, 2022. Angular.js contribution guideline. URL: https://github.com/angular/angular.js/blob/master/CONTRIBUTING.md.

Avgeriou, P., Kruchten, P., Ozkaya, I., Seaman, C., 2016. Managing technical debt in software engineering (dagstuhl seminar 16162). In: DROPS-IDN/6693. Vol. 6, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, http://dx.doi.org/10.4230/DAGREP.6.4.110, URL: http://www.dagstuhl.de/16162.

Avgeriou, P.C., Taibi, D., Ampatzoglou, A., Arcelli Fontana, F., Besker, T., Chatzigeorgiou, A., Lenarduzzi, V., Martini, A., Moschou, A., Pigazzini, I., Saarimaki, N., Sas, D.D., De Toledo, S.S., Tsintzira, A.A., 2021. An overview and comparison of technical debt measurement tools. IEEE Softw. 38 (3), 61–71. http://dx.doi.org/10.1109/MS.2020.3024958.

Bacchelli, A., Bird, C., 2013. Expectations, outcomes, and challenges of modern code review. In: Proceedings - International Conference on Software Engineering. pp. 712–721. http://dx.doi.org/10.1109/ICSE.2013.6606617.

Baggen, R., Correia, J.P., Schill, K., Visser, J., 2012. Standardized code quality benchmarking for improving software maintainability. Softw. Qual. J. 20 (2), 287–307. http://dx.doi.org/10.1007/S11219-011-9144-9/TABLES/6, URL: https://link.springer.com/article/10.1007/s11219-011-9144-9.

Baum, T., Leßmann, H., Schneider, K., 2017. The choice of code review process: A survey on the state of the practice. In: Lecture Notes in Computer Science. In: LNCS, vol. 10611, Springer Verlag, pp. 111–127. http://dx.doi.org/10.1007/978-3-319-69926-4{_}9/FIGURES/4, URL: https://link.springer.com/chapter/10.1007/978-3-319-69926-4_9.

Baum, T., Liskin, O., Niklas, K., Schneider, K., 2016. Factors influencing code review processes in industry. In: Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering. Vol. 13-18-November-2016, Association for Computing Machinery, pp. 85–96. http://dx.doi.org/10.1145/2950290.2950323.

Bavota, G., Qusef, A., Oliveto, R., De Lucia, A., Binkley, D., 2015. Are test smells really harmful? An empirical study. Empir. Softw. Eng. 20 (4), 1052–1094. http://dx.doi.org/10.1007/S10664-014-9313-0/FIGURES/15, URL: https://link.springer.com/article/10.1007/s10664-014-9313-0.

Bavota, G., Russo, B., 2015. Four eyes are better than two: On the impact of code reviews on software quality. In: 2015 IEEE 31st International Conference on Software Maintenance and Evolution, ICSME 2015 - Proceedings. Institute of Electrical and Electronics Engineers Inc., pp. 81–90. http://dx.doi.org/10.1109/ICSM.2015.7332454.

Beller, M., Bacchelli, A., Zaidman, A., Juergens, E., 2014. Modern code reviews in open-source projects: Which problems do they fix? In: 11th Working Conference on Mining Software Repositories, MSR 2014 - Proceedings. Association for Computing Machinery, pp. 202–211. http://dx.doi.org/10.1145/2597073.2597082.

Bosu, A., Greiler, M., Bird, C., 2015. Characteristics of useful code reviews: An empirical study at microsoft. In: IEEE International Working Conference on Mining Software Repositories. Vol. 2015-August, IEEE Computer Society, pp. 146–156. http://dx.doi.org/10.1109/MSR.2015.21.

Broy, M., Deissenboeck, F., Pizka, M., 2006. Demystifying maintainability. In: Proceedings of the 2006 International Workshop on Software Quality. WoSQ '06, ACM Press, New York, New York, USA, pp. 21–26. http://dx.doi.org/10.1145/1137702.

Caballero-Espinosa, E., Carver, J.C., Stowers, K., 2023. Community smells—The sources of social debt: A systematic literature review. Inf. Softw. Technol. 153, 107078. http://dx.doi.org/10.1016/J.INFSOF.2022.107078.

Campbell, G.A., Papapetrou, P.P., 2013. Sonar in Action, first ed. Manning Publications Co.

Cesium Contribution Guideline, 2022. Cesium contribution guideline. URL: https://github.com/CesiumGS/cesium/blob/main/CONTRIBUTING.md.

Chatterjee, S., 2021. A new coefficient of correlation. J. Amer. Statist. Assoc. 116 (536), http://dx.doi.org/10.1080/01621459.2020.1758115, URL: https://www.tandfonline.com/action/journalInformation?journalCode=uasa20.

Chouchen, M., Ouni, A., Kula, R.G., Wang, D., Thongtanunam, P., Mkaouer, M.W., Matsumoto, K., 2021. Anti-patterns in modern code review: Symptoms and prevalence. In: Proceedings - 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering. SANER 2021, Institute of Electrical and Electronics Engineers Inc., pp. 531–535. http://dx.doi.org/10.1109/SANER50967.2021.00060.

Cleland-Huang, J., Gotel, O.C., Hayes, J.H., Mäder, P., Zisman, A., 2014. Software traceability: Trends and future directions. In: Future of Software Engineering, FOSE 2014 - Proceedings. Association for Computing Machinery, pp. 55–69. http://dx.doi.org/10.1145/2593882.2593891.

Codabux, Z., Williams, B., 2013. Managing technical debt: An industrial case study. In: 2013 4th International Workshop on Managing Technical Debt, MTD 2013 - Proceedings. IEEE Computer Society, pp. 8–15. http://dx.doi.org/10.1109/MTD.2013.6608672.

Conejero, J.M., Rodríguez-Echeverría, R., Hernández, J., Clemente, P.J., Ortiz-Caraballo, C., Jurado, E., Sánchez-Figueroa, F., 2018. Early evaluation of technical debt impact on maintainability. J. Syst. Softw. 142, 92–114. http://dx.doi.org/10.1016/J.JSS.2018.04.035.

Cunha, A., Conte, T., Gadelha, B., 2021. Code review is just reviewing code? A qualitative study with practitioners in industry. In: ACM International Conference Proceeding Series. Association for Computing Machinery, pp. 269–274. http://dx.doi.org/10.1145/3474624.3477063, URL: https://dl.acm.org/doi/10.1145/3474624.3477063.

Cunningham, W., 1992. The WyCash portfolio management system. ACM SIGPLAN OOPS Messenger 4 (2), 29–30. http://dx.doi.org/10.1145/157710.157715, URL: https://dl.acm.org/doi/10.1145/157710.157715.

D'Ambros, M., Bacchelli, A., Lanza, M., 2010. On the impact of design flaws on software defects. In: Proceedings - International Conference on Quality Software. pp. 23–31. http://dx.doi.org/10.1109/QSIC.2010.58.

Davila, N., Nunes, I., 2021. A systematic literature review and taxonomy of modern code review. J. Syst. Softw. 177, 110951. http://dx.doi.org/10.1016/J.JSS.2021.110951.

De Mello, R., Uchoa, A., Oliveira, R., Oizumi, W., Souza, J., Mendes, K., Oliveira, D., Fonseca, B., Garcia, A., 2019. Do research and practice of code smell identification walk together? A social representations analysis. In: International Symposium on Empirical Software Engineering and Measurement. IEEE Computer Society, http://dx.doi.org/10.1109/ESEM.2019.8870141.

Denning, P.J., 1992. Editorial: what is software quality? Commun. ACM 35 (1), 13–15. http://dx.doi.org/10.1145/129617.384272, URL: https://dl.acm.org/doi/10.1145/129617.384272.

Digkas, G., Chatzigeorgiou, A., Ampatzoglou, A., Avgeriou, P., 2022. Can clean new code reduce technical debt density? IEEE Trans. Softw. Eng. 48 (5), 1705–1721. http://dx.doi.org/10.1109/TSE.2020.3032557.

Doğan, E., Tüzün, E., 2022. Towards a taxonomy of code review smells. Inf. Softw. Technol. 142, 106737. http://dx.doi.org/10.1016/J.INFSOF.2021.106737.

Dueñas, S., Cosentino, V., Robles, G., Gonzalez-Barahona, J.M., 2018. Perceval: Software project data at your will. In: Proceedings - International Conference on Software Engineering. IEEE Computer Society, pp. 1–4. http://dx.doi.org/10.1145/3183440.3183475.

Eslint Contribution Guideline, 2022. Eslint contribution guideline. URL: https://github.com/eslint/eslint/blob/main/CONTRIBUTING.md.

Fagan, M.E., 1976. Design and code inspection to reduce errors in program development. IBM Syst. J. 15 (3), 182–211. http://dx.doi.org/10.1147/SJ.153.0182.

Falessi, D., Kazman, R., 2021. Worst smells and their worst reasons. In: Proceedings - 2021 IEEE/ACM International Conference on Technical Debt. TechDebt 2021, Institute of Electrical and Electronics Engineers Inc., pp. 45–54. http://dx.doi.org/10.1109/TECHDEBT52882.2021.00014.

Fontana, F.A., Pigazzini, I., Roveda, R., Zanoni, M., 2017. Automatic detection of instability architectural smells. In: Proceedings - 2016 IEEE International Conference on Software Maintenance and Evolution. ICSME 2016, Institute of Electrical and Electronics Engineers Inc., pp. 433–437. http://dx.doi.org/10.1109/ICSME.2016.33.

Fowler, R.L., 1987. Power and robustness in product-moment correlation. 11 (4), 419–428. http://dx.doi.org/10.1177/014662168701100407, URL: https://journals.sagepub.com/doi/10.1177/014662168701100407.

Fowler, M., 1999. Refactoring: Improving the design of existing code.

Galster, M., Weyns, D., Tofan, D., Michalik, B., Avgeriou, P., 2014. Variability in software systems-a systematic literature review. IEEE Trans. Softw. Eng. 40 (3), 282–306. http://dx.doi.org/10.1109/TSE.2013.56.

Ghost Contribution Guideline, 2022. Ghost contribution guideline. URL: https://github.com/TryGhost/Ghost/blob/main/.github/CONTRIBUTING.md.

Git Garbage Collection Documentation, 2022. Git garbage collection documentation. URL: https://git-scm.com/docs/git-gc.

Han, D.G., Ragkhitwetsagul, C., Krinke, J., Paixao, M., Rosa, G., 2020. Does code review really remove coding convention violations? In: Proceedings - 20th IEEE International Working Conference on Source Code Analysis and Manipulation. SCAM 2020, Institute of Electrical and Electronics Engineers Inc., pp. 43–53. http://dx.doi.org/10.1109/SCAM51674.2020.00010.

Han, X., Tahir, A., Liang, P., Counsell, S., Blincoe, K., Li, B., Luo, Y., 2022. Code smells detection via modern code review: a study of the OpenStack and qt communities. Empir. Softw. Eng. 27 (6), 1–42. http://dx.doi.org/10.1007/s10664-022-10178-7.

Izurieta, C., Vetrò, A., Zazworka, N., Cai, Y., Seaman, C., Shull, F., 2012. Organizing the technical debt landscape. In: 2012 3rd International Workshop on Managing Technical Debt, MTD 2012 - Proceedings. pp. 23–26. http://dx.doi.org/10.1109/MTD.2012.6225995.

Jiang, J., Lv, J., Zheng, J., Zhang, L., 2021. How developers modify pull requests in code review. IEEE Trans. Reliab. http://dx.doi.org/10.1109/TR.2021.3093159.

Jørgensen, M., 1999. Software quality measurement. Adv. Eng. Softw. 30 (12), 907–912. http://dx.doi.org/10.1016/S0965-9978(99)00015-0.

Khomh, F., Penta, M.D., Guéhéneuc, Y.G., Antoniol, G., 2012. An exploratory study of the impact of antipatterns on class change- and fault-proneness. Empir. Softw. Eng. 17 (3), 243–275. http://dx.doi.org/10.1007/S10664-011-9171-Y/FIGURES/E, URL: https://link.springer.com/article/10.1007/s10664-011-9171-y.

Kovalenko, V., Tintarev, N., Pasynkov, E., Bird, C., Bacchelli, A., 2020. Does reviewer recommendation help developers? IEEE Trans. Softw. Eng. 46 (7), 710–731. http://dx.doi.org/10.1109/TSE.2018.2868367.

Kruchten, P., Nord, R.L., Ozkaya, I., 2012. Technical debt: From metaphor to theory and practice. IEEE Softw. 29 (6), 18–21. http://dx.doi.org/10.1109/MS.2012.167.

Krueger, R.A., Casey, M.A., 2014. Focus Groups: A Practical Guide for Applied Research. SAGE Publication, Inc.

Krutauz, A., Dey, T., Rigby, P.C., Mockus, A., 2020. Do code review measures explain the incidence of post-release defects?: Case study replications and Bayesian networks. Empir. Softw. Eng. 25 (5), 3323–3356. http://dx.doi.org/10.1007/S10664-020-09837-4/FIGURES/5, URL: https://link.springer.com/article/10.1007/s10664-020-09837-4.

Lenarduzzi, V., Nikkola, V., Saarimäki, N., Taibi, D., 2021. Does code quality affect pull request acceptance? An empirical study. J. Syst. Softw. 171, 110806. http://dx.doi.org/10.1016/J.JSS.2020.110806.

Lenarduzzi, V., Saarimäki, N., Taibi, D., 2020. Some SonarQube issues have a significant but small effect on faults and changes. a large-scale empirical study. J. Syst. Softw. 170, 110750. http://dx.doi.org/10.1016/J.JSS.2020.110750.

Li, Z., Avgeriou, P., Liang, P., 2015. A systematic mapping study on technical debt and its management. J. Syst. Softw. 101, 193–220. http://dx.doi.org/10.1016/J.JSS.2014.12.027.

Li, Z., Yu, Y., Wang, T., Yin, G., Li, S.S., Wang, H., 2022. Are you still working on this? An empirical study on pull request abandonment. IEEE Trans. Softw. Eng. 48 (6), 2173–2188. http://dx.doi.org/10.1109/TSE.2021.3053403.

Lussier, S., 2004. New tricks: How open source changed the way my team works. IEEE Softw. 21 (1), 68–72. http://dx.doi.org/10.1109/MS.2004.1259222.

MacLeod, L., Greiler, M., Storey, M.A., Bird, C., Czerwonka, J., 2018. Code reviewing in the trenches: Challenges and best practices. IEEE Softw. 35 (4), 34–42. http://dx.doi.org/10.1109/MS.2017.265100500.

Mäntylä, M.V., Lassenius, C., 2009. What types of defects are really discovered in code reviews? IEEE Trans. Softw. Eng. 35 (3), 430–448. http://dx.doi.org/10.1109/TSE.2008.71.

Martini, A., Besker, T., Bosch, J., 2020. Process debt: A first exploration. In: Proceedings - Asia-Pacific Software Engineering Conference. APSEC, Vol. 2020-December, IEEE Computer Society, pp. 316–325. http://dx.doi.org/10.1109/APSEC51365.2020.00040.

Martini, A., Stray, V., Moe, N.B., 2019. Technical-, social- and process debt in large-scale agile: An exploratory case-study. Lect. Not. Bus. Inf. Process. 364, 112–119. http://dx.doi.org/10.1007/978-3-030-30126-2{_}14/TABLES/3, URL: https://link.springer.com/chapter/10.1007/978-3-030-30126-2_14.

McIntosh, S., Kamei, Y., Adams, B., Hassan, A.E., 2016. An empirical study of the impact of modern code review practices on software quality. Empir. Softw. Eng. 21 (5), 2146–2189. http://dx.doi.org/10.1007/S10664-015-9381-9/FIGURES/15, URL: https://link.springer.com/article/10.1007/s10664-015-9381-9.

Microsoft Word Transcription, 2022. Microsoft word transcription. URL: https://support.microsoft.com/en-us/office/transcribe-your-recordings-7fc2efec-245e-45f0-b053-2a97531ecf57.

Morales, R., Mcintosh, S., Khomh, F., 2015. Do code review practices impact design quality? A case study of the Qt, VTK, and ITK projects. In: 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015 - Proceedings. Institute of Electrical and Electronics Engineers Inc., pp. 171–180. http://dx.doi.org/10.1109/SANER.2015.7081827.

Munaiah, N., Kroh, S., Cabrey, C., Nagappan, M., 2017. Curating GitHub for engineered software projects. Empir. Softw. Eng. 22 (6), 3219–3253. http://dx.doi.org/10.1007/S10664-017-9512-6/FIGURES/11, URL: https://link.springer.com/article/10.1007/s10664-017-9512-6.

Nugroho, A., Visser, J., Kuipers, T., 2011. An empirical model of technical debt and interest. In: Proceedings - International Conference on Software Engineering. pp. 1–8. http://dx.doi.org/10.1145/1985362.1985364, URL: https://dl.acm.org/doi/10.1145/1985362.1985364.

Openlayers Contribution Guideline, 2022. Openlayers contribution guideline. URL: https://github.com/openlayers/openlayers/blob/main/CONTRIBUTING.md.

Paixao, M., Maia, P.H., 2019. Rebasing in code review considered harmful: A large-scale empirical investigation. In: Proceedings - 19th IEEE International Working Conference on Source Code Analysis and Manipulation. SCAM 2019, Institute of Electrical and Electronics Engineers Inc., pp. 45–55. http://dx.doi.org/10.1109/SCAM.2019.00014.

Palinkas, L.A., Horwitz, S.M., Green, C.A., Wisdom, J.P., Duan, N., Hoagwood, K., 2015. Purposeful sampling for qualitative data collection and analysis in mixed method implementation research. Admin. Policy Mental Health 42 (5), 533. http://dx.doi.org/10.1007/S10488-013-0528-Y, URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4012002/.

Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., De Lucia, A., 2014. Do they really smell bad? A study on developers' perception of bad code smells. In: Proceedings - 30th International Conference on Software Maintenance and Evolution. ICSME 2014, Institute of Electrical and Electronics Engineers Inc., pp. 101–110. http://dx.doi.org/10.1109/ICSME.2014.32.

Palomba, F., Bavota, G., Penta, M.D., Fasano, F., Oliveto, R., Lucia, A.D., 2018a. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. Empir. Softw. Eng. 23 (3), 1188–1221. http://dx.doi.org/10.1007/S10664-017-9535-Z/TABLES/11, URL: https://link.springer.com/article/10.1007/s10664-017-9535-z.

Palomba, F., Panichella, A., Zaidman, A., Oliveto, R., De Lucia, A., 2018b. The scent of a smell: An extensive comparison between textual and structural smells. IEEE Trans. Softw. Eng. 44 (10), 977–1000. http://dx.doi.org/10.1109/TSE.2017.2752171.

Panichella, S., Arnaoudova, V., Di Penta, M., Antoniol, G., 2015. Would static analysis tools help developers with code reviews? In: 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015 - Proceedings. Institute of Electrical and Electronics Engineers Inc., pp. 161–170. http://dx.doi.org/10.1109/SANER.2015.7081826.

Pascarella, L., Spadini, D., Palomba, F., Bacchelli, A., 2019. On the effect of code review on code smells. http://dx.doi.org/10.48550/arxiv.1912.10098, URL: https://arxiv.org/abs/1912.10098v1.

Pascarella, L., Spadini, D., Palomba, F., Bruntink, M., Bacchelli, A., 2018. Information needs in contemporary code review. Proc. ACM Human-Comput. Interact. 2 (CSCW), 27. http://dx.doi.org/10.1145/3274404, URL: https://dl.acm.org/doi/abs/10.1145/3274404.

Pdf.js Contribution Guideline, 2022. Pdf.js contribution guideline. URL: https://github.com/mozilla/pdf.js/wiki/Contributing.

Qamar, K.A., Sülün, E., Tüzün, E., 2022. Taxonomy of bug tracking process smells: Perceptions of practitioners and an empirical analysis. Inf. Softw. Technol. 150, 106972. http://dx.doi.org/10.1016/J.INFSOF.2022.106972.

R Stats Package, 2022. R stats package. URL: https://www.rdocumentation.org/packages/stats/versions/3.6.2/topics/cor.test.

Rahman, A., Parnin, C., Williams, L., 2019. The seven sins: Security smells in infrastructure as code scripts. In: Proceedings - International Conference on Software Engineering. Vol. 2019-May, IEEE Computer Society, pp. 164–175. http://dx.doi.org/10.1109/ICSE.2019.00033.

React Contribution Guideline, 2022. React contribution guideline. URL: https://reactjs.org/docs/how-to-contribute.html.

Rigby, P.C., Bird, C., 2013. Convergent contemporary software peer review practices. In: 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. http://dx.doi.org/10.1145/2491411.2491444.

Rigby, P., Cleary, B., Painchaud, F., Storey, M.A., German, D., 2012. Contemporary peer review in action: Lessons from open source development. IEEE Softw. 29 (6), 56–61. http://dx.doi.org/10.1109/MS.2012.24.

Rigby, P.C., German, D.M., Cowen, L., Storey, M.A., 2014. Peer review on open-source software projects. ACM Trans. Softw. Eng. Methodol. (TOSEM) 23 (4), http://dx.doi.org/10.1145/2594458, URL: https://dl.acm.org/doi/10.1145/2594458.

Rios, N., Mendonça Neto, M.G.d., Spínola, R.O., 2018. A tertiary study on technical debt: Types, management strategies, research trends, and base information for practitioners. Inf. Softw. Technol. 102, 117–145. http://dx.doi.org/10.1016/J.INFSOF.2018.05.010.

Sadowski, C., Söderberg, E., Church, L., Sipko, M., Bacchelli, A., 2018. Modern code review: A case study at google. In: Proceedings - International Conference on Software Engineering. IEEE Computer Society, pp. 181–190. http://dx.doi.org/10.1145/3183519.3183525.

Sas, D., Avgeriou, P., Uyumaz, U., 2022. On the evolution and impact of architectural smells—an industrial case study. Empir. Softw. Eng. 27 (4), 1–45. http://dx.doi.org/10.1007/S10664-022-10132-7/FIGURES/12, URL: https://link.springer.com/article/10.1007/s10664-022-10132-7.

Schober, P., Schwarte, L.A., 2018. Correlation coefficients: Appropriate use and interpretation. Anesth. Analg. 126 (5), 1763–1768. http://dx.doi.org/10.1213/ANE.0000000000002864, URL: https://journals.lww.com/anesthesia-analgesia/Fulltext/2018/05000/Correlation_Coefficients__Appropriate_Use_and.50.aspx.

Seaman, C.B., 1999. Qualitative methods in empirical studies of software engineering. IEEE Trans. Softw. Eng. 25 (4), 557–572. http://dx.doi.org/10.1109/32.799955.

Serrano, N., Calzada, S., Sarriegui, J.M., Ciordia, I., 2004. From proprietary to open source tools in information systems development. IEEE Softw. 21 (1), 56–58. http://dx.doi.org/10.1109/MS.2004.1259219.

Sharma, T., Fragkoulis, M., Spinellis, D., 2016. Does your configuration code smell? In: Proceedings of the 13th International Conference on Mining Software Repositories. ACM, New York, NY, USA, http://dx.doi.org/10.1145/2901739.

Shields Contribution Guideline, 2022. Shields contribution guideline. URL: https://github.com/badges/shields/blob/master/CONTRIBUTING.md.

Shimagaki, J., Kamei, Y., McIntosh, S., Hassan, A.E., Ubayashi, N., 2016. A study of the quality-impacting practices of modern code review at sony mobile. In: Proceedings - International Conference on Software Engineering. IEEE Computer Society, pp. 212–221. http://dx.doi.org/10.1145/2889160.2889243.

Sobrinho, E.V.D.P., De Lucia, A., Maia, M.D.A., 2021. A systematic literature review on bad smells-5 W's: Which, when, what, who, where. IEEE Trans. Softw. Eng. 47 (1), 17–66. http://dx.doi.org/10.1109/TSE.2018.2880977.

Soh, Z., Yamashita, A., Khomh, F., Guéhéneuc, Y.G., 2016. Do code smells impact the effort of different maintenance programming activities? In: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering. SANER 2016, Vol. 1, Institute of Electrical and Electronics Engineers Inc., pp. 393–402. http://dx.doi.org/10.1109/SANER.2016.103.

Soliman, M., Avgeriou, P., Li, Y., 2021. Architectural design decisions that incur technical debt — An industrial case study. Inf. Softw. Technol. 139, 106669. http://dx.doi.org/10.1016/J.INFSOF.2021.106669.

SonarQube Issue Severity Levels, 2022. SonarQube issue severity levels. URL: https://docs.sonarqube.org/9.5/user-guide/issues.

SonarQube Metric Definitions, 2022. SonarQube metric definitions. URL: https://docs.sonarqube.org/9.5/user-guide/metric-definitions.

SonarQube Rules, 2022. SonarQube rules. URL: https://docs.sonarqube.org/9.5/user-guide/rules.

SonarSource Customer List, 2022a. SonarSource customer list. URL: https://www.sonarsource.com/company/customers.

SonarSource Customer List, 2022b. SonarSource customer list. URL: https://www.sonarsource.com/.

Strapi Contribution Guideline, 2022. Strapi contribution guideline. URL: https://github.com/strapi/strapi/blob/master/CONTRIBUTING.md.

Stray, V., Moe, N.B., Mikalsen, M., Hagen, E., 2021. An empirical investigation of pull requests in partially distributed BizDevOps teams. In: 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, Institute of Electrical and Electronics Engineers Inc., pp. 110–119. http://dx.doi.org/10.1109/ICSSP-ICGSE52873.2021.00021.

Sülün, E., Tüzün, E., Doğrusöz, U., 2021. RSTrace+: Reviewer suggestion using software artifact traceability graphs. Inf. Softw. Technol. 130, 106455. http://dx.doi.org/10.1016/J.INFSOF.2020.106455.

Tamburri, D.A., Kruchten, P., Lago, P., Van Vliet, H., 2013. What is social debt in software engineering? In: 2013 6th International Workshop on Cooperative and Human Aspects of Software Engineering, CHASE 2013 - Proceedings. pp. 93–96. http://dx.doi.org/10.1109/CHASE.2013.6614739.

Tamburri, D.A., Kruchten, P., Lago, P., Vliet, H.v., 2015. Social debt in software engineering: insights from industry. J. Internet Serv. Appl. 6 (1), 1–17. http://dx.doi.org/10.1186/S13174-015-0024-6/FIGURES/11, URL: https://jisajournal.springeropen.com/articles/10.1186/s13174-015-0024-6.

Tamburri, D.A., Palomba, F., Kazman, R., 2021. Exploring community smells in open-source: An automated approach. IEEE Trans. Softw. Eng. 47 (3), 630–652. http://dx.doi.org/10.1109/TSE.2019.2901490.

Thongtanunam, P., McIntosh, S., Hassan, A.E., Iida, H., 2015. Investigating code review practices in defective files: An empirical study of the qt system. In: IEEE International Working Conference on Mining Software Repositories. Vol. 2015-August, IEEE Computer Society, pp. 168–179. http://dx.doi.org/10.1109/MSR.2015.23.

Tom, E., Aurum, A., Vidgen, R., 2013. An exploration of technical debt. J. Syst. Softw. 86 (6), 1498–1516. http://dx.doi.org/10.1016/J.JSS.2012.12.052.

Tomasdottir, K.F., Aniche, M., Van Deursen, A., 2020. The adoption of JavaScript linters in practice: A case study on ESLint. IEEE Trans. Softw. Eng. 46 (8), 863–891. http://dx.doi.org/10.1109/TSE.2018.2871058.

Tufano, M., Palomba, F., Bavota, G., Oliveto, R., Penta, M.D., De Lucia, A., Poshyvanyk, D., 2017. When and why your code starts to smell bad (and whether the smells go away). IEEE Trans. Softw. Eng. 43 (11), 1063–1088. http://dx.doi.org/10.1109/TSE.2017.2653105.

Tuna, E., Kovalenko, V., Tüzün, E., 2022. Bug tracking process smells in practice. In: ICSE-SEIP '22: Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice. Association for Computing Machinery (ACM), pp. 77–86. http://dx.doi.org/10.1145/3510457.3513080, URL: https://dl.acm.org/doi/10.1145/3510457.3513080.

Tuna, E., Seaman, C., Tüzün, E., 2023a. Replication package of the paper. URL: https://doi.org/10.6084/m9.figshare.25608012.

Tuna, E., Seaman, C., Tüzün, E., 2023b. Supplementary appendix of the paper. URL: https://doi.org/10.6084/m9.figshare.25608036.

Turzo, A.K., Bosu, A., 2024. What makes a code review useful to OpenDev developers? An empirical investigation. Empir. Softw. Eng. 29 (1), 1–38. http://dx.doi.org/10.1007/S10664-023-10411-X/TABLES/8, URL: https://link.springer.com/article/10.1007/s10664-023-10411-x.

Uchoa, A., Barbosa, C., Oizumi, W., Blenilio, P., Lima, R., Garcia, A., Bezerra, C., 2020. How does modern code review impact software design degradation? An in-depth empirical study. In: Proceedings - 2020 IEEE International Conference on Software Maintenance and Evolution. ICSME 2020, Institute of Electrical and Electronics Engineers Inc., pp. 511–522. http://dx.doi.org/10.1109/ICSME46990.2020.00055.

Vassallo, C., Proksch, S., Jancso, A., Gall, H.C., Penta, M.D., 2020. Configuration smells in continuous delivery pipelines: A linter and a six-month study on GitLab ACM reference format. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ACM, New York, NY, USA, http://dx.doi.org/10.1145/3368089.

Wasserman, A.I., Capra, E., 2007. Evaluating software engineering processes in commercial and community open source projects. In: First International Workshop on Emerging Trends in FLOSS Research and Development. FLOSS'07, http://dx.doi.org/10.1109/FLOSS.2007.6.

Webpack Contribution Guideline, 2022. Webpack contribution guideline. URL: https://github.com/webpack/webpack/blob/main/CONTRIBUTING.md.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A., 2012. Planning. Exp. Softw. Eng. 89–116. http://dx.doi.org/10.1007/978-3-642-29044-2{_}8, URL: http://link.springer.com/10.1007/978-3-642-29044-2_8.

XICOR CRAN package, 2022. XICOR CRAN package. URL: https://cran.r-project.org/web/packages/XICOR/index.html.

Yamashita, A., Counsell, S., 2013. Code smells as system-level indicators of maintainability: An empirical study. J. Syst. Softw. 86 (10), 2639–2653. http://dx.doi.org/10.1016/J.JSS.2013.05.007.

Yamashita, A., Moonen, L., 2013. Exploring the impact of inter-smell relations on software maintainability: An empirical study. In: Proceedings - International Conference on Software Engineering. pp. 682–691. http://dx.doi.org/10.1109/ICSE.2013.6606614.

Yu, Y., Wang, H., Yin, G., Wang, T., 2016. Reviewer recommendation for pull-requests in GitHub: What can we learn from code review and bug assignment? Inf. Softw. Technol. 74, 204–218. http://dx.doi.org/10.1016/J.INFSOF.2016.01.004.

Zampetti, F., Vassallo, C., Panichella, S., Canfora, G., Gall, H., Di Penta, M., 2020. An empirical characterization of bad practices in continuous integration. Empir. Softw. Eng. 25 (2), 1095–1135. http://dx.doi.org/10.1007/S10664-019-09785-8/TABLES/13, URL: https://link.springer.com/article/10.1007/s10664-019-09785-8.

Zazworka, N., Shaw, M.A., Shull, F., Seaman, C., 2011. Investigating the impact of design debt on software quality. In: Proceedings - International Conference on Software Engineering. pp. 17–23. http://dx.doi.org/10.1145/1985362.1985366.

Erdem Tuna received a M.Sc. in Computer Engineering from Bilkent University. His research interests are empirical software engineering and software analytics. Erdem holds a B.Sc. in Electrical and Electronics Engineering from Middle East Technical University. Contact him at erdem.tuna@bikent.edu.tr.

Carolyn Seaman is a Professor of Information Systems at the University of Maryland Baltimore County (UMBC). She is also the Director of the Center for Women in Technology, also at UMBC. Her research consists mainly of empirical studies of software engineering, with particular emphases on maintenance, organizational structure, communication, measurement, and technical debt. She also investigates qualitative research methods in software engineering, as well as computing pedagogy. She holds a PhD in Computer Science from the University of Maryland, College Park, a MS from Georgia Tech, and a BA from the College of Wooster (Ohio). More can be found at https://userpages.umbc.edu/~cseaman/.

Eray Tüzün is an Assistant Professor leading the Software Engineering and Data Analytics Research Group in the Department of Computer Engineering at Bilkent University. His research interests include software analytics, empirical software engineering, and software development productivity. He received his Bachelor's and Master's degrees in Computer Science and holds a PhD in Information Systems. He is the representative of Bilkent University in the International Software Engineering Research Network and a Senior Member of IEEE. Additional information is available at www.eraytuzun.com.