Information and Software Technology 142 (2022) 106737

Contents lists available at ScienceDirect

Information and Software Technology

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

Towards a taxonomy of code review smells

Emre Doğan [*], Eray Tüzün

Bilkent University, Department of Computer Engineering, Ankara, Turkey

ARTICLE INFO

Keywords:
Modern code review
Bad practices
Conformance checking
Code review smell
Process smell
Process debt

ABSTRACT

Context: Code review is a crucial step of the software development life cycle in order to detect possible problems in source code before merging the changeset to the codebase. Although there is no consensus on a formally defined life cycle of the code review process, many companies and open source software (OSS) communities converge on common rules and best practices. In spite of minor differences in different platforms, the primary purpose of all these rules and practices leads to a faster and more effective code review process. Non-conformance of developers to this process does not only reduce the advantages of the code review but can also introduce waste in later stages of the software development.
Objectives: The aim of this study is to provide an empirical understanding of the bad practices followed in the code review process, that are *code review (CR) smells*.
Methods: We first conduct a multivocal literature review in order to gather code review bad practices discussed in white and gray literature. Then, we conduct a targeted survey with 32 experienced software practitioners and perform follow-up interviews in order to get their expert opinion. Based on this process, a taxonomy of code review smells is introduced. To quantitatively demonstrate the existence of these smells, we analyze 226,292 code reviews collected from eight OSS projects.
Results: We observe that a considerable number of code review smells exist in all projects with varying degrees of ratios. The empirical results illustrate that 72.2% of the code reviews among eight projects are affected by at least one code review smell.
Conclusion: The empirical analysis shows that the OSS projects are substantially affected by the code review smells. The provided taxonomy could provide a foundation for best practices and tool support to detect and avoid code review smells in practice.

1. Introduction

Code review has been a widely accepted and applied best practice in software development for more than 40 years. The initial expectations from the code review process were only to find defects in code as early as possible and to increase software quality [1]. Over the years, it has been established that when properly applied, code review has some other benefits such as increasing the knowledge transferred within the development team, building team assessment and increasing the shared code ownership [2].

The first known systematic code review process was proposed by Michael Fagan in 1976 [1]. Fagan introduced the term *code inspection* to denote the meetings that developers come together and find defects in the source code before it is merged to the project codebase. Despite the success of these meetings in earlier days, the immense increase in the size of development teams and the rising popularity of distributed software development have raised the necessity of a more lightweight and flexible code review process, also known as *modern code review*.

Prior work investigates the code review process and its impacts on the code quality. McIntosh et al. [3] analyze the effect of code review coverage and participation on the software quality by mining code review histories of three open source projects. They find that commits with low review participation are more likely to have post-release defects. Thompson and Wagner [4] perform a similar study on a large dataset consisting of review histories of 3126 GitHub projects. They aim to observe the effect of code review coverage and participation on the software quality and security in terms of issues and security bugs related to the previously reviewed pull requests. Their results reveal that a high coverage and participation rate in the code review process reduces the number of future issues and severity bugs related to the commits reviewed. According to a recent report [5], 55% of developers are not satisfied with their current code review process.

Deviating from best practices or providing sub-optimal solutions is known as debt, a popular concept in software development that reflects the implied cost of additional rework caused by choosing an easy

* Corresponding author.
E-mail address: emredogan7@outlook.com (E. Doğan).

https://doi.org/10.1016/j.infsof.2021.106737
Received 1 January 2021; Received in revised form 13 September 2021; Accepted 24 September 2021
Available online 10 October 2021
0950-5849/© 2021 Published by Elsevier B.V.

solution now instead of using a better approach that would take longer. Providing sub-optimal solutions for short-term benefits usually brings waste and thus should be avoided for long-term benefits. According to Li et al. there are many subcategories of technical debt such as architecture debt, code debt, test debt, etc. [6]. There are also non-technical debt categories such as Social Debt [7] and Process Debt [8,9]. Martini et al. [10] defined the term *process debt* as *"a sub-optimal activity or process that might have short-term benefits, but generates a negative impact in the medium-long term"*. In this study, we focused on the process debt and more particularly in the code review context. Process debt in the short run, may seem faster (not conducting a code review for a commit may speed up the merge initially), however it would be potentially harmful in the long run (lack of a code review might trigger rework).

To denote the bad practices in the code review process that would lead to the process debt, we use the term *code review smell*. Previous work has referred to some of the problems in the code review process from different aspects such as lack of participation and review coverage [3,4]. Recently, Choucken et al. [11] listed five code review anti-patterns and manually investigated them on 100 code review instances. To define a catalog of code review smells and investigate the existence of smells in practice, in this study we systematically categorize bad practices in the code review process. Within our research, we define the following research questions:

RQ1- What are the bad practices followed by developers during the code review process?

In order to answer this research question, the following steps are followed:

1. We conduct a multivocal literature review (MLR) [12] to collect an initial set of bad practices in the code review process *(code review smells)*. After analyzing the white literature (20 studies published in conference proceedings and journals) and the gray literature (19 sources), an initial set of code review smells is formed.
2. In order to validate the initial set of smells and gather the feedback of practitioners on these code review smells, we conduct a comprehensive survey among 32 software practitioners having a wide experience in software development and code review. We perform follow-up interviews to further discuss the smell definitions.
3. The survey and interview results lead us to define seven bad practices in the code review process.

After compiling a list of seven code review smells with respect to the MLR, practitioner survey and the interview results, we also searched for quantitative evidence for the defined smells. This leads us to define the following research question:

RQ2- How frequently does each code review smell occur in practice?

To answer this research question, each code review smell is empirically investigated by mining code review histories of eight OSS projects (QT, Eclipse, Wireshark, LibreOffice, GitHub Desktop, Visual Studio Code, Tensorflow and Django) including 226,292 code review instances.

The rest of the paper is organized as follows. In the following section, we present the background information. In Section 3, the research methodology followed in this study is described. Section 4 illustrates each code review smell within the taxonomy. Section 5 gives the details of the empirical evaluation on eight OSS projects. In Section 6, the empirical results are discussed. Section 7 addresses validity threats of this study and finally, Section 8 presents our conclusion and future work.

2. Background & related work

2.1. Background

Code review is the examination of source code by developers other than the author in order to maintain software quality. It has been a widely approved and applied best practice in the software development for a long time [13]. However, the mindset of code review has changed significantly due to the transformation of software development methodologies.

The first known code reviews were based on the formal inspection methodology defined by Michael Fagan [14]. This formal and strictly structured review methodology was based on inspecting the source code in face-to-face meetings. Although inspection meetings in those days were very helpful to detect possible software bugs as early as possible, the lack of adaptation of this approach to fast-paced Agile methodologies [15] and cost ineffectiveness in terms of time and organizational resources [16] have led practitioners to come up with a more lightweight and tool-based code review methodology, known as *modern code review*.

Despite some minor changes in different organizations, a generic code review process consists of the following steps (As illustrated in Fig. 1) :

1. A developer is assigned as the author for the implementation of a development task (either fixing a bug or implementing a new feature).
2. When the developer completes the assigned task, they create a changeset/ pull request from their commits and start to wait for a developer to review their changeset.
3. At this stage, one or more proper code reviewers should be assigned for the pull request. This assignment can be done by a bot, a team leader, or even the authors themselves.
4. Every reviewer assignment does not necessarily end up with a completed code review. Sometimes, the assigned reviewer might reject the review request due to availability reasons. At this point, the assigner has to reassign another developer for the pull request until a reviewer accepts the review request. The same procedure is followed for each reviewer if there exists a team/company policy on the multiple numbers of reviewers for each pull request.
5. As the code review process starts, the reviewer gives feedback to the author and requests some code changes if necessary. The author updates their pull request by applying the changes requested by the reviewer. This loop continues until the reviewer is satisfied with the pull request.
6. When all code reviewers are satisfied, the pull request becomes ready to be merged to the project codebase. However, the person responsible for the merging operation might vary in different development teams.

2.2. Related work

2.2.1. Process mining in software engineering

Process mining, a combination of data science and business process, is a concept first introduced by Wil van der Aalst [17]. Due to the recent improvements in big data applications, process mining has become a promising subfield of business intelligence and has been applied in different domains. It investigates the life cycle of business processes from different aspects by mining event logs [18]. One important type of process mining is business process conformance, checking whether there exists a mismatch between a formally defined process model and real-life event logs progressing this model [18]. In recent years, many different conformance checking studies have been proposed in various industries such as healthcare [19] and manufacturing [20].

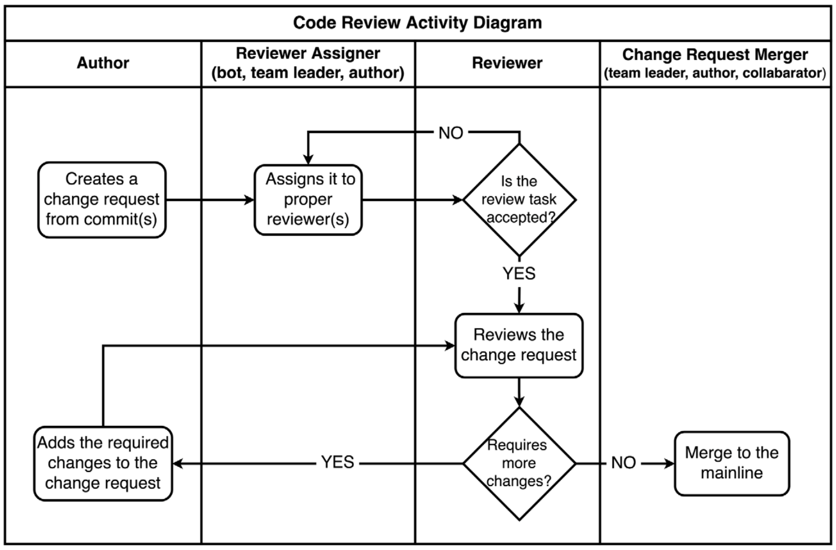

Code Review Activity Diagram
Author
Reviewer Assigner (bot, team leader, author)
Reviewer
Change Request Merger (team leader, author, collabarator)
Creates a change request from commit(s)
Assigns it to proper reviewer(s)
Is the review task accepted?
NO
YES
Reviews the change request
Requires more changes?
YES
Adds the required changes to the change request
NO
Merge to the mainline

Fig. 1. Activity diagram for the code review process.

Throughout the years, the software engineering community has developed and introduced many software development life cycles, models and processes [21]. The main objective of following such processes is to ensure the development of software within the limited resources and limitations (time and budget) [21]. Due to the considerable amount of process logs collected from real-life software projects, it has recently become possible to mine these processes in order to find cases conflicting with the ideal process definition.

Lemos et al. [22] investigate the conformance of software development processes from a software company with more than 2000 projects. The results illustrate that the formal software development process defined by the company is violated in different stages. For instance, 25.2% of the investigated processes skip the whole planning stage and initialize the process with the development. Zazworka et al. [21] introduce a tool-based approach to detect the process non-conformances and their future impacts. Poncin et al. [23] and Rubin et al. [24] propose their own software process mining frameworks based on ProM [25], a generic process mining tool used in different domains.

Process mining is also a useful and powerful tool to observe software artifact life cycles. Sunindyo et al. [26] compare the designed and actual processes to support OSS project managers in improving the process flows. Gupta [27,28] proposes a framework called Nirikshan in order to observe inconsistencies between the runtime process model (real-life model) and the design time model (ideal model) within the bug life cycle of an OSS project. In another study, the bug life cycle of the project Chromium is elaborated by mining issue tracking, peer code review and version control systems. Furthermore, some deviations from the ideal process and bottlenecks within the life cycle are investigated [29].

In the following subsection, we will specifically focus on the studies that investigates the code review process.

2.2.2. Code review process

In the literature, there are many studies investigating the code review process and the human factors affecting this flow. Baum et al. [30] investigates the process variations of the code review process among different software companies by conducting interviews among software developers from 19 companies. Their results indicate that the code review activity is not followed in a regular manner for some companies *(irregular, non-systematic review)*. By interviewing developers, they also conclude that there are some factors directly shaping the code review process within a company such as *company culture, tools used, the complexity of the software developed* etc. Fatima et al. [31] proposes a list of the wastes introduced in the code review process and maps them to the existing wastes in the software engineering domain. Alami et al. [32] interviews OSS contributors from four communities in order to explore the reasons why the code review process is helpful in open source projects. Egelman et al. [33] investigates the negative experiences within the code review process and calls them as *pushbacks*. Their results reveal that pushbacks are rare in the software projects but might have serious consequences. Caulo et al. [34] conducted an empirical study to quantitatively illustrate the effect of the code review activity to the knowledge transfer between the developers. Bird et al. [35] reports their experiences on building a code review analytics tool, *CodeFlow Analytics*. Beyond the design and building steps of this tool, they provide a set of suggestions for the future researchers to implement some data intensive software analytics tools.

The closest study to ours is conducted by Chouchen et al. [11]. They conducted a preliminary study to come up with a list of anti-patterns followed in the code review process. Their catalog consists of five anti-patterns previously mentioned in the literature. Then, by manually inspecting 100 code review instances from a single OSS project, *OpenStack*, they examine the occurrence ratios of these anti-patterns. Although their study lists an initial set of *code review bad practices*, we create our taxonomy in a more systematic and comprehensive manner by conducting: (1) white & gray literature reviews, (2) survey and interviews with developers. We also provided an empirical analysis on eight OSS projects from two different CR platforms with 226,292 code review instances.

3. Research methodology

In this study, we follow a *mixed-methods* based approach. The main idea behind this research methodology is to support empirical quantitative results with qualitative analysis [36].

The overview of the research methodology followed in this study is given in Fig. 2. First, a multivocal literature review is conducted to mine an initial set of code review bad practices from both academic and industry perspectives. Then, 32 experienced developers actively conducting code reviews are surveyed in order to get their expert opinion on code review smells. Their feedback leads us to calibrate the final set of code review smells. We also ask for their opinion on how these smells can be detected. After getting a final list of code review smells, an empirical investigation on the code review repositories of eight OSS projects is performed to support our qualitative results *(MLR, developer survey, and semi-structured interviews)* with quantitative analysis. In the following subsections, we detail the steps of our research methodology.

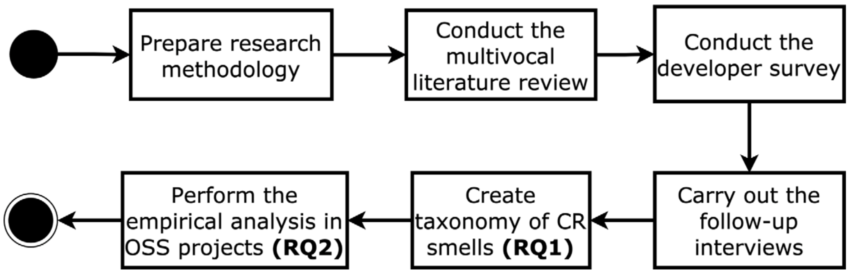

Prepare research methodology
Conduct the multivocal literature review
Conduct the developer survey
Carry out the follow-up interviews
Create taxonomy of CR smells (RQ1)
Perform the empirical analysis in OSS projects (RQ2)

Fig. 2. Research methodology followed in this study.

3.1. White & gray literature search

To establish a foundation on the bad practices followed in the code review process, an MLR is conducted. We will be describing the components of MLR (the white literature search and the gray literature search) and the elicitation of the code review smell types from the MLR in the following.

3.1.1. White literature

To scan the white literature, the guidelines of Kitchenham [37] are followed. According to Kitchenham, a typical systematic literature review (SLR) consists of 13 steps that can be grouped into three categories: planning, conducting and reporting the review. The following subsections include the details of these steps briefly.

- **Planning the Review:** The main purpose of this study is to illustrate the bad practices followed during the code review process and gather them within a taxonomy. In order not to miss any previous studies referring to any of these smells, the need for scanning the white literature arises.
  After clarifying the motivation of this SLR, a review protocol has been developed and discussed by two authors. Before conducting the search process, a generic query is defined in the following way:
  "code review" AND ("bad practice" OR "smell*" OR "challenge*" OR "anti-pattern*")*
- **Conducting the Review:** A detailed description of the search and selection process of primary studies is given in Fig. 3. Once the search query was decided, the most popular digital libraries *IEEE Explore*, *ACM Digital Library* and *Springer* were used to conduct the search. In IEEE Explore and ACM Digital Library, the abstracts of the studies were searched. Springer on the other hand, only supported the title search and the whole text search. Since the whole text search resulted in a very large number of studies, the query search was conducted in the study titles for Springer. Also, due to the limitations with the wildcard character (*) usage in different digital libraries, we made minor modifications on the individual search queries.
  After the search processes in digital libraries were completed, the references (backward snowballing) and the citations (forward snowballing) of the resulting papers were manually checked in order to include any studies missing from our search. In the snowballing process, Google Scholar[1] was used. The main inclusion criterion for our case was the relevancy of the study to our topic, *i.e. non-ideal practices followed in the code review process*. We excluded the studies written in another language than English. Also, the search process resulted with many studies investigating the code review process from different aspects. Since the scope of our study is *the bad practices followed during the code review process*, we eliminated the ones not mentioning at least one bad practice. For the studies that have both conference and journal versions, the journal versions were included in our study. To ensure these criteria, each study was investigated and summarized. The references (*backward snowballing*) and the citations (*forward snowballing*) of each study are checked manually in order not to miss any related studies. As a result of this step, a list of 20 primary studies published until the end of 2020 is created. For all the primary studies, we extracted the code review bad practice related parts (for further analysis that will be detailed in Section 3.1.3) to a spreadsheet per each different type of smell.
- **Reporting the Review:** Since we combined the results of white literature search and gray literature search, this step is described in 3.1.3.

Table 1
Code review smells appeared in white & gray literature.

Smell name	White literature	Gray literature
Lack of review	[3,11,38–41]	[42–46]
Review buddies	[47–49]	[42,46,50]
Ping-pong	[2,32,51,52]	[50,53,54]
Looks good to me (LGTM) reviews	[3,11,32,38,47,55]	[42,53,56]
Sleeping review	[2,52,57]	[46,53,56,58,59]
Missing context	[2,51,60,61]	[42,46,56,59,62–67]
Large changesets	[2,51,55,68,69]	[46,53,56,63,70,71]

3.1.2. Gray literature

The decision to whether include gray literature review within our study is made with respect to the criteria defined in the guidelines of Garousi et al. [12]. The goal of our study is to verify the scientific outcomes with practical experiences. Therefore, a combination of evidence for code review smells from both industrial and academic communities is essential. Given all this, we conduct a gray literature review by following the steps defined by Garousi et al. [12]: (1) Search process, (2) Source selection and (3) Quality assessment of sources.

We run a Google search for the term "code review". Each of the resulting 42 pages (411 results) is checked respectively. Then some modified versions of the generic query used in the white literature review are searched on Google (*e.g. "code review bad practices"*). The following inclusion/exclusion criteria is used in our search:

- Does the source discuss practices about code review process?
- Is the source in English, and is it fully accessible?
- Does the source contain author information?
- Does the source contain non-duplicate information?

Similar to the white literature, a snowballing technique is performed on the resulting sources. Two authors individually applied the criteria for all the sources. In the case of a conflicting assessment, the conflicts were resolved via an additional discussion session among the authors.

For the quality assessment of the resulting sources, the checklist proposed by Garousi et al. [12] is followed. Finally, 19 sources indicating at least one bad practice/smell in the code review process are collected.

[1] https://scholar.google.com/.

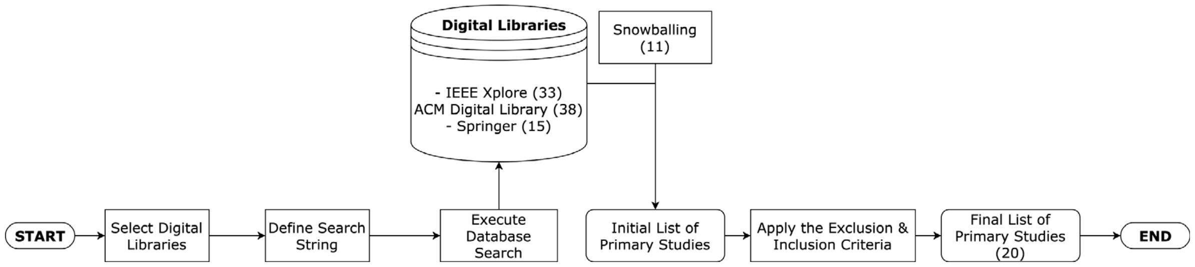


Fig. 3. The workflow of the search process (The numbers within the parentheses correspond to the number of studies.).

3.1.3. Elicitation of the code review smell types

After we compiled a list of 39 primary studies identified in the previous phase (20 white, 19 gray literature), two authors independently analyzed these studies. The authors used a shared spreadsheet to encode the specific smell/anti-pattern instance using a short sentence. Upon adding the short sentence, each author could either select one of the previously defined short descriptions from a drop-down menu, or add a new one. If no description suited the specific case, the author added a new short description in the list of possible ones, making it available for the upcoming annotations. After we achieved the initial set of categories, we merged the related process smells (for instance we merged the self-review with no-reviews). After a few iterations, and discussions among both of the authors, we ended up with the final list of code review smell categories.

As a result of the white & gray literature reviews, a set of seven code review smells is achieved. Sources from white & gray literature related to each code review smell are illustrated in Table 1. The literature sources with their corresponding smells are given in Appendix.

3.2. Developer survey & follow-up interviews

After completing the MLR, a comprehensive developer survey was prepared for software practitioners actively conducting code reviews. The questions of our survey are available online in our replication package.[2] The main objectives of this survey were:

1. To observe whether the definitions of code review smells resulted in the MLR are agreeable to expert practitioners.
2. To find any other code review smells that we missed during our literature reviews.
3. To ask the practitioners' opinion about the detection mechanism of the code review smells (inquiring about thresholds for calling an instance as a code review smell).
4. To observe the perception of practitioners on the code review smells.

Instead of relying on a public survey, we performed convenience sampling to select the participants. Since our survey was targeted for the experienced developers and took a long time to fill out (average of 40 min), we contacted 47 experienced developers selected from our personal & professional network. 32 developers out of 47 filled out the survey and three of them volunteered to perform a follow-up interview. The majority of respondents (19 out of 32) were working for multinational software companies that were considered to be top software companies in the world (e.g., Google, Amazon, Microsoft, Facebook, Udemy and Atlassian). At any given time, the respondents were told to ask clarifying questions to the authors.

The respondents have an average programming experience of 15.7 years and code review experience of 11.7 years. In the survey, each smell was explained briefly with a real-life example. Then, various questions related to respondents' familiarity with each smell were asked. We also asked for their opinion on some configurable thresholds to be used in our empirical analysis. The demographic information about the survey respondents is given in Table 2.

Then, three respondents were interviewed to learn about their perception on each bad practice. Each interview took about an hour and was recorded for further analysis. By using the answers to the open-ended survey questions and the interview transcriptions, we conducted a thematic analysis, *a systematic methodology to extract the recurring themes from a set of documents*, in order to find developers' opinions on the possible root causes and potential side effects of each smell. Within our analysis, we followed the guidelines of Cruzes and Dybå [72]. First, all of the survey/interview materials and the documents from literature reviews were examined. Secondly, initial codes for each document were extracted and reported. Then, similar codes were merged in order to group and label the themes. All of these steps were followed separately by two authors. When there was a disagreement on the codes or themes, the problems were discussed until a consensus was reached.

By combining the results of literature reviews and the answers of survey participants, we prepared a traceability table in order to show evidence for each smell's *definition*, *possible root causes* and *potential side effects* from these resources. This traceability table is available online in our replication package[2].

3.3. Empirical analysis

After the survey and follow-up interviews, a finalized taxonomy of seven code review smells is generated. Then, each smell is evaluated on eight OSS projects using Gerrit or GitHub as their code review tool. While selecting these projects, we performed purposive sampling where projects are selected based on specific characteristics. As selection criteria, we considered the following factors:

- **Development History:** Since the beginning phases of software projects might tend to have more bad practices than usual, we tried to choose projects with at least four years of development histories.
- **Number of Code Reviews:** Since we wanted to mine and analyze code review process in our quantitative analysis, we selected software projects with an adequate number of code review instances. All of the projects used in our study have more than 3000 code review instances.
- **Different Platforms:** In order to show that the introduced CR smells are not related to the code review platform and exist regardless of the platform, we chose four projects from both Gerrit and GitHub.
- **Company & Community OSS Projects:** Although all of the projects used in our study are open source (meaning that their code review histories are publicly accessible), some of them are company driven projects (Desktop project by GitHub, VSCode by Microsoft, QT projects by the QT Company) whereas others are community driven projects (Eclipse, Django, TensorFlow). By analyzing both types of software projects in our study, we aimed to observe that the CR smells exist regardless of the fact that the project is community or company driven.

The further details of the study setup are expanded in Section 5.

[2] https://doi.org/10.6084/m9.figshare.12890864 .

Table 2
Demographic information of the survey respondents.

Company type	Company ID	Num. of employees	Num. of survey respondents	Interval of CR experience (In years)	Avg. CR experience (In years)
Multinational software company	#1	100,000+	5	10-19	14.6
	#2	100,000+	2	6-7	6.5
	#3	100,000+	2	20-23	21.5
	#4	50,000+	5	6-20	15.2
	#5	1000+	3	4-7	6
	#6	1000+	2	10-10	10
Midsize software company	#7	1000+	6	4-15	12.5
	#8	1000+	1	6-6	6
	#9	1000+	1	4-4	4
	#10	200+	2	3-14	8.5
Small-sized software company	#11	10-50	1	14-14	14
	#12	10-50	1	12-12	12
Consultancy	#13	N/A	1	15-15	15
		Total respondents:	32	3-23	11.7

4. Taxonomy of code review smells

The literature review leads us to seven code review smells focusing on the bad practices in the code review process from different aspects. Table 3 illustrates the definition, possible root causes and potential side effects of each code review smell. As it can be seen from the table, some of the possible root causes or potential side effects could be shared among different code review smells. For example “availability reasons” could potentially lead to “Lack of CR” or “LGTM review” process smells.

In the survey and follow-up interviews, we ask practitioners about the importance of each smell in real life. The survey results show that the practitioners mostly agree with the proposed smell definitions. The perception of survey respondents on each code review smell resulting from the MLR is illustrated in Table 4. Also, in order to illustrate the diverging opinions of survey participants, the distributions of answers to the Likert scaled questions are given in Figs. 4 and 5.

In the following subsection, the process of synthesizing the literature reviews and the developer survey is presented. Then, the resulting taxonomy of code review smells is given in Section 4.2.

4.1. Synthesizing literature reviews & developer survey

According to the survey results, the majority of the respondents agree with our set of code review bad practices. Relatively lower agreements related to the *Reviewer-Author Ping-pong* and *Sleeping Review* smells were due to the threshold that we initially picked (i.e. in our original definition, a review taking more than 24 h would be called a sleeping review). We adjusted our thresholds (48 h for the sleeping review) according to the survey respondents. Related to the criticality of each smell, except ping-pong smell (17/32), the rest of the code review smells got a score of 3 or more by at least 26 participants indicating the importance of the code review smells. Finally, we asked our survey respondents about how often they encountered these smells. Since we deliberately picked the majority of our set of respondents from top software companies with many years of experience, they were less likely to encounter these smells. (Follow-up interviews and open-ended questions indicated that their companies already had the necessary guidelines & rules and incentive mechanisms to enforce good practices.)

In this section, the perception of developers is described by using some quotations taken from the open-ended survey questions and the MLR resources.

Lack of Code Review: The majority of the OSS projects warn the developers against self/unreviewed commits in their contribution guidelines. In the email list of the popular Kitware project, *VTK (The Visualization Toolkit)* [43], it is stated:

“...We do not allow self reviews, even for trivial commits. At some point in the future we will be taking measures to remove the ability to perform self-reviews in Gerrit, but until then we ask that all developers with elevated permissions from reviewing their own commits...”

A similar warning to developers against approving their own code change is available in the review policy of QT Community [42].

Bavota and Russo [41] found that the unreviewed commits have over two times higher chances of introducing bugs compared to the reviewed ones.

Review Buddies: The issue of selecting the same reviewer(s) without considering the suitability of them for the code changeset is discussed in both white and gray literature. QT Community warns the contributors in the following way [42]:

“Do not approve just because it would be convenient for your colleague across the room/corridor”.

The survey respondents mostly agree with this type of bad reviewer selections and their common precaution to prevent it is to assign developer groups instead of individuals. One of the survey respondents states that:

“Assigning a code review to a reviewer group instead of a specific user will align the team. When a developer gets random comments from a group member, he or she gets different points for self-improvement”.

German et al. [48] conducted an empirical study among OpenStack developers to observe the fairness during the code review process. Their results illustrate that some developers look out for their friends instead of looking out for the projects.

Reviewer-Author Ping-pong: The number of iterations within the code review process is discussed in the gray literature. A post from the Microsoft developer blog [53] explains the situation:

“If one or two comments back and forth doesn’t resolve a problem, it won’t be solved in code review. Instead, talk to the reviewer in person, on the phone, or via chat. Remember, it’s okay to agree to disagree”.

One of the survey respondents claims the same issue:

“In my company, people are encouraged to take the review offline (e.g. have a short meeting to discuss all issues) and get to a resolution quickly in such cases”.

According to the survey conducted among more than 3000 Google developers, the review processes consisting of long iterations are perceived 21 times more frustrating compared with a regular one [52].

LGTM Reviews: On this type of reviews, a survey participant shares his opinion in the following way:

“... In my company, developer promotion process considers this fact as an input. The expectation from a CR is not whether it looks good or not, it is whether they feel comfortable if they took ownership of the change, and commit the changes under their name...”

Similarly, this issue is discussed in the Microsoft developer blog [53]:

“LGTM” (a.k.a. “Looks Good To Me”) is the easiest, least time-consuming reviewer response, but it’s harmful to a codebase. If you know

Table 3
Taxonomy of code review smells.

CR smell	Definition	Possible root causes	Potential side effects
Lack of CR	Unreviewed & self reviewed changesets.	- Availability reasons - High self-confidence - Time pressure	- Missing code reviews.
Review buddies	The author assigns the same reviewer(s).	- Convenience reasons	- Ineffective code reviews - Low shared code ownership
Ping-pong	Excessively long loops between author and the reviewer.	- The reviewer cannot propose all problems all in once. - The author cannot apply all review suggestions at once.	- Increase in the review time - Blocking other developers depending on the reviewed file
LGTM reviews	The reviewer performs a lax code review and directly approves the changeset.	- Irrelevant reviewer - Lax reviewer - Availability reasons	- Missing/ineffective code reviews.
Sleeping reviews	The process takes too long in terms of time.	- The reviewer's being too busy with other tasks - Lack of notifying the reviewer about the review request	- Forgetting the changeset - Blocking other developers dependent on the commit waiting for a review
Missing context	The changeset is not properly explained and the related issue(s) are not provided.	- Time pressure	- Lack of context information about the changeset. - Decrease in the traceability of artifacts
Large changesets	The changeset is too large to be reviewed.	- Large tasks - Everything in a single changeset	- Unwilling developers for review - Ineffective code reviews

Table 4
Survey results.

	Do you agree with the smell definition? (Yes)	How critical is this smell? (Answers with a score of 3 or higher on a scale of 5)	How often do you encounter this smell? (Answers with a score of 3 or higher on a scale of 5)
Lack of review	32/32	31/32	4/32
Review buddies	31/32	28/32	21/32
Ping-pong	25/32	17/32	11/32
LGTM reviews	32/32	31/32	15/32
Sleeping reviews	28/32	26/32	21/32
Missing context	32/32	28/32	14/32
Large changesets	32/32	28/32	11/32

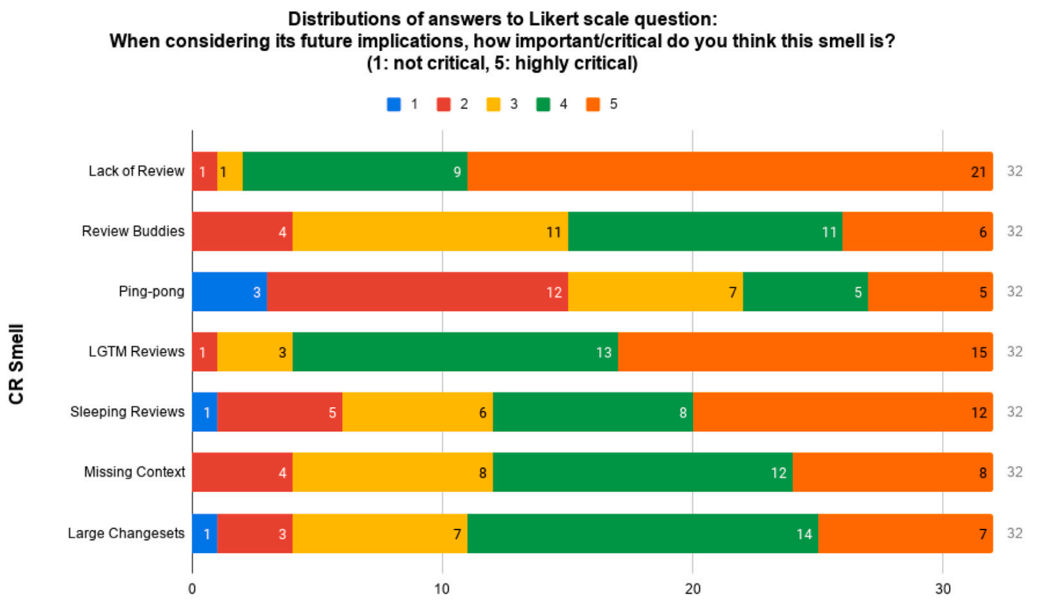


Fig. 4. Distribution of the answers to the question: *When considering its future implications, how important/critical do you think this smell is?*.

your reviewer only signed off because you applied heavy pressure ("I'm blocked by your review".), it does not help anyone.

According to the study of McInthosh et al. [3] on the code review metrics affecting the software quality, the hastily reviewed changesets tend to be more defect-prone.

Sleeping Reviews: Code review speed is a common discussion in the industry. In Google's Engineering Practices documentation, it is stated that [56]:

"If you are not in the middle of a focused task, you should do a code review shortly after it comes in. One business day is the maximum time it should take to respond to a code review request (i.e. first thing the next morning)".

One of the survey respondents explains why this practice corresponds to a smell:

"This bad practice slows down the development life cycle in different ways. Firstly, the author starts to forget the code. If there is a feedback from the reviewer after some time, the author tends to spend more time than usual since they are less familiar with the code they have written. Secondly, the

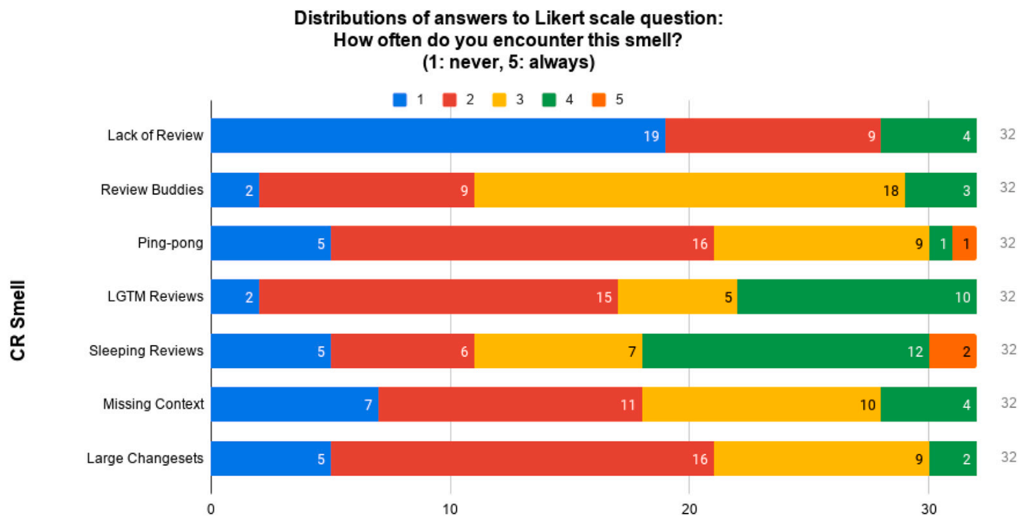


Fig. 5. Distribution of the answers to the question: *How often do you encounter this smell?*.

context switch between their current tasks and the review task is sometimes hard to handle, especially when the context is different. It also affects the author's other tasks since the author will spend more time on the review task".

MacLeod et al. [2] conducted a study to find the challenges and best practices in the code review process. In their survey conducted among 911 Microsoft developers, it was seen as the biggest challenge to receive feedback in a timely manner. One of the survey respondents explain this in the following way:
Usually you write up some code and then you send it out for review, and then about a day later you ping them to remind them... and then about half a day later you go to their office and knock on their door [2].

Missing Context in Reviews: A survey respondent clearly illustrates the importance of the changeset description:

"Changesets without description make the job of the reviewer harder. Knowing upfront what I'm reviewing helps me focus on the changes much better. If missing sometimes I ask the author for these details over email".

In a keynote, Linus Torvalds mentions the same issue [62]:

"...So commit messages to me are almost as important as the code change itself. Sometimes the code change is so obvious that no message is really required, but that is very rare. And so one of the things I hope developers are thinking about, the people who are actually writing code, is not just the code itself, but explaining why the code does something, and why some change was needed. Because that then in turn helps the managerial side of the equation, where if you can explain your code to me, I will trust the code..."

Ebert et al. [51] also shows the same issue, *unclear description of a pull request*, as one of the confusions faced during the code review process.

Large Changesets: In a blog post of Palantir [63], it is given that:

"Changes should have a narrow, well-defined, self-contained scope that they cover exhaustively. Shorter changes are preferred over longer ones. If a CR makes substantive changes to more than 5 files, or took longer than 1–2 days to write, or would take more than 20 min to review, consider splitting it into multiple self-contained CRs".

Another comment on this smell made by a survey respondent is:

"In my opinion, this is a very important smell that will improve the overall code review experience. Reviewing a changeset is really hard when the change size is large and developers tend to lose focus after a while. The larger the change the less attention it gets from reviewers, reducing the quality of review and probably the overall code quality".

In the survey of MacLeod et al. [2] conducted among Microsoft developers, the respondents also mention that they struggle with the large reviews. One of them illustrate this issue in the following way: *"It's just this big incomprehensible mess... then you can't add any value because they are just going to explain it to you and you're going to parrot back what they say"* [2].

Beyond these comments, we also analyzed the contribution guidelines of eight OSS projects selected for our empirical analysis. Table 5 illustrates the CR smells mentioned in the contribution guidelines of eight projects used in our empirical analysis. The results indicate that 75% of the projects warn the contributors against the missing context smell. The reviewer-author ping-pong smell is not considered in any of these projects whereas the LGTM, review buddies and sleeping review smells are raised in only one project. The large changeset smell is mentioned in two guidelines. For the lack of review smell, half of the project guidelines warn their developers explicitly whereas the other four guidelines imply the necessity of at least one code reviewer for each pull request.

4.2. Final taxonomy

In the following subsections, each smell is introduced with a detailed explanation, along with its possible root causes and potential side effects.

4.2.1. Lack of code review

Code review activity has multiple motivations such as code improvement, finding bugs and increasing knowledge transfer within the development team [2]. However, in order to benefit from the code review process for these motivations, this activity should be completed by a developer other than the changeset author.

If a changeset is not reviewed by a developer other than the author before it is merged (unreviewed commits), or it is reviewed by only the author themselves (self-reviewed commits), then it is a potential indicator that the code review process is not followed properly. We call this type of bad practice as *lack of code review*.

Possible Root Causes:

Availability Reasons: The author cannot find an available reviewer at that moment so that they push their changeset with a self-review or without a review at all.

High Self-Confidence: The author might think that the commit does not strictly need a code review so that they push it with a self-review or without a review at all.

Time Pressure: When the author has a strict deadline for a changeset, they might merge it with a self-review or without a review at all.

Potential Side Effects:

The lack of a proper code review will lead to defects merged into the code base undetected.

Table 5
Contribution guidelines of eight OSS projects used in our empirical analysis mentioning the code review smells. (Guidelines with asterisks (✓*) do not warn against the smells explicitly but mention the smell implicitly.)

	Lack of review	Review buddies	Ping-pong	LGTM	Sleeping	Missing context	Large changeset
QT [42]	✓	✓		✓		✓	
Eclipse [73]	✓					✓	
Wireshark [66]	✓*					✓	
LibreOffice [59]	✓*					✓	
GH Desktop [74]	✓				✓		
VS Code [75]	✓*					✓	✓
Tensorflow [76]	✓						
Django [77]	✓*					✓	✓*

4.2.2. Review buddies

Selecting a proper reviewer is an important initial step for effective code reviews [78]. Although getting a file reviewed by an expert or a senior developer seems to be an advantage, the code review activity must be balanced within the team to increase the shared code ownership [2].

The code reviewer selection might be problematic when a developer has a tendency to get their changesets reviewed by the same reviewer. We call this type of smell *review buddies*.

Possible Root Causes: The main reason behind this smell is the convenience of picking the same reviewer. When a developer does not want to deal with finding a proper reviewer, they request the same reviewer, *e.g. a close friend*, to review the changeset.

Potential Side Effects: This type of smell might cause ineffective code reviews and more importantly, decreases the shared code ownership, and causing some parts of the codebase to be known by only a very small number of developers.

4.2.3. Reviewer-author ping-pong

According to the defined code review processes in the white and gray literature, when the reviewer requests the author to make some additional changes on the code changeset, the author is supposed to update their changeset by considering the requests of the reviewer. The loop between the author and reviewer continues until the reviewer is satisfied with the changeset and approves that it is ready to be merged to the codebase.

If this loop gets excessively long, it might slow down or even block the code review process. We name this type of bad practice as *reviewer-author ping-pong*.

Possible Root Causes:

Reviewer Related Reasons: The reviewer cannot detect all of the problems in the changeset at once.

Author Related Reasons: The author cannot apply all the changes requested by the reviewer or can introduce some new bugs while fixing the previous problems.

Potential Side Effects:

A large number of iterations between the author and reviewer increases the review time. Also, it may block other developers depending on the reviewed file(s).

4.2.4. Looks good to me reviews

Even though the code review process has a variety of benefits, its main purpose is to find defects in the source code as early as possible. When reviewers find a defect or have some suggestions for the author, they are supposed to state their opinion by providing some feedback through comments. The absence of these comments defeats the purpose of getting feedback through review comments. Lack of this feedback might potentially lead to some future uncaught bugs in the source code.

Our claim is that some developers do code reviews without paying much attention and directly approve the changeset. We call this type of reviews as *looks good to me (LGTM) reviews* referring to the popular phrase used in the open source community "looks good to me (LGTM)".

Possible Root Causes:

Irrelevant Reviewer: The reviewer is unfamiliar with the changeset and has to respond to the review request due to an organizational regulation.

Availability Reasons: The reviewer might be too busy with other tasks and cannot reject the review request due to an organizational regulation.

Lax Reviewer: The requested reviewer does not pay attention to the review task and just approves the changeset.

Potential Side Effects:

In such scenarios, the author cannot get feedback from the reviewer. The lack of a proper review on changesets might lead to some future bugs.

4.2.5. Sleeping reviews

It has always been a key motivation of software development to find software defects as early as possible in order to save time, effort and money [79]. Fagan's inspection methodology aimed to put this motivation into practice by inspecting software artifacts at separated checkpoints. In some cases this might take a long time such as weeks. With the modern code review tools, it has become possible to complete a review within days, or sometimes in hours [80]. In Google, code reviews are completed in a short time, with a median of less than 4 h [81]. Whereas, in the study of Rigby and Bird [82], the median review completion times of Microsoft, AMD, and the Chrome and Android projects are found to be between 14.7 and 20.8 h.

By considering all these results from industry and OSS projects, a code review process is named a *sleeping review* if it takes an excessively long time to be completed.

Possible Root Causes:

Availability Reasons: The reviewer might be too busy with other tasks and forget the review task.

Lack of Reminder: The non-responding reviewers are not notified of the review task at regular intervals.

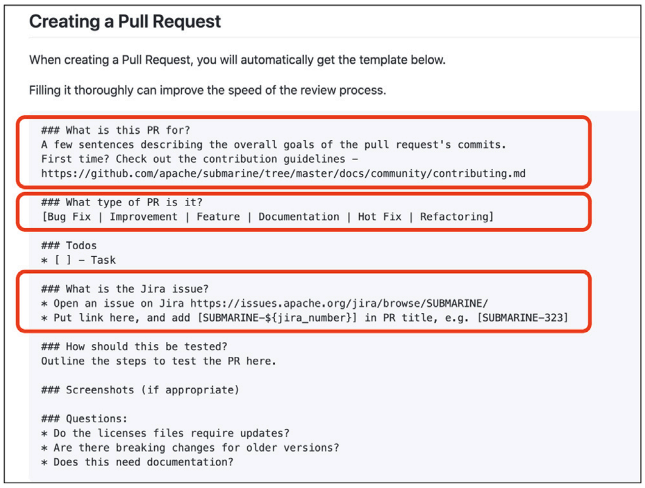
Potential Side Effects:

Merge Conflict: When another developer needs to work on the file under review and their work is dependent on the commit being reviewed, they might have to wait for a long time.

Forgetting Code: When a review task takes a long time, it becomes harder for the author to remember their commits and apply the required changes by the reviewer without introducing new defects.

4.2.6. Missing context in reviews

Traceability among different software artifacts is an essential and helpful factor to improve the software development and maintenance life cycles [83]. Code review is the inspection of a code changeset that might be created due to several reasons: bug, improvement, feature, documentation, etc. This relation between the artifacts of code review and issue tracking processes makes it necessary to link them to each other.



Creating a Pull Request

When creating a Pull Request, you will automatically get the template below.

Filling it thoroughly can improve the speed of the review process.

```
### What is this PR for?
A few sentences describing the overall goals of the pull request's commits.
First time? Check out the contribution guidelines -
https://github.com/apache/submarine/tree/master/docs/community/contributing.md

### What type of PR is it?
[Bug Fix | Improvement | Feature | Documentation | Hot Fix | Refactoring]

### Todos
* [ ] - Task

### What is the Jira issue?
* Open an issue on Jira https://issues.apache.org/jira/browse/SUBMARINE/
* Put link here, and add [SUBMARINE-${jira_number}] in PR title, e.g. [SUBMARINE-323]

### How should this be tested?
Outline the steps to test the PR here.

### Screenshots (if appropriate)

### Questions:
* Do the licenses files require updates?
* Are there breaking changes for older versions?
* Does this need documentation?
```

Fig. 6. Contribution guidelines of the project apache submarine[3].

In order to ensure this linkage between code review and issue tracking repositories, most of the OSS projects have a strict contribution policy on linking the related issue with the commit submitted for a review. The contribution guidelines for submitting a new pull request to Apache's Submarine repository is given in Fig. 6. One of the required fields in the given template is the Jira issue related to the commit under review.

Dalipaj et al. [60] also show the lack of linkage between bug and review repositories on the OpenStack project.

From a reviewer's perspective, inspecting the changeset without prior knowledge on the related issue might decrease the review quality since the issue introduces the problem that is solved by the submitted commit. Therefore, if a code review is not explicitly linked to an issue or explained adequately, it is affected by the smell: the *missing context in reviews*.

Possible Root Causes:

The main reason for this smell is the nonconformance of developers to the formal software development process. The commit author might be in a hurry and think that it is a waste of time to provide a proper explanation and the related issue(s).

Potential Side Effects:

Lack of Traceability: The absence of a proper changeset description decreases the traceability within the software project. When a bug is reopened, the bug assignee should be able to find a proper explanation in the re-visited changeset.

Lack of Information for Reviewers: The reviewer cannot get enough information about the changeset before they start to review it.

4.2.7. Large changesets

For a code review process to consist of quick and frequent iterations, the changeset must include small code changes [82]. Large changesets have negative impacts on the review process in different aspects: Rigby et al. [80] find that commits should include small and complete changesets. Bosu et al. [84] and Czerwonka et al. [57] validate this claim by illustrating that there exists a relation between the useful comments made by the reviewer and the size of the changeset.

The impact of large changesets is also discussed in the industry projects: Sadowski et al. [81] claim that one main reason for fast code reviews at Google is that 90% of code reviews include fewer than 10 changed files and the median value of changed lines of code (LOC) is 24. Similarly at Microsoft, large changesets are found to be one of the most common challenges in the code review process among developers [2].

In this context, a changeset is called large if it consists of a large number of changed LOC.

Possible Root Causes:

Large Tasks: If the task is too complicated to realize in a small changeset, then the author has to create large changesets. To fix such problems, the tasks should be generated in an atomic manner.

Everything in a Single Changeset: Some developers try to complete a whole large task in a single changeset leading to the smell: *large changesets*.

Potential Side Effects:

Unwilling Developers for Review: Since large changesets are harder to review, most of the developers avoid reviewing them.

Ineffective Reviews: The results of the gray literature review and developer survey show that developers cannot focus on the whole of large changesets. This fact leads to ineffective reviews introducing possible future bugs.

Table 6
Summary statistics for data collected from four gerrit repositories.

Project name	Total reviews	Filtered reviews	Start date	End date
QT	96,722	74,755	2017-01-01	2020-04-20
Eclipse	71,993	57,585	2017-01-01	2020-04-20
Wireshark	17,407	16,336	2017-01-01	2020-04-21
LibreOffice	58,781	54,032	2017-01-01	2020-04-20
Total	244,903	202,708		

Table 7
Summary statistics for data collected from four GitHub repositories.

Project name	Total PRs	Filtered PRs	Start date	End date
GitHub Desktop	3602	2993	2016-05-11	2020-06-05
Visual studio Code	7343	5206	2015-11-16	2020-06-06
TensorFlow	14,498	9807	2015-11-09	2020-06-06
Django	13,008	5578	2012-04-28	2020-06-06
Total	38,451	23,584		

5. Empirical analysis

This section includes the details of the empirical analysis setup and the quantitative evidence for code review smells. In Section 5.1, the details regarding the datasets are explored. Section 5.2 illustrates the preprocessing steps followed within this study. Finally, Section 5.3 presents the quantitative evidence for each code review smell among eight OSS projects.

5.1. Dataset types and analysis

In order to explore and quantify code review smells in real-life scenarios, we investigated eight popular open source projects using Gerrit and GitHub as their code review tool.

Gerrit is a lightweight, web-based modern code review tool supporting integration with Git. In Gerrit, the code changesets are represented in "patch sets". If the reviewer is satisfied with the current patch set, then the changeset is merged to the codebase. If not, the reviewer requests that the author makes some additional changes and create a new patch set.

GitHub is a popular Git repository hosting service. Beyond its main purpose as a version control system, it has many other services such as bug tracking, feature requests, task management and continuous integration/delivery. The code review tool in GitHub is integrated into

[3] https://github.com/apache/submarine/blob/master/docs/community/contributing.md.

Table 8
Counts and percentages of code review smells in four Gerrit projects.

	QT		Eclipse		Wireshark		LibreOffice		Total	
Total instances	74,755		57,585		16,336		54,032		202,708	
CR smells	Count	Perc. (%)	Count	Perc. (%)	Count	Perc. (%)	Count	Perc. (%)	Count	Perc. (%)
Lack of review	2987	4.0	28,535	49.6	7486	45.8	32,621	60.4	71,629	35.3
Ping-pong	8577	11.5	1843	3,2	685	4.2	576	1.1	11,681	5.8
Sleeping reviews	15,718	21.0	16,438	28.5	1768	10.8	10,551	19.5	44,475	21.9
Large changesets	3564	4.8	6263	10.9	1006	6.2	2508	4.6	13,341	6.6
Missing context	20,345	27.2	18,883	32.8	4789	29.3	22,188	41.1	66,205	32.7
Combined	39,709	53.1	47,831	83.1	11,461	70.2	45,124	83.5	144,125	71.1

Table 9
Counts and percentages of code review smells in four GitHub projects.

	GitHub Desktop		Visual studio code		TensorFlow		Django		Total	
Total instances	2993		5206		9807		5578		23,584	
CR smells	Count	Perc. (%)	Count	Perc. (%)	Count	Perc. (%)	Count	Perc. (%)	Count	Perc. (%)
Lack of review	440	14.7	3000	57.6	1276	13.0	3341	59.9	8057	34.2
Ping-pong	209	7.0	92	1.8	449	4.6	7	0.1	757	3.2
Sleeping reviews	1240	41.4	2089	40.1	4690	47.8	1887	33.8	9906	42.0
Large changesets	160	5.3	415	8.0	975	9.9	162	2.9	1712	7.3
Missing context	335	11.2	1277	24.5	4330	44.2	2138	38.3	8080	34.3
Combined	1868	62.4	4316	82.9	8015	81.7	4990	89.5	19,189	81.4

the pull request management service. When a developer creates commit(s), they create a pull request and send it to appropriate developers to accomplish the code review task. When the reviewer requires some additional changes, the author creates new commit(s) and adds them to the pull request.

We fetched the code review data of eight OSS projects by using Perceval [85] and made the datasets available online.[4] The summary statistics for Gerrit and GitHub projects are given in Tables 6 and 7 respectively.

The empirical analysis is performed on interactive Python notebooks and shared online[5] with instructions to replicate this study.

5.2. Data cleaning & preprocessing

After fetching the data, a manual inspection of the raw data is completed for each project. Data instances affected by at least one of the following conditions are removed from the dataset to improve the correctness of our study:

- The scope of our empirical analysis is limited to the code review instances ending up with a merge to the codebase since the majority of the smells defined in our taxonomy analyze the completed code review processes. For this reason, code review instances other than the merged ones are ignored.
- Some review tasks are performed by review-bots. Since our study investigates the nonconformance of developers to the code review process, the reviews performed by bots are removed from our dataset. To this end, all developer names are checked manually.
- Instances with missing ID information of the author or reviewers *(e.g. deleted GitHub & Gerrit accounts)* are removed.
- Some commits seem to have a changeset with no changed lines of code. When we inspect the webpages of these instances, it is observed that these commits consist of a cherry pick operation, applying a commit from one branch into another one. Since the changeset comes from another commit, Gerrit does not reflect the actual changed lines of code and shows this value as zero.

The numbers of instances in Gerrit and GitHub projects after the preprocessing step are given in Tables 6 and 7.

[4] https://doi.org/10.6084/m9.figshare.13040474.
[5] https://doi.org/10.6084/m9.figshare.12890864.

Table 10
Size labels of changesets defined by Gerrit.

Changed lines of code	Size label
[0,10)	XS
[10,50)	S
[50,200)	M
[200,1000)	L
1000+	XL

5.3. Quantitative results

According to the taxonomy detailed in Section 4, a detection method for each smell is proposed except for *LGTM Reviews*. The reason to exclude this smell is the feedback provided in our developer survey. Although the majority of the respondents agreed on the smell definition, they shared serious concerns about how accurately it can be detected. In a follow-up interview, one of the respondents noted that:

"When I read the definition of code review smell, it completely makes sense. However, I have some serious doubts on whether it can be accurately detected or not. While conducting reviews, I sometimes cannot find anything wrong (bug, typo, etc.) about the code and just approve it immediately. According to your definition, this is a smell which in fact is not".

The remaining six code review smells are evaluated in terms of the number of occurrences (smell counts) and percentages in eight OSS projects. The resulting statistics for Gerrit and GitHub projects are given in Tables 8 and 9.

In the following subsections, we first introduce the detection method of each smell. Then, the analyzed projects are compared with respect to their smell characteristics.

5.3.1. Lack of code review

To detect this smell, the following procedure is followed:

Smell Detection Method:

1. If the changeset is merged to the project codebase without a code review, then it is an unreviewed commit.
2. If the one and only reviewer of a changeset is the author of it, then it is a self-reviewed commit.
3. If a review consists of unreviewed or self-reviewed changesets, then it is affected by the smell: *lack of code review.*

Table 11
Ratios of code reviews with lack of review smell for different sizes (XS: 0–10, S: 10–50, M: 50–200, L: 200–1000, XL: 1000+).

Changeset size	QT	Eclipse	Wireshark	LibreOffice	GitHub Desktop	VS Code	TensorFlow	Django
XS	0.05	0.57	0.47	0.55	0.17	0.63	0.13	0.74
S	0.04	0.49	0.43	0.60	0.13	0.52	0.12	0.55
M	0.03	0.46	0.43	0.65	0.15	0.51	0.10	0.50
L	0.03	0.43	0.50	0.71	0.15	0.60	0.11	0.52
XL	0.04	0.45	0.52	0.71	0.10	0.76	0.27	0.60

By following these steps, eight projects are examined respectively. The empirical results are given in Tables 8 and 9.

Although Eclipse, Wireshark and LibreOffice projects show similar characteristics (45.8% to 60.4%), QT has a significantly lower smell percentage (4%). Such a major difference leads us to investigate the contribution guidelines & review policies of these four projects. [42, 59,66,73]

In the QT guidelines, developers are strictly warned against self/unreviewed changesets with the exact words:

"Do not approve your own changes". [42]

While the other three projects do not have such a warning; in the guidelines of LibreOffice, core developers are allowed to give a +2 (approval in Gerrit) to themselves:

"+2 is used by the author to signal no review is needed (this can only be done by core developers, and should be used with care)". [59]

A similar investigation is also conducted on GitHub projects. *Desktop* and *TensorFlow* projects have significantly lower smell ratios than *Visual Studio Code* and *Django* projects. Despite such differences between projects, the percentages of the lack of review smell in Gerrit and GitHub projects are close to each other (35.3% and 34.1%).

Since some of the survey respondents claim that this bad practice is related to the changeset size, we investigate the relation between the lack of review smell and the changeset size. Fig. 7 illustrates the smell counts of different sized changesets in Gerrit and GitHub projects. These size intervals (XS through XL) are defined by Gerrit itself and can be seen in Table 10. The smell ratios for these size intervals in eight projects are given in Table 11.

The results show that *QT*, *Desktop* and *TensorFlow* projects are not affected by the lack of review smell in a significant manner. The smell ratios seem to be higher in the very small and large changesets.

5.3.2. Review buddies

In order to detect this type of smell, the following steps are performed:

Smell Detection Method:

1. Self-reviewed and unreviewed commits are eliminated.
2. Commits authored by a developer having fewer than 50 contributions are ignored in order to obtain the core developers of each project. This threshold is applied in order to avoid the situation that when a developer has a small number of contributions, the reviewers assigned for these commits become the review buddies of this developer artificially. We asked this threshold value to the survey participants and further discussed it in the follow-up interviews. Although there is not a strict consensus among the participants, the majority of them find the value of 50 as reasonable.
3. All (Author, Reviewer) pairs and their corresponding numbers of occurrence are listed for each core author.
4. If there exists a reviewer who reviewed at least half of the commits submitted by an author, then this reviewer is called the *review buddy* of the author.

Since this smell is related to the developers rather than the code review processes, its results are given separately in Tables 12 and 13. More than one fourth of 492 developers in four Gerrit projects assign a specific reviewer for more than half of their commits. On the other hand, GitHub projects show significantly lower smell ratios due to the smaller number of developers with at least 50 commits.

Table 12
Review buddies in four Gerrit projects.

Project	Developers having a review buddy	Developers having more than 50 contributions	Smell percentage (%)
QT	31	170	18.2
Eclipse	82	204	40.2
Wireshark	6	29	20.7
LibreOffice	31	89	34.8
Total	150	492	30.5

Table 13
Review buddies in four GitHub projects.

Project	Developers having a review buddy	Developers having more than 50 contributions	Smell percentage (%)
GitHub Desktop	0	6	0.0
Visual studio code	1	13	7.7
TensorFlow	1	31	3.2
Django	1	9	11.1
Total	3	59	5.1

As a result, this smell is a more common practice in Gerrit projects. The dominance of review buddies may lead to ineffective code reviews and decreases the shared code ownership in these projects.

5.3.3. Reviewer-author ping-pong

The procedure to detect this smell is given in the following steps:

Smell Detection Method:

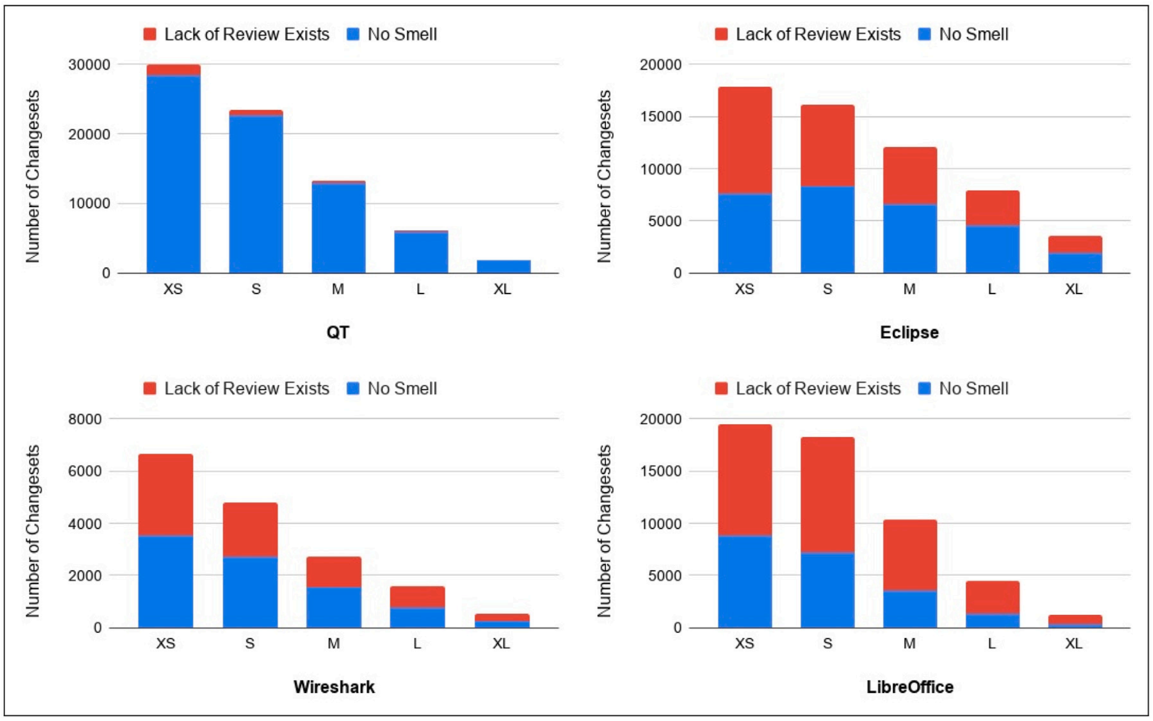
1. If a review process consists of an excessively large number of iterations between the author and reviewer, it is affected by the smell: *reviewer-author ping-pong*.
2. To decide the threshold value for *the excessively large changeset*, the survey participants were asked how many iterations there should be between the author and the reviewer at most. The majority of the respondents (23 out of 32) agreed that this loop should not exceed three iterations.
3. If a review process consists of more than three iterations between the author & reviewer, then it is affected by the smell: *reviewer-author ping-pong*.

The results in Tables 8 and 9 show that the code review instances in Gerrit projects lead longer author–reviewer iterations than the GitHub projects.

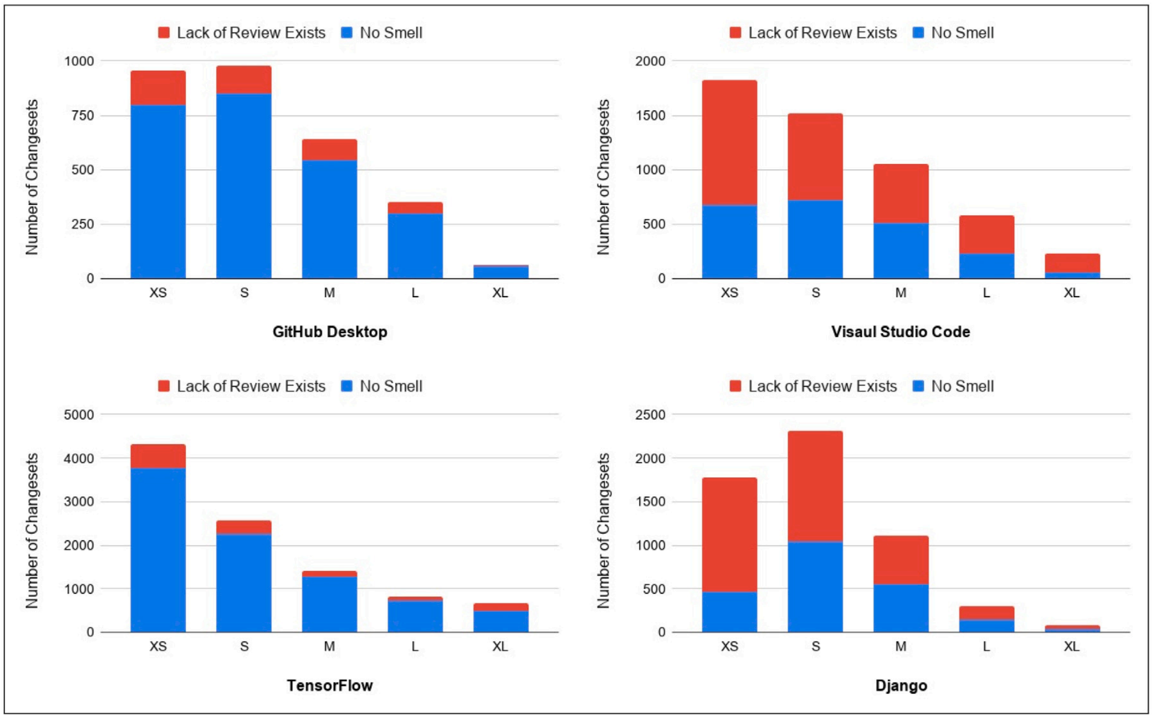
Fig. 8 shows the smell percentages in different-sized changesets. From the size label XS through L, the number of reviewer-author ping-pong cases increases as the changeset size increases However, for the XL sized changesets, the number of occurrences drop in five projects. One possible explanation for this behavior might be that it becomes harder to find the issues in the very large changesets.

5.3.4. Sleeping reviews

In order to detect this type of smell, the following steps are performed:



(a) Gerrit Projects



(b) GitHub Projects

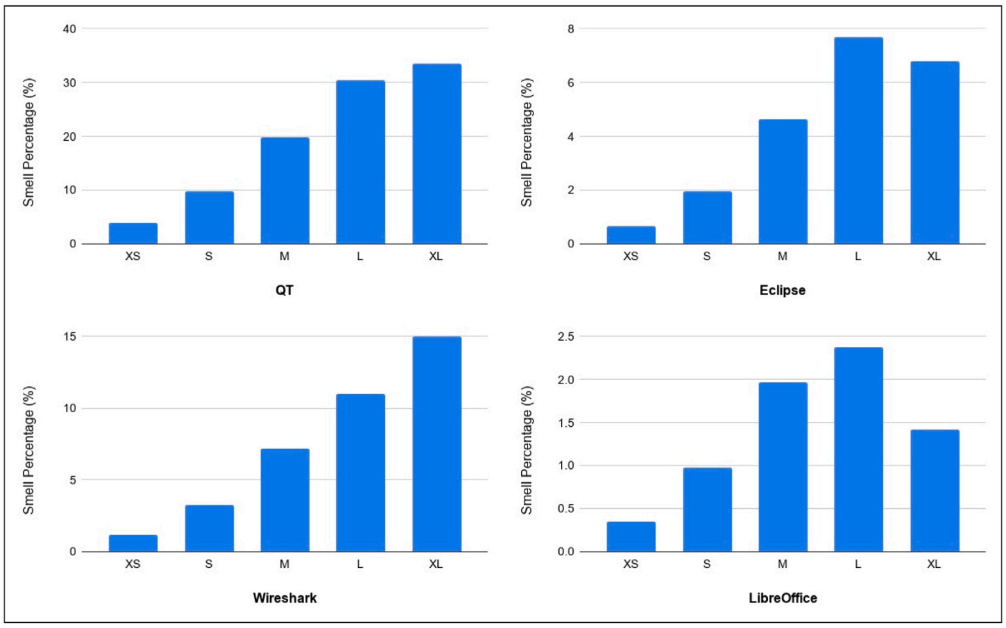
Fig. 7. Number of different-sized changesets with the smell: *Lack of Code Review* in Gerrit and GitHub projects.

Smell Detection Method:

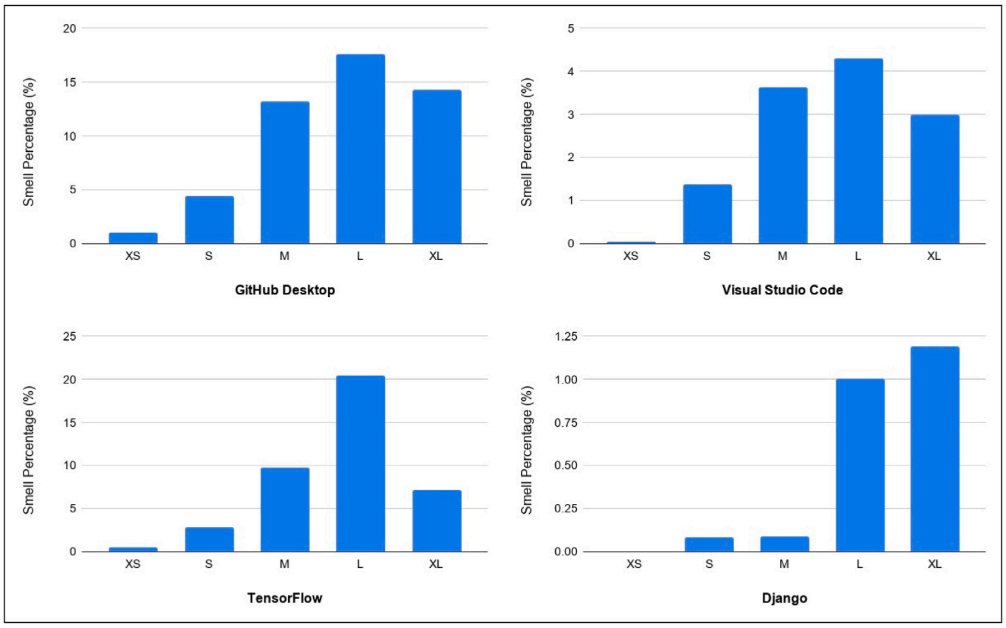
1. The elapsed time between the creation and completion moments of each code review process is calculated and named *review sleeping time (RSTime)*.

$$RSTime = t_{reviewCompleted} - t_{reviewCreated}$$

2. According to Google Engineering Practices documentation guidelines [56], a code review should not last more than 1 business day. In our survey, the participants are consulted on how long a code review process should take. The majority of the respondents (29 out of 32) claim that a code review process should not exceed two days (48 h). Based on the survey results, we set our threshold for a sleeping review as 48 h.



(a) Gerrit Projects



(b) GitHub Projects

Fig. 8. Smell percentages of different-sized changesets with the smell: *Reviewer-Author Ping-pong* in four Gerrit (a) and four GitHub (b) projects.

3. Relying on the statistics established in the white and gray literature and the survey results, *sleeping review* occurs when the code review takes more than two days.

In the detection method of sleeping reviews, there exists a minor risk of choosing the threshold value as two days without considering the weekends and holidays. These days are not considered since some of the open source projects are developed on a volunteer basis and it is not straight forward to distinguish weekends/holidays from weekdays.

The occurrence statistics of this smell are given in Tables 8 and 9. Wireshark seems to have faster code reviews whereas more than one-fifth of the reviews in other projects take longer than 48 h.

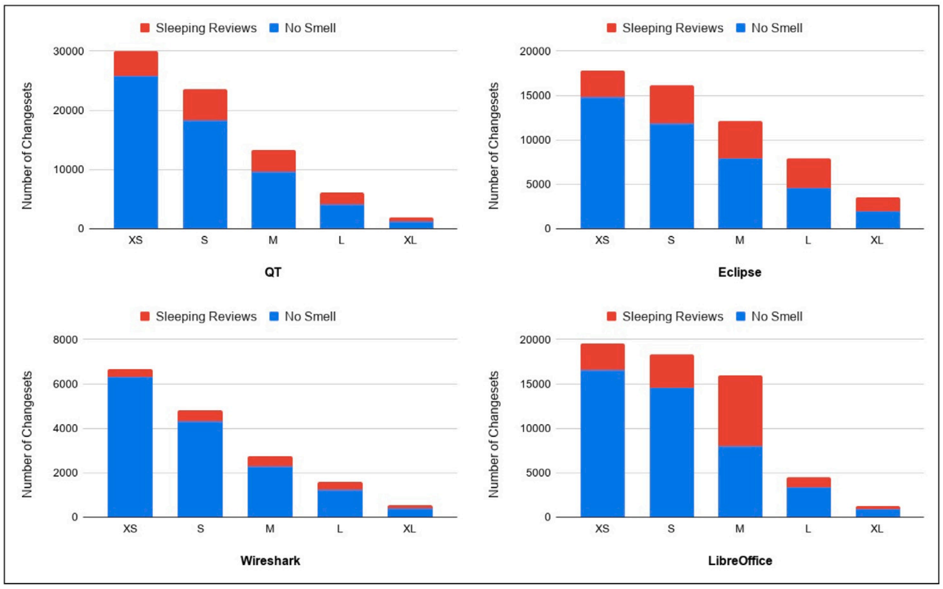
We also investigated the relation between the changeset size and sleeping review counts in each project. The histogram in Fig. 9 illustrates this relation in each project. Also, the smell ratios for the size intervals in eight projects are given in Table 14. It is expected for the reviews of large changesets to take a longer time. However, the long review processes of small changesets indicate unwanted delays in the process.

5.3.5. *Missing context in reviews*

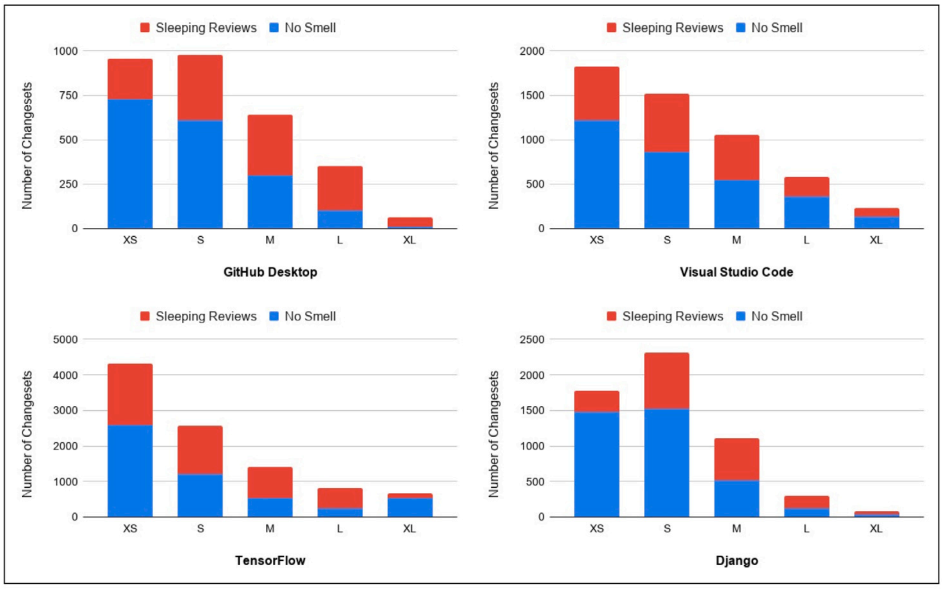
In order to detect this type of smell, the following steps are performed:

Smell Detection Method:

1. Since each OSS project has its own contribution guidelines, the commit message format might vary in different projects. The text pattern to link the related issues of reviews in each project is achieved by analyzing the related guidelines.



(a) Gerrit Projects



(b) GitHub Projects

Fig. 9. Number of different-sized changesets with the smell: *Sleeping Reviews* in four Gerrit (a) and four GitHub (b) projects.

Table 14
Ratios of code reviews with sleeping review smell for different sizes (XS: 0–10, S: 10–50, M: 50–200, L: 200–1000, XL: 1000+).

Changeset size	QT	Eclipse	Wireshark	LibreOffice	GitHub Desktop	VS Code	TensorFlow	Django
XS	0.14	0.17	0.05	0.15	0.24	0.33	0.40	0.16
S	0.22	0.27	0.10	0.20	0.37	0.43	0.53	0.34
M	0.27	0.35	0.17	0.50	0.54	0.48	0.62	0.53
L	0.33	0.42	0.21	0.25	0.71	0.38	0.70	0.58
XL	0.35	0.44	0.24	0.26	0.83	0.44	0.20	0.55

2. Heading and changeset description of each review instance are mined in order to check whether they include a related issue number/ID or a proper explanation of the changeset. If the PR author leaves the description field empty or copies the PR title and pastes it into the PR description field, then it is missing a proper description.
3. If a review process is not linked to a related issue and a proper description of the changeset is not provided, then it is affected by the smell: *missing context in reviews*.

Gerrit and GitHub allow the developers to provide their commit details in two different fields: a heading to summarize the changeset and a body section to give further details. We observe that many developers write a short description as a heading, then copy the exact same text into the body section. In this study, a changeset/PR is affected by the smell *missing context in reviews* if its body field is the same as the heading field or does not include any further description/linked issue information.

The obtained results for this smell are illustrated in Tables 8 and 9. It is shown that almost one code review process out of three is affected by the lack of a proper changeset description.

When the smell occurrence ratios in different-sized changesets are investigated in Fig. 10, it is clearly seen that this bad practice is more common in the small changesets among both Gerrit and GitHub projects. The smell ratios for the size intervals in eight projects are given in Table 15. It is observed that in general, the changesets with the size labels *XS* and *XL* tend to suffer this smell more likely.

5.3.6. *Large changesets*

In order to detect this type of smell, the following steps are performed:

Smell Detection Method:

1. The number of changed LOC is calculated by summing up the number of added and deleted LOC.
2. If a changeset consists of more than 500 changed LOC, then the code review process is affected by *large changesets* smell.

To define a threshold value for the large changesets, we asked the survey respondents for their opinion on this threshold value. The majority of the respondents (28 out of 32) agreed that a changeset should not exceed 500 changed LOC. Therefore, we decided to call a changeset as a large one if it consists of more than 500 changed LOC.

The quantitative results for the large changesets are given in Tables 8 and 9. It is illustrated that all of the projects are affected by this smell with a percentage range between 2.9 and 10.9. Despite minor differences between projects, Gerrit (6.6%) and GitHub (7.3%) platforms show similar characteristics in terms of large changesets.

After evaluating six code review smells quantitatively, the occurrence counts and percentages of the code review processes having at least one code review smell are obtained. The bottom lines of Tables 8 and 9 illustrate that 71.1% of code reviews in Gerrit and 81.4% of GitHub PRs are affected by at least one smell defined in our taxonomy.

5.4. *Statistical analysis of the results*

In order to statistically analyze the effect of the project type (*community* or *company* driven) and the changeset size to the code review smell percentages, we used the *Chi-Square Test of Independence* test.

5.4.1. *Project type vs smell percentages*

To analyze the effect of project type to the smell percentages, first, each code review instance is categorized with respect to its type (community or company). To assess the project type, each project is searched online and the project type is tagged as either community or company. According to our investigation, the projects *QT*, *GitHub Desktop*, *VS Code* and *Tensorflow* are managed or supported by a company. On the other hand, the other four projects *(Eclipse, Wireshark, LibreOffice and Django)* are driven by a community.

After tagging each code review instance, we proposed the null hypothesis and the alternative hypothesis for each applicable CR smell (lack of review, ping-pong, sleeping review, missing context and combined smell):

H_{0_a}: There is no statistically significant relationship between the **project type** and whether or not a code review is affected by the <**smell type**>.

H_{1_a}: There is a statistically significant relationship between the **project type** and whether or not a code review is affected by the <**smell type**>.

Having set the significance level, $\alpha = 0.01$ (with a level of 99% confidence), we tested each alternative hypothesis proposed for different CR smells. Considering the different smell percentages among community and company driven projects, it is not surprising that the P-values are less than the significance level for all CR smells meaning that we can reject the null hypotheses in favor of the alternative hypotheses. As it can be observed Table 16, the smell percentages except for the *ping-pong* and *sleeping review* smells are significantly higher in the community projects whereas the ping-pong smell seems to occur more frequently in the company projects.

5.4.2. *Changeset size vs smell percentages*

In order to investigate the relation between the changeset size and whether or not a code review is affected by a smell, we proposed the following null and alternative hypotheses for each smell:

H_{0_b}: There is no statistically significant relationship between the **changeset size** and whether or not a code review is affected by the <**smell type**>.

H_{1_b}: There is a statistically significant relationship between the **changeset size** and whether or not a code review is affected by the <**smell type**>.

In order to test our hypotheses, we applied the Chi-Square Test of Independence to each project separately and then to all instances. The resulting p-values are given in Table 17.

The p-values indicate that, except for the lack of review smell test in GitHub Desktop project (which is 0.13), all other p-values are smaller than the significance level $\alpha = 0.01$ favoring the alternative hypotheses.

6. Discussion

6.1. *Code review smells in different platforms (Gerrit & GitHub)*

The results of the empirical analysis show that each code review smell occurs with different ratios in eight OSS projects. In this section, the similarities and differences between Gerrit & GitHub projects are discussed.

Lack of Code Review: In GitHub, it is not allowed to apply *self-review* on a pull request. Therefore, *the lack of review* smell consists of only unreviewed pull requests in GitHub projects. Nevertheless, the total ratios of changesets with this smell are quite close to each other (*%35.3 and %34.1*).

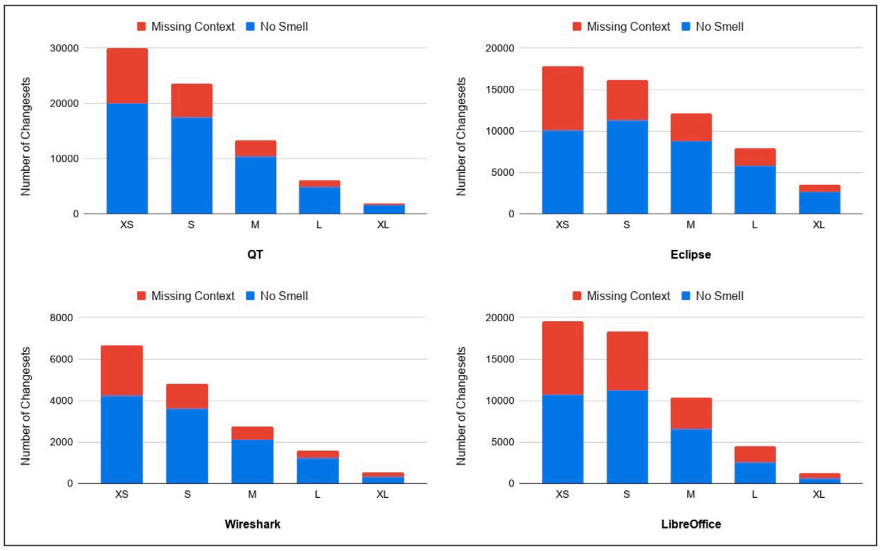
Review Buddies: While Gerrit projects have a significant number of developers with a review buddy, GitHub projects are not affected by this smell that much. The main reason behind this difference is the structural difference between Gerrit and GitHub projects. Gerrit projects are larger in terms of the number of developers and code review instances. In GitHub, the average number of contributions per developer is lower than Gerrit resulting a smaller number of developers with a review buddy.

Reviewer-Author Ping-pong: On average, code reviews in Gerrit take slightly larger numbers of iterations compared with the GitHub projects (5.8% and 3.2%). 11.5% of the code reviews in the QT project are affected by this smell where LibreOffice and Django projects have the best results among all.

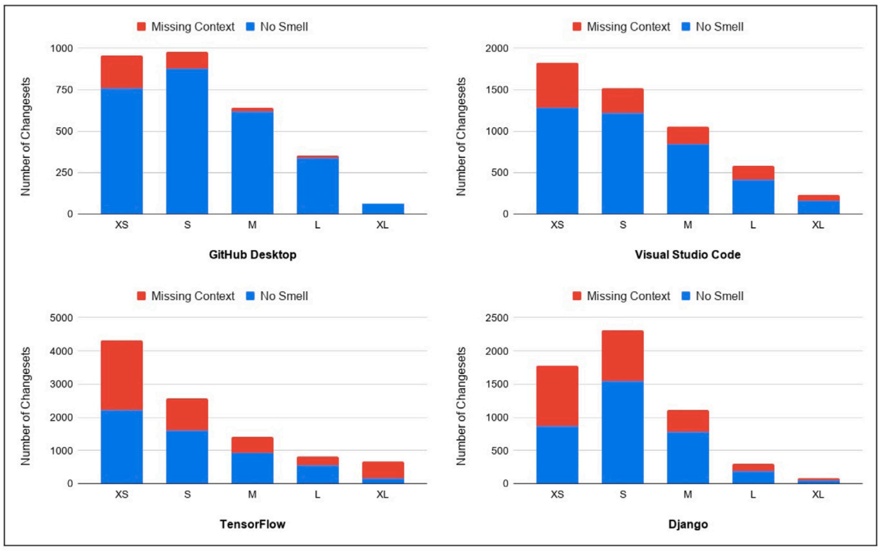
Sleeping Reviews: This smell depends on the project structure rather than the code review platform. Each project's guidelines define the maximum time for a code review task differently and these restrictions affect the sleeping review percentages among different projects.

The smells *Large Changesets* and *Missing Context in Reviews* show very similar characteristics among GitHub and Gerrit platforms. Although there are some project-based differences, the total results of platforms are close to each other.

In summary, 71.1% of Gerrit and 81.4% of GitHub code review instances are affected by at least one smell. The QT project shows the best results with 53.1% among 8 projects. One possible reason might be the comprehensive guidelines of QT on the code review [42]. The other seven projects are affected by at least one code review smell with a range of 62.4% to 89.5%.



(a) Gerrit Projects



(b) GitHub Projects

Fig. 10. Number of different-sized changesets with the smell: *Missing Context in Code Reviews* in four Gerrit (a) and four GitHub (b) projects.

Table 15
Ratios of code reviews with missing context smell for different sizes (XS: 0–10, S: 10–50, M: 50–200, L: 200–1000, XL: 1000+).

Changeset size	QT	Eclipse	Wireshark	LibreOffice	GitHub Desktop	VS Code	TensorFlow	Django
XS	0.33	0.43	0.36	0.45	0.21	0.29	0.49	0.51
S	0.26	0.30	0.25	0.38	0.10	0.20	0.38	0.33
M	0.21	0.28	0.23	0.36	0.04	0.19	0.34	0.29
L	0.19	0.27	0.22	0.43	0.03	0.29	0.33	0.37
XL	0.19	0.25	0.40	0.51	0.02	0.30	0.76	0.43

Table 16
Percentages of the code review instances affected by each smell and the *p*-value for the corresponding hypothesis.

	Lack of review	Ping pong	Sleeping review	Missing context	Combined smell
Smell percentage of community projects (%)	53.49	2.33	22.95	35.95	81.93
Smell percentage of company projects (%)	8.30	10.05	25.59	28.34	58.11
p-value	0.00	0.00	2.40e−47	0.00	0.00

6.2. Selecting the right thresholds

Out of the seven code review smell categories that we described, three of the code review smell definitions are based on selecting some thresholds. We used thresholds in review buddies (50 or more contributions required per author), ping-pong (more than 3 code review iterations), sleeping review (a code review taking place more than 48 h) and large changesets (changeset is more than 500 LOC). These

Table 17
P-values for the statistical test of different-sized changesets (XS, S, M, L, XL) and the smelling code review instances for eight projects and their combination.

	Lack of review	Ping pong	Sleeping review	Missing context	Combined smell
QT	5.52e−31	0.00	0.00	1.50e−240	0.00
Eclipse	5.94e−128	2.93e−256	0.00	2.68e−293	3.55e−260
Wireshark	4.48e−09	3.04e−120	7.12e−125	1.41e−67	4.11e−103
LibreOffice	2.91e−136	2.26e−53	7.93e−106	5.74–70	1.53e−142
GitHub Desktop	0.13	5.84e−35	1.41e−72	3.77e−31	1.02e−37
VS Code	1.39e−17	1.09e−15	2.85e−15	4.31e−14	5.63e−18
TensorFlow	9.19e−29	1.04e−161	1.99e−129	1.11e−96	3.70e−56
Django	6.34e−48	9.41e−06	1.20e−110	1.24e−40	2.19e−15
Total	2.73e−25	0.00	0.00	0.00	0.00

Table 18
Survey results for the question: *How many changed LOC should a changeset/pull request include at most?*

Number of changed lines of code	Number of survey respondents
0–200	13
200–500	15
500–1000	4
1000+	0

Table 19
Survey answers to the question: *What should be the maximum number of iterations for the loop between the author and the reviewer?*

Number of iterations	Number of survey respondents
1	0
2	4
3	18
4	4
5+	6

thresholds actually depend on some project characteristics (such as size of the project and domain) and project contribution guidelines. In this study, to select these thresholds we relied on gray literature review and expert opinion of the surveyed developers. When applying the findings of this study in practice, we believe that the practitioners should pick their own thresholds based on the specifics of their projects. In the following, we provide justification of our threshold choices.

Threshold for The Large Changeset Smell: As given in Table 4, all of 32 survey respondents agreed on the smell definition. Beyond that, 28 respondents agreed on that a pull request to be reviewed should not exceed 500 lines of code. The detailed responses of survey participants are given in Table 18. We also plot the LOC distribution graphs of the eight projects in Fig. 11. It can be seen that the majority of the code review instances have less than 500 changed LOC.

Threshold for Ping-pong Smell: To the question asked in the survey, *What should be the maximum number of iterations for the loop between the author and the reviewer?*, 22 out of 32 respondents agreed on that the loop between the code author and reviewer should not exceed more than three iterations (Table 19). Also, the eight projects used in our empirical analysis are analyzed in terms of the number of iterations in the code review processes. As it can be seen in Fig. 12, the majority of the code review instances consists of three of less iterations between the author and reviewer.

Threshold for Review Buddies Smell: We analyzed the distribution of the number of contributions made by developers in Gerrit and GitHub projects used in our empirical analysis. The related histograms for the number of contributions made by a developer are given in Fig. 13. The results indicate that all projects have a highly right-skewed distribution meaning that there are lots of developers with less than 50 contributions. Since we are looking for the core contributors of OSS projects (by core, the developers, who have contributed to the project enough to have a review buddy), we decided to keep the related threshold as 50. This threshold could be configured differently depending on the definition of the core contributors.

6.3. Implications for research

Classifications form a body of knowledge within a field, enabling researchers and practitioners to generalize, communicate, and apply the findings [86]. In the literature, many taxonomies have been proposed in all software engineering (SE) knowledge areas to provide a systematic description and organization of the investigated subjects. According to Usman et al. [87], out of 271 taxonomies published in the literature, only Bayona-Oré et al. [88] described a systematic approach to develop their taxonomy, whereas the remaining 270 taxonomies are described in an ad-hoc manner. The authors [87] identified some issues associated with Bayona-Oré et al.'s taxonomy design method. Thus, they proposed a revised taxonomy development methodology. We mapped our taxonomy development methodology according to the guidelines in [87] in Table 20.

In this study our intention is to classify and organize the bad practices/anti-patterns in the code review practice. The subject matter is the code review bad practices in this study. The classification procedure as we have described in our study is based on an MLR. We further verified this taxonomy with expert opinion using a survey methodology. Finally, we validated the existence of the proposed code review smells in 8 different open source projects. In our taxonomy, we defined our smell categories with a short descriptive name, a longer definition, possible root causes for this smell, and potential side effects (Table 3). As we have discussed in the paper, we further define the details of a code review smell with possible (and configurable) thresholds, and their corresponding smell detection mechanism.

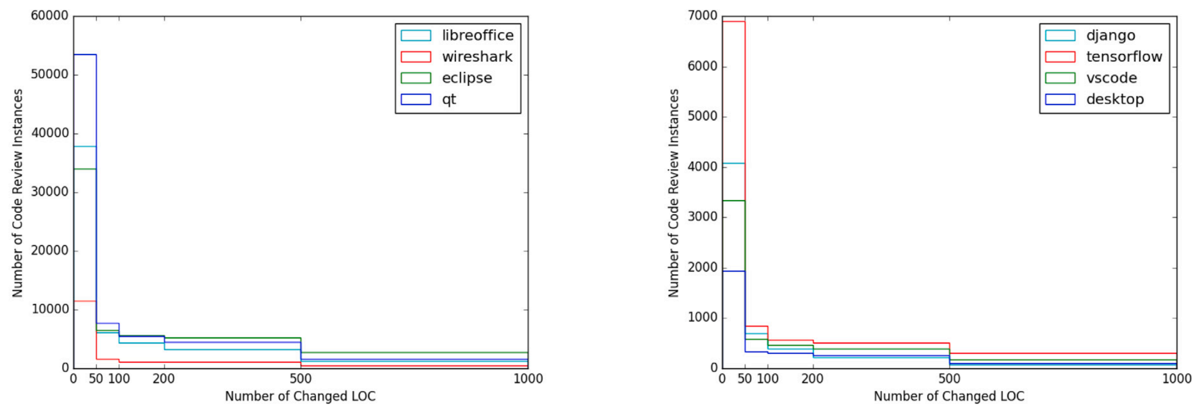
With the introduction of code review smell taxonomy,

- We provided a common terminology and a taxonomy for code review smells, which eases the sharing of knowledge among researchers.

- The taxonomy and the overall classification effort helps to identify gaps in the knowledge. For example, as we discussed earlier, it is not straightforward to mine LGTM smells since it is difficult to differentiate between a proper LGTM (reviewer has done a thorough job and was not able to find defects) vs an LGTM smell (the reviewer just approved the code review without adequately reviewing). Another possible research direction is to quantify the impact of these code review smells in productivity or rework.

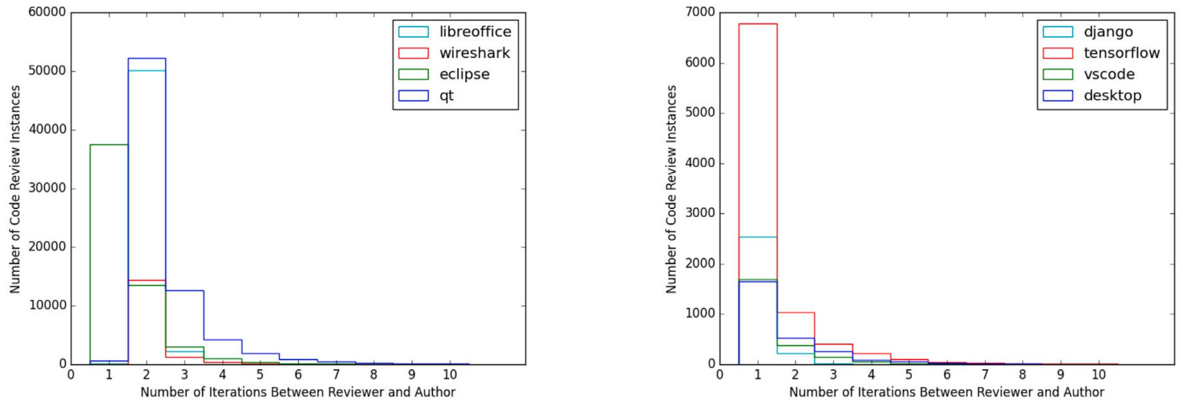
- As all the other SE taxonomies, we expected that the code review smell taxonomy will evolve over time incorporating new knowledge

Code review smells are also closely correlated with the software development waste research [89]. Waste is defined as any activity that consumes time, resources, or space but does not add any value to the product as perceived by the customer. According to Sedano et al. [89], there are nine software development wastes. We found that the smells in the code review process could potentially lead to five different types of waste. Process smells in the code review process might lead to rework (lack of review and LGTM), extraneous cognitive load (missing context and large changeset), waiting/multitasking (sleeping review), knowledge loss (review buddies smell), and ineffective communication (ping-pong smell). In the future, we are planning to explore the impact of code review smells on software development waste.



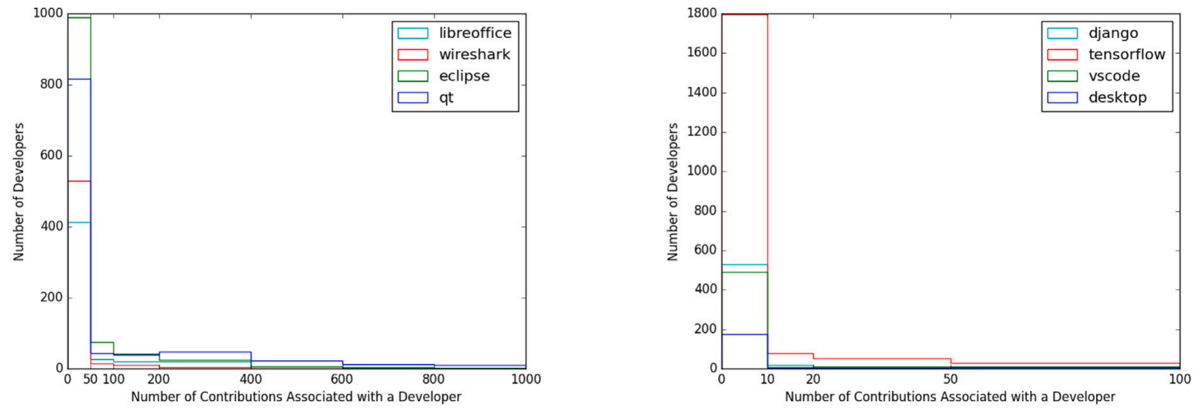
(a) Gerrit Project (b) GitHub Project

Fig. 11. Histograms for the numbers of the changed LOC in Gerrit and GitHub projects.



(a) Gerrit Project (b) GitHub Project

Fig. 12. Histograms for the number of iterations between author & reviewer in Gerrit and GitHub projects.



(a) Gerrit Project (b) GitHub Project

Fig. 13. Histograms for the number of developer contributions in Gerrit and GitHub projects.

6.4. Implications for industry and software engineering practice

The survey results reveal that the code review smells proposed in our taxonomy are considered as critical actions and should be avoided in order to enhance the software development process. Also, the results indicate that all of the code review smells introduced in our taxonomy exist in different ratios. The implications of this study for software engineering practice are listed as follows:

- Practitioners can use the proposed taxonomy to potentially avoid the code review smells. To this end, proper code review guidelines and rules can be prepared (or they can be updated if already

Table 20
The taxonomy development activities according to the guidelines at [87].

Taxonomy development activity	Description
B1: Define SE knowledge area	The software engineering knowledge area associated to the designed taxonomy is Software Quality.
B2: Describe the objectives of the taxonomy	The main objective of the proposed taxonomy is to define a set of categories that enables to classify process smells in the code review process.
B3: Describe the subject matter to be classified	The subject matter of the designed taxonomy is the anti-patterns/bad practices/smells in the code review process.
B4: Select classification structure type	The taxonomy was designed using a hierarchy classification structure.
B5: Select classification procedure type	The procedure used to classify the code review smells is qualitative.
B6: Identify the sources of information	The basis of the taxonomy consists of code review smells drawn from both multivocal literature review and a developer survey conducted with 32 professionals.
B7: Extract all terms	All the categories were extracted from the literature.
B8: Perform terminology control	It was not necessary to perform terminology control, because the categories are the result of an aggregation of concepts from existing literature.
B9: Identify and describe taxonomy dimensions	Since it is a hierarchical classification structure, it has a single dimension.
B10: Identify and describe categories of each dimension	Seven categories were identified, as follows: lack of a code review, review buddies, reviewer-author ping-pong, looks good to me reviews, sleeping reviews, missing context in reviews and large changesets. The categories are further identified by stating the short name, description, representative example, root causes, potential impacts, (possible) thresholds, and a corresponding detection method.
B11 : Identify and describe the relationships	Not applicable.
B12: Define the guidelines for using and updating the taxonomy	The taxonomy deriving process is described. The necessary scripts to mine each code review smell type for popular code review tools (GitHub and Gerrit) are provided publicly.
B13: Validate the taxonomy	To validate the definitions of the taxonomy, we conducted a survey with expert professionals. To validate the existence of code review smells in practice, we mined 4 repositories of GitHub, and 4 repositories of Gerrit.

exist). The existence of such guidelines does not guarantee to avoid all smells but can decrease the smell percentages. For example, the QT project has the best results for the smell *lack of review* within our empirical analysis. This might be due to the strict warning in the QT Review Policy [42] about the lack of review smell.

- Practitioners can enhance their code review process by introducing appropriate tooling for code review. For example, code review tools can be configured in order to block developers to merge unreviewed/self-reviewed changesets. Again, reminding developers the review task with periodic e-mails can reduce the possibility of sleeping reviews. For instance, the tool *Pull Reminders*[6] notifies the developers with Slack notifications in order to remind the forgotten pull requests and avoid the smell: *sleeping reviews*.
- The initial taxonomy can be used as a starting point to develop (semi) automated recommendation systems to detect code review smells by mining software repositories. These tools are not only limited to the code review smells but can be generalized among the bad practices followed in different steps of the software development such as bug life cycle, testing and continuous integration. Detecting bad practices in different steps can enhance the software process quality in a more significant way.
- Software development life cycle consists of different steps. The previous work in this area investigated the bad practices followed within some of these steps. Garcia et al. [90] introduced bad smells in the software architectures. Rompaey et al. [91] defined the symptoms of poorly designed tests as *test smells*. Zampetti et al. [92] categorized the bad practices followed in the continuous integration process. In the future, other steps/processes within the software development life cycle can be investigated to detect and avoid the smells.

7. Threats to validity

This section discusses the threats to the internal and external validity of our study. Internal validity is concerned with the causal relations investigated within the study [93]. To minimize the risk of any subjective activity during white & gray literature review, web searches are conducted in the private mode of browsers.

During the systematic literature review, we mainly had two validity threats. (1) *Digital libraries:* The selection of digital libraries to a conduct a systematic review is crucial. To mitigate this issue, we searched three of the most popular digital libraries: IEEE Xplore, ACM Digital Library and Springer. Although it is not feasible to check all digital libraries, we tried to mitigate this threat by applying forward & backward snowballing. (2) *Search string:* Before creating our search string, we examined a few previous studies investigating anti-patterns in other software engineering domains. Then, we created our initial draft search string and made the first search by using it. As we came across with more related studies, we adjusted our string and repeated the search process.

Regarding the developer survey, there are two main potential threats to validity: (1) The respondents may misunderstand the smell definitions. (2) Some of the inexperienced respondents may give misleading answers. To mitigate these issues, a detailed description for each smell with a real-life example is provided and the survey is conducted only with experienced software practitioners working in industry. The respondents were also told to contact the authors if they would need any further clarifications related to the survey. After the survey, we asked all the participants whether they wanted to be further interviewed on their answers to the open-ended survey questions. Although the self-volunteer based participation in the interview exposes a self-selection bias threat, the main purpose of these follow-up interviews is to gather further insights and possible clarifications, and since these interviews do not have an effect on the survey statistics, we believe the impact of the threat is not significant.

Our empirical analysis is designed to observe the occurrence ratio of each smell by keeping all other parameters fixed. To increase the replicability of our study, we shared the datasets[7] and the source code[8] online. Some other validity threats regarding our study setup and dataset can be listed in the following way:

- **Data Preprocessing:** In the data cleaning step, code review instances other than the merged ones are ignored. We applied the same preprocessing step for all the smells to ensure consistency. The reason behind this choice was the fact that the specific code

6 https://pullreminders.com/.

7 https://doi.org/10.6084/m9.figshare.13040474.

8 https://doi.org/10.6084/m9.figshare.12890864.

review instance has not been finalized yet. This decision has a potential risk of underestimating the sleeping review counts, since there might be some sleeping code review instances that will never be merged.

- **Tool Dependency:** In our empirical analysis, code review histories of four Gerrit and four GitHub projects are analyzed. Although our taxonomy consists of code review smells defined in a generalized manner, the study setup for each smell detection process is influenced by Gerrit and GitHub specific features.
- **Assumptions in the Smell Definitions:** Although we derive the list of code review smells by elaborately scanning the literature and surveying developers, these smells might not be valid for each scenario. For instance, the review buddies smell is mentioned in the literature many times and recognized by the survey respondents. However, there might be still some scenarios where a review buddy is conducting proper reviews and not leading to a suboptimal scenario.
- **Configurable Thresholds:** Within the smell detection methods, we made some assumptions on the configurable parameters and definitions. Although we justified these thresholds by getting expert opinions through the survey, these thresholds are still subject to discussion and could be configured depending on the project. In our survey, we did not ask the participants about their OSS experience since the code review smell categories that we identified from our MLR are not really specific to OSS or commercial projects. We argue that the smell categories would apply to both scenarios. The respondents' OSS experience could potentially be an issue for selecting thresholds. We used thresholds in review buddies (50 or more contributions per author), ping-pong (more than 3 iterations), sleeping review (a code review taking place more than 48 h) and large changesets (large change is more than 500 LOC). These thresholds actually depend on some project characteristics (such as size and domain) and project contribution guidelines as well. So even if we surveyed developers with high OSS experience, this (not all the OSS projects are the same) could still be a threat.

  For instance, the threshold value of 50 changesets defined in the review buddies smell is highly dependent on the project size. Since Gerrit projects in our empirical analysis are larger than GitHub projects in terms of the number of code review instances, using the same threshold value for all projects might pose a threat to the validity of our results. Another example of a potential threat is related to selecting threshold value as two days without considering the weekends and holidays. Depending on the nature of a project, working on a weekend may or may not be expected. As a future work, some rules can be formulated in order to obtain project dependent threshold values.

Threats to external validity are concerned with to what extent our results can be generalized [93]. To mitigate this threat, our study is evaluated empirically on eight large OSS projects using Gerrit or GitHub as the code review tool. As future work, we are planning to evaluate code review smells on closed-source projects and other code review platforms to diminish the generalizability concerns. The survey conducted with 32 experienced professionals also supports the importance and existence of code reviews smells in practice.

8. Conclusion and future work

In this study, we propose a taxonomy of code review smells to demonstrate bad practices in the code review process. The taxonomy is based on a multivocal literature review which later further validated by 32 expert software professionals. Our taxonomy consists of seven code review smells (lack of a code review, review buddies, reviewer-author ping-pong, looks good to me reviews, sleeping reviews, missing context in reviews and large changesets). To demonstrate the existence of these code review smells, we conduct an empirical evaluation by mining code review histories of eight open source projects: QT, Eclipse, Wireshark, LibreOffice, GitHub Desktop, Visual Studio Code, Tensorflow and Django. Some of our findings from the investigation of 226,292 code review instances are listed below:

- 35.2% of the changesets are merged to the codebase with a self-review or no review at all.
- 27.8% of developers in eight projects have a review buddy.
- 5.5% of code review instances take more than three review iterations.
- 24.0% of the code review instances take longer than two days.
- 32.8% of code reviews have a missing context.
- 72.2% of the code reviews among eight projects are affected by at least one code review smell.

These findings reveal that OSS projects are affected by the code review smells within our taxonomy. These smells are potential threats to some appreciated values among the software development communities such as code quality, team assessment and productivity.

In our study, each code review smell is evaluated by getting domain experts' opinions and illustrating the occurrence ratios in eight open source projects. A future direction would be to measure the impact of each smell on code quality in order to observe the bad effects of such practices quantitatively.

Another future work for this study is to implement some practical tools to detect the CR smells. After the term *code smell* is introduced by Kent Beck [94], several smell detection tools have been proposed in order to enhance software maintainability by automatically detecting code smells [95]. Similarly, a detection tool for code review smells is essential in order to speed up the process and enhance the review quality. In addition to smell detection, some tools/extensions can be useful to avoid these smells *(e.g., reminder mails for sleeping reviews, unreviewed/self-reviewed PR blocker etc.)*. The usage of such tools in the software development teams can reduce the risk of wasting the developers' effort and time.

As explained in Section 4, *LGTM Reviews* smell is excluded in our empirical analysis. The proper detection of this smell remains as a future direction. Although we scanned both white & gray literature, we may have missed some other bad practices in the code review process. As future work, our taxonomy can be extended with these smells and their corresponding empirical analysis.

CRediT authorship contribution statement

Emre Doğan: Methodology, Software, Formal analysis, Investigation, Data curation, Writing – original draft, Visualization. **Eray Tüzün:** Conceptualization, Validation, Resources, Writing – review & editing, Supervision, Project administration.

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.

Acknowledgments

This study was supported by The Scientific and Technological Research Council of Turkey (TUBITAK) 1505 program *(Project Number: 5200078)*.

Appendix. Literature sources

As the result of the multivocal literature review, 20 academic studies and 19 gray literature sources are shared with the corresponding smells in Table A.21.

Table A.21
Literature review results with the related smells.

Source	Type	Lack of code review	Review buddies	Reviewer-author ping-pong	LGTM reviews	Sleeping reviews	Missing context	Large changesets
The impact of code review coverage and code review participation on software quality [3]	White	✓			✓			
An empirical study of the impact of modern code review practices on software quality [38]	White	✓			✓			
Four eyes are better than two: On the impact of code reviews on software quality [41]	White	✓						
Review participation in modern code review [39]	White	✓						
Investigating code review quality: Do people and participation matter? [40]	White	✓						
Anti-patterns in Modern Code Review: Symptoms and Prevalence [11]	White	✓			✓		✓	
Individual, social and personnel factors influencing modern code review process [47]	White		✓		✓			
Was my contribution fairly reviewed? [48]	White		✓					
Process aspects and social dynamics of contemporary code review [49]	White		✓					
Code reviewing in the trenches: Challenges and best practices [2]	White			✓		✓	✓	✓
Confusion in code reviews: Reasons, impacts, and coping strategies [51]	White			✓			✓	✓
Pushback: Characterizing and detecting negative interpersonal interactions in code review [52]	White			✓				
Why Does Code Review Work for Open Source Software Communities?[32]	White			✓				
A study of the quality-impacting practices of modern code review at Sony mobile [55]	White				✓			✓
Predicting developers' negative feelings about code review [33]	White				✓	✓		
Code reviews do not find bugs [57]	White					✓		
Software engineering artifact in software development process-linkage between issues and code review processes [60]	White						✓	
What makes a code change easier to review: an empirical investigation on code change reviewability [61]	White						✓	
Small patches get in! [68]	White							✓
The influence of non-technical factors on code review [69]	White							✓
QTReviewPolicy [42]	Gray	✓	✓		✓		✓	
[vtk-developers]NoselfreviewsonGerritplease(e-mail) [43]	Gray	✓						
Whycodereviewsmatter [44]	Gray	✓						
7CodeReviewBestPracticesandDynamicstoImplement [45]	Gray	✓						
10FaultyBehaviorsofCodeReview [46]	Gray	✓	✓			✓	✓	✓
IntroducingCodeReviewandCollaboration [50]	Gray		✓	✓				
HowWeDoCodeReview [53]	Gray			✓	✓	✓		✓
CodeReviewGuidelines-GitLab [54]	Gray			✓				
Google'sEngineeringPracticesDocumentation [56]	Gray				✓	✓	✓	✓
7CodeReviewBestPracticesAndDynamics [58]	Gray					✓		
ContributionGuidelines-LibreOffice [59]	Gray					✓	✓	
LinusTorvalds:'IDoNoCodingAnyMore' [62]	Gray						✓	

(*continued on next page*)

Table A.21 (*continued*).

Source	Type	Lack of code review	Review buddies	Reviewer-author ping-pong	LGTM reviews	Sleeping reviews	Missing context	Large changesets
CodeReviewBestPractices-Palantir [63]	Gray						✓	✓
HowtoMakeGoodCodeReviewsBetter [64]	Gray						✓	
Pullrequestbestpractices [65]	Gray						✓	
ContributionGuidelines-Wireshark [66]	Gray						✓	
HowtoWriteaGoodPullRequestDescription [67]	Gray						✓	
WhatisCodeReview? [70]	Gray							✓
CodeReviewsatGoogle [71]	Gray							✓

References

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

[2] L. MacLeod, M. Greiler, M.-A. Storey, C. Bird, J. Czerwonka, Code reviewing in the trenches: Challenges and best practices, IEEE Softw. 35 (4) (2017) 34–42.

[3] S. McIntosh, Y. Kamei, B. Adams, A.E. Hassan, The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects, in: Proceedings of the 11th Working Conference on Mining Software Repositories, 2014, pp. 192–201.

[4] C. Thompson, D. Wagner, A large-scale study of modern code review and security in open source projects, in: Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering, 2017, pp. 83–92.

[5] The state of code review in 2019: Trends, tools, and insights for dev collaboration, 2020, https://smartbear.com/resources/ebooks/the-state-of-code-review-2019/ (Accessed on 08/28/2020).

[6] Z. Li, P. Avgeriou, P. Liang, A systematic mapping study on technical debt and its management, J. Syst. Softw. 101 (2015) 193–220.

[7] D.A. Tamburri, P. Kruchten, P. Lago, H. van Vliet, What is social debt in software engineering? in: 2013 6th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE), IEEE, 2013, pp. 93–96.

[8] Z. Codabux, B. Williams, Managing technical debt: An industrial case study, in: 2013 4th International Workshop on Managing Technical Debt (MTD), IEEE, 2013, pp. 8–15.

[9] N.S. Alves, L.F. Ribeiro, V. Caires, T.S. Mendes, R.O. Spínola, Towards an ontology of terms on technical debt, in: 2014 Sixth International Workshop on Managing Technical Debt, IEEE, 2014, pp. 1–7.

[10] A. Martini, V. Stray, N.B. Moe, Technical-, social-and process debt in large-scale agile: an exploratory case-study, in: International Conference on Agile Software Development, Springer, 2019, pp. 112–119.

[11] M. Chouchen, A. Ouni, R.G. Kula, D. Wang, P. Thongtanunam, M.W. Mkaouer, K. Matsumoto, Anti-patterns in modern code review: Symptoms and prevalence, in: 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, 2021, pp. 531–535.

[12] V. Garousi, M. Felderer, M.V. Mäntylä, Guidelines for including grey literature and conducting multivocal literature reviews in software engineering, Inf. Softw. Technol. 106 (2019) 101–121.

[13] J. Klünder, R. Hebig, P. Tell, M. Kuhrmann, J. Nakatumba-Nabende, R. Heldal, S. Krusche, M. Fazal-Baqaie, M. Felderer, M.F.G. Bocco, et al., Catching up with method and process practice: An industry-informed baseline for researchers, in: 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), IEEE, 2019, pp. 255–264.

[14] M. Fagan, Design and code inspections to reduce errors in program development, in: Software Pioneers, Springer, 2002, pp. 575–607.

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

[16] L.G. Votta Jr., Does every inspection need a meeting?, in: Proceedings of the 1st ACM SIGSOFT Symposium on Foundations of Software Engineering, 1993, pp. 107–114.

[17] W. Van Der Aalst, Data science in action, in: Process Mining, Springer, 2016, pp. 3–23.

[18] A.R.C. Maita, L.C. Martins, C.R. Lopez Paz, L. Rafferty, P.C. Hung, S.M. Peres, M. Fantinato, A systematic mapping study of process mining, Enterpr. Inf. Syst. 12 (5) (2018) 505–549.

[19] X. Lu, R.S. Mans, D. Fahland, W.M. van der Aalst, Conformance checking in healthcare based on partially ordered event data, in: Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA), IEEE, 2014, pp. 1–8.

[20] M. Ehrendorfer, J.-A. Fassmann, J. Mangler, S. Rinderle-Ma, Conformance checking and classification of manufacturing log data, in: 2019 IEEE 21st Conference on Business Informatics (CBI), Vol. 1, IEEE, 2019, pp. 569–577.

[21] N. Zazworka, V.R. Basili, F. Shull, Tool supported detection and judgment of nonconformance in process execution, in: 2009 3rd International Symposium on Empirical Software Engineering and Measurement, IEEE, 2009, pp. 312–323.

[22] A.M. Lemos, C.C. Sabino, R.M. Lima, C.A. Oliveira, Using process mining in software development process management: A case study, in: 2011 IEEE International Conference on Systems, Man, and Cybernetics, IEEE, 2011, pp. 1181–1186.

[23] W. Poncin, A. Serebrenik, M. Van Den Brand, Process mining software repositories, in: 2011 15th European Conference on Software Maintenance and Reengineering, IEEE, 2011, pp. 5–14.

[24] V. Rubin, C.W. Günther, W.M. Van Der Aalst, E. Kindler, B.F. Van Dongen, W. Schäfer, Process mining framework for software processes, in: International Conference on Software Process, Springer, 2007, pp. 169–181.

[25] B.F. Van Dongen, A.K.A. de Medeiros, H. Verbeek, A. Weijters, W.M. van Der Aalst, The prom framework: A new era in process mining tool support, in: International Conference on Application and Theory of Petri Nets, Springer, 2005, pp. 444–454.

[26] W. Sunindyo, T. Moser, D. Winkler, D. Dhungana, Improving open source software process quality based on defect data mining, in: International Conference on Software Quality, Springer, 2012, pp. 84–102.

[27] M. Gupta, Nirikshan: process mining software repositories to identify inefficiencies, imperfections, and enhance existing process capabilities, in: Companion Proceedings of the 36th International Conference on Software Engineering, 2014, pp. 658–661.

[28] M. Gupta, A. Sureka, Nirikshan: Mining bug report history for discovering process maps, inefficiencies and inconsistencies, in: Proceedings of the 7th India Software Engineering Conference, 2014, pp. 1–10.

[29] M. Gupta, A. Sureka, S. Padmanabhuni, Process mining multiple repositories for software defect resolution from control and organizational perspective, in: Proceedings of the 11th Working Conference on Mining Software Repositories, 2014, pp. 122–131.

[30] T. Baum, O. Liskin, K. Niklas, K. Schneider, Factors influencing code review processes in industry, in: Proceedings of the 2016 24th Acm Sigsoft International Symposium on Foundations of Software Engineering, 2016, pp. 85–96.

[31] N. Fatima, S. Nazir, S. Chuprat, Software engineering wastes-a perspective of modern code review, in: Proceedings of the 3rd International Conference on Software Engineering and Information Management, 2020, pp. 93–99.

[32] A. Alami, M.L. Cohn, A. Wasowski, Why does code review work for open source software communities? in: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), IEEE, 2019, pp. 1073–1083.

[33] C.D. Egelman, E. Murphy-Hill, E. Kammer, M.M. Hodges, C. Green, C. Jaspan, J. Lin, Predicting developers' negative feelings about code review, in: 2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE), IEEE, 2020, pp. 174–185.

[34] M. Caulo, B. Lin, G. Bavota, G. Scanniello, M. Lanza, Knowledge transfer in modern code review, in: Proceedings of the 28th International Conference on Program Comprehension, 2020, pp. 230–240.

[35] C. Bird, T. Carnahan, M. Greiler, Lessons learned from building and deploying a code review analytics platform, in: 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories, IEEE, 2015, pp. 191–201.

[36] M. Di Penta, D.A. Tamburri, Combining quantitative and qualitative studies in empirical software engineering research, in: 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C), IEEE, 2017, pp. 499–500.

[37] B. Kitchenham, S. Charters, Guidelines for Performing Systematic Literature Reviews in Software Engineering, Citeseer, 2007.

[38] S. McIntosh, Y. Kamei, B. Adams, A.E. Hassan, An empirical study of the impact of modern code review practices on software quality, Empir. Softw. Eng. 21 (5) (2016) 2146–2189.

[39] P. Thongtanunam, S. McIntosh, A.E. Hassan, H. Iida, Review participation in modern code review, Empir. Softw. Eng. 22 (2) (2017) 768–817.

[40] O. Kononenko, O. Baysal, L. Guerrouj, Y. Cao, M.W. Godfrey, Investigating code review quality: Do people and participation matter? in: 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2015, pp. 111–120.

[41] G. Bavota, B. Russo, Four eyes are better than two: On the impact of code reviews on software quality, in: 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2015, pp. 81–90.

[42] Review policy - qt wiki, 2020, https://wiki.qt.io/Review_Policy (Accessed on 08/21/2020).

[43] [Vtk-developers] no self reviews on gerrit please, 2020, https://vtk.org/pipermail/vtk-developers/2012-April/011368.html (Accessed on 08/21/2020).

[44] Why code reviews matter (and actually save time!), 2020, https://www.atlassian.com/agile/software-development/code-reviews (Accessed on 08/21/2020).

[45] 7 code review best practices and dynamics to implement (part 1) - dzone agile, 2020, https://dzone.com/articles/7-code-review-best-practices-and-dynamics-to-imple (Accessed on 08/21/2020).

[46] 10 faulty behaviors of code review - itake unconference - speaker deck, 2020, https://speakerdeck.com/lemiorhan/10-faulty-behaviors-of-code-review-itake-unconference (Accessed on 08/21/2020).

[47] N. Fatima, S. Nazir, S. Chuprat, Individual, social and personnel factors influencing modern code review process, in: 2019 IEEE Conference on Open Systems (ICOS), IEEE, 2019, pp. 40–45.

[48] D. German, G. Robles, G. Poo-Caamaño, X. Yang, H. Iida, K. Inoue, "Was my contribution fairly reviewed?" A framework to study the perception of fairness in modern code reviews, in: 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), IEEE, 2018, pp. 523–534.

[49] A. Bosu, J.C. Carver, C. Bird, J. Orbeck, C. Chockley, Process aspects and social dynamics of contemporary code review: Insights from open source development and industrial practice at microsoft, IEEE Trans. Softw. Eng. 43 (1) (2016) 56–75.

[50] Manage pull requests at scale with code review — pluralsight, 2020, https://www.pluralsight.com/blog/platform/introducing-code-review-and-collaboration---a-better-way-to-mana (Accessed on 08/21/2020).

[51] F. Ebert, F. Castor, N. Novielli, A. Serebrenik, Confusion in code reviews: Reasons, impacts, and coping strategies, in: 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, 2019, pp. 49–60.

[52] C.D. Egelman, E. Murphy-Hill, E. Kammer, M.M. Hodges, C. Green, C. Jaspan, J. Lin, Pushback: Characterizing and detecting negative interpersonal interactions in code review, in: 2020 IEEE/ACM 42st International Conference on Software Engineering (ICSE), IEEE, 2020.

[53] How we do code review — app center blog, 2020, https://devblogs.microsoft.com/appcenter/how-the-visual-studio-mobile-center-team-does-code-review/ (Accessed on 08/21/2020).

[54] Code review guidelines — gitlab, 2020, https://docs.gitlab.com/ee/development/code_review.html (Accessed on 08/21/2020).

[55] J. Shimagaki, Y. Kamei, S. McIntosh, A.E. Hassan, N. Ubayashi, A study of the quality-impacting practices of modern code review at Sony mobile, in: Proceedings of the 38th International Conference on Software Engineering Companion, 2016, pp. 212–221.

[56] Google engineering practices documentation — eng-practices, 2020, https://google.github.io/eng-practices/ (Accessed on 08/26/2020).

[57] J. Czerwonka, M. Greiler, J. Tilford, Code reviews do not find bugs. how the current code review best practice slows us down, in: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, 2, IEEE, 2015, pp. 27–28.

[58] 7 code review best practices and dynamics — pluralsight, 2020, https://www.pluralsight.com/blog/platform/code-review-best-practices (Accessed on 08/21/2020).

[59] Development/getinvolved - the document foundation wiki, 2020, https://wiki.documentfoundation.org/Development/GetInvolved (Accessed on 08/26/2020).

[60] D. Dalipaj, J.M. Gonzalez-Barahona, D.I. Cortazar, Software engineering artifact in software development process-linkage between issues and code review processes, in: New Trends in Software Methodologies, Tools and Techniques: Proceedings of the Fifteenth SoMeT_16, Vol. 286, 2016, p. 115.

[61] A. Ram, A.A. Sawant, M. Castelluccio, A. Bacchelli, What makes a code change easier to review: an empirical investigation on code change reviewability, in: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2018, pp. 201–212.

[62] Linus torvalds: 'i do no coding any more' - slashdot, 2020, https://linux.slashdot.org/story/20/07/03/2133201/linus-torvalds-i-do-no-coding-any-more (Accessed on 08/21/2020).

[63] Code review best practices. The internet provides a wealth of…— by palantir — palantir blog — medium, 2020, https://medium.com/palantir/code-review-best-practices-19e02780015f (Accessed on 08/21/2020).

[64] How to make good code reviews better - stack overflow blog, 2020, https://stackoverflow.blog/2019/09/30/how-to-make-good-code-reviews-better/ (Accessed on 08/21/2020).

[65] Pull request best practices - the pragmatic engineer, 2020, https://blog.pragmaticengineer.com/pull-request-or-diff-best-practices/ (Accessed on 08/21/2020).

[66] Development/SubmittingPatches - the wireshark wiki, 2020, https://wiki.wireshark.org/Development/SubmittingPatches (Accessed on 08/26/2020).

[67] S. Sharma, How to write a good pull request description – and why it's important, 2020, URL https://www.freecodecamp.org/news/how-to-write-a-pull-request-description/.

[68] P. Weißgerber, D. Neu, S. Diehl, Small patches get in!, in: Proceedings of the 2008 International Working Conference on Mining software Repositories, 2008, pp. 67–76.

[69] O. Baysal, O. Kononenko, R. Holmes, M.W. Godfrey, The influence of non-technical factors on code review, in: 2013 20th Working Conference on Reverse Engineering (WCRE), IEEE, 2013, pp. 122–131.

[70] What is code review?, 2020, https://smartbear.com/learn/code-review/what-is-code-review/ (Accessed on 08/21/2020).

[71] M. Greiler, Code reviews at google are lightweight and fast, 2020, https://www.michaelagreiler.com/code-reviews-at-google/ (Accessed on 08/21/2020).

[72] D.S. Cruzes, T. Dyba, Recommended steps for thematic synthesis in software engineering, in: 2011 International Symposium on Empirical Software Engineering and Measurement, IEEE, 2011, pp. 275–284.

[73] Eclipse development process — the eclipse foundation, 2020, https://www.eclipse.org/projects/dev_process/development_process_2010.php (Accessed on 08/26/2020).

[74] Desktop, Desktop/Pull-Requests.Md at development . desktop/desktop, 0000. URL https://github.com/desktop/desktop/blob/development/docs/process/pull-requests.md.

[75] Microsoft, How to contribute . microsoft/vscode wiki, 0000. URL https://github.com/microsoft/vscode/wiki/How-to-Contribute.

[76] Tensorflow, Tensorflow/contributing.md at master . tensorflow/tensorflow, 2021, URL https://github.com/tensorflow/tensorflow/blob/master/CONTRIBUTING.md.

[77] Django, Django/contributing.rst at main . django/django, 2015, URL https://github.com/django/django/blob/main/CONTRIBUTING.rst.

[78] E. Doğan, E. Tüzün, K.A. Tecimer, H.A. Güvenir, Investigating the validity of ground truth in code reviewer recommendation studies, in: 2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), IEEE, 2019, pp. 1–6.

[79] M. Fagan, A history of software inspections, in: Software Pioneers, Springer, 2002, pp. 562–573.

[80] P. Rigby, B. Cleary, F. Painchaud, M.-A. Storey, D. German, Contemporary peer review in action: Lessons from open source development, IEEE Softw. 29 (6) (2012) 56–61.

[81] C. Sadowski, E. Söderberg, L. Church, M. Sipko, A. Bacchelli, Modern code review: a case study at Google, in: Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice, 2018, pp. 181–190.

[82] P.C. Rigby, C. Bird, Convergent contemporary software peer review practices, in: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, 2013, pp. 202–212.

[83] R. Oliveto, G. Antoniol, A. Marcus, J. Hayes, Software artefact traceability: the never-ending challenge, in: 2007 IEEE International Conference on Software Maintenance, IEEE, 2007, pp. 485–488.

[84] A. Bosu, M. Greiler, C. Bird, Characteristics of useful code reviews: An empirical study at microsoft, in: 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories, IEEE, 2015, pp. 146–156.

[85] S. Dueñas, V. Cosentino, G. Robles, J.M. Gonzalez-Barahona, Perceval: Software project data at your will, in: Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, 2018, pp. 1–4.

[86] I. Vessey, V. Ramesh, R.L. Glass, A unified classification system for research in the computing disciplines, Inf. Softw. Technol. 47 (4) (2005) 245–255.

[87] M. Usman, R. Britto, J. Börstler, E. Mendes, Taxonomies in software engineering: A systematic mapping study and a revised taxonomy development method, Inf. Softw. Technol. 85 (2017) 43–59.

[88] S. Bayona-Oré, J.A. Calvo-Manzano, G. Cuevas, T. San-Feliu, Critical success factors taxonomy for software process deployment, Softw. Qual. J. 22 (1) (2014) 21–48.

[89] T. Sedano, P. Ralph, C. Péraire, Software development waste, in: 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), IEEE, 2017, pp. 130–140.

[90] J. Garcia, D. Popescu, G. Edwards, N. Medvidovic, Identifying architectural bad smells, in: 2009 13th European Conference on Software Maintenance and Reengineering, IEEE, 2009, pp. 255–258.

[91] B. Van Rompaey, B. Du Bois, S. Demeyer, Characterizing the relative significance of a test smell, in: 2006 22nd IEEE International Conference on Software Maintenance, IEEE, 2006, pp. 391–400.

[92] F. Zampetti, C. Vassallo, S. Panichella, G. Canfora, H. Gall, M. Di Penta, An empirical characterization of bad practices in continuous integration, Empir. Softw. Eng. 25 (2) (2020) 1095–1135.

[93] P. Runeson, M. Höst, Guidelines for conducting and reporting case study research in software engineering, Empir. Softw. Eng. 14 (2) (2009) 131.

[94] M. Fowler, Refactoring: Improving the Design of Existing Code, Addison-Wesley Professional, 2018.

[95] E. Fernandes, J. Oliveira, G. Vale, T. Paiva, E. Figueiredo, A review-based comparative study of bad smell detection tools, in: Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering, 2016, pp. 1–12.