TOWARDS A TAXONOMY OF CODE REVIEW SMELLS

A THESIS SUBMITTED TO

THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE

OF BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR

THE DEGREE OF

MASTER OF SCIENCE

IN

COMPUTER ENGINEERING

By

Emre Doğan

September 2020

TOWARDS A TAXONOMY OF CODE REVIEW SMELLS
By Emre Doğan
September 2020

ABSTRACT

TOWARDS A TAXONOMY OF CODE REVIEW SMELLS

Emre Doğan
M.S. in Computer Engineering
Advisor: Eray Tüzün
Co-Advisor: Halil Altay Güvenir
September 2020

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 is to lead a faster and more effective code review process. Non-conformance of developers to this process does not only hinder the advantages of the code review but can also negatively affect the other steps of the software development life cycle.

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*. To this end, 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 survey with 32 experienced software practitioners and perform follow-up interviews in order to get their expert opinion. Based on the multivocal literature review and expert opinion of experienced developers, a taxonomy of code review smells *(lack of code review, review buddies, reviewer-author ping pong, looks good to me (LGTM) reviews, sleeping reviews, missing context in reviews and large changesets)* is introduced. To quantitatively demonstrate the existence of these smells, we analyze 283,354 code reviews collected from eight OSS projects. We observe that a considerable number of code review smells exist in all projects with varying degree of ratios.

Keywords: modern code review, bad practices, conformance checking, code review smells.

ÖZET

KOD GÖZDEN GEÇİRME SÜRECİNDEKİ KÖTÜ UYGULAMALARIN SINIFLANDIRILMASI

Emre Doğan
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Danışmanı: Eray Tüzün
İkinci Tez Danışmanı: Halil Altay Güvenir
Eylül 2020

Kod gözden geçirme, yazılım geliştirme yaşam döngüsünün önemli aşamalarından birisidir. Bu sürecin temel amacı, kod değişikliğindeki olası hataları kod tabanına göndermeden önce saptamaktır. Kod gözden geçirme süreci konusunda resmi bir fikir birliği olmasa da, birçok şirket ve açık kaynak toplulukları bu süreçte izlenmesi gereken aşamalar ve örnek uygulamalar konusunda birleşmişlerdir. Bu uygulamaların temel amacı, daha hızlı ve etkili kod gözden geçirme süreçlerine sahip olmaktır. Yazılım geliştiricilerinin bu örnek uygulamalara uymayışı, kod gözden geçirme sürecinin avantajlarını ortadan kaldırdığı gibi, yazılım geliştirme yaşam döngüsünün diğer aşamalarını da olumsuz etkileyebilir.

Bu çalışmanın amacı, yazılım geliştiricilerin kod gözden geçirme sürecindeki kötü alışkanlıklarını ampirik olarak incelemektir. Bu amaçla, akademik ve gri literatür taraması yaparak kod gözden geçirme sürecindeki kötü alışkanlıklar bir araya getirildi. Daha sonra, toplanılan veriler 32 tecrübeli yazılım geliştiricisine sorularak onaylatıldı ve bunun sonucunda, kod gözden geçirme sürecindeki kötü alışkanlıkların sınıflandırılması tamamlandı *(kod gözden geçirme yoksunluğu, aynı kişilerin kod gözden geçirmesi, kod gözden geçirme döngüsünün uzaması, özensiz kod gözden geçirme, uzun süren kod gözden geçirme, içeriği belli olmayan kod gözden geçirme)*. Bu alışkanlıkları nicel olarak değerlendirmek için, sekiz açık kaynak projeden topladığımız 283,354 kod gözden geçirme süreci analiz edildi. Sonuç olarak, bu kötü alışkanlıkların ciddi miktarlarda açık kaynak projelerde var olduğu gözlenmiştir.

Anahtar sözcükler: kod gözden geçirme, kötü alışkanlıklar, uygunluk kontrolü, kod gözden geçirme uygunsuzlukları.

Acknowledgement

First and foremost, I would like to thank my advisor Asst. Prof. Eray Tüzün for his endless support and guidance during these two years. Beyond his invaluable contributions to this thesis, he helped me a lot in my personal development. I am also grateful to my co-advisor, Prof. Halil Altay Güvenir, for his valuable guidance during my research studies. I would like to thank the rest of my thesis committee, Prof. Uğur Doğrusöz and Asst. Prof. Hande Alemdar, for spending time to review this thesis.

I would like to thank Alperen Çetin, Barış Ardıç and other BILSEN *(Bilkent University Software Engineering and Data Analytics Research Group)* members for their countless feedback through out my research studies. I am also grateful to my office mates at EA-527. We had some great memories until the pandemic breakdown.

Last by not least, I would like to express my deepest gratitude to my parents, Fatma and Ali Doğan, and my brother Emir Doğan, for their unconditional and endless support for every decision that I have taken up to now. Finally, thanks to my beloved, Gökçe who was always by my side whenever I felt down or anxious.

In Memory of the Thesis in the Time of the Coronavirus

September 2020, Ankara

Contents

List of Figures

List of Tables

Chapter 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*.

1.1 Research Problem

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 a 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 3,126 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. Deviation from the best practices does not only hinder the advantages of the code review but can also negatively affect the other steps of the software development life cycle as well.

In this study, we investigate the bad practices followed in the code review process. To denote these bad practices, we use the term *code review smell*. The intuition for the term *smell* comes from the code smell definition of Kent Beck for the surface indications corresponding to a deeper problem in the source code [6]. Although 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], none of them have gathered them with a systematic approach. To the best of our knowledge, this is the first study to 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) [7] to collect an initial set of bad practices in the code review process *(code review smells)*. After analyzing white literature (17 studies published in conference proceedings and journals) and gray literature (18 sources), an initial set of code review smells is generated.

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 with three of the respondents to discuss further about 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, survey and interview results, we also explore 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 283,354 code review processes.

1.2 Contributions of the Thesis

The main contributions of this thesis are:

- We provide the first taxonomy introduced for the bad practices followed in the code review process, *i.e. code review smells*. To validate these smells, we conduct a multivocal literature review and a developer survey among 32 experienced software practitioners.

- To quantitatively illustrate each smell, an empirical analysis is conducted by mining the code review histories of projects using different code review tools (Gerrit and GitHub).

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

Chapter 2

Background

2.1 Code Review

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. However, the mindset of code review has changed significantly due to the transformation of software development methodologies.

First known code reviews were based on the formal inspection methodology defined by Michael Fagan [1]. 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 [8] and cost ineffectiveness in terms of time and organizational resources [9] have led practitioners to come up with a more lightweight and tool-based code review methodology [8], known as *modern code review*.

Despite some minor changes in different organizations, a generic code review process consists of the following steps:

1. A proper 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.

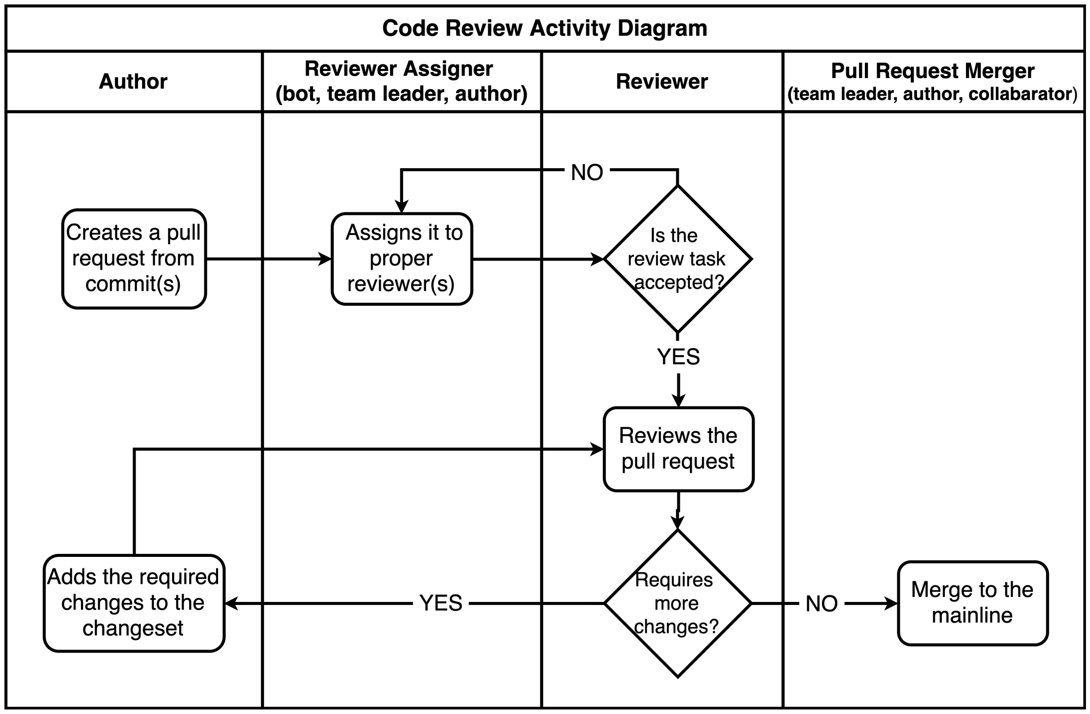
3. At this stage, one or more proper code reviewers should be assigned for the pull request. This assignment can be done by a bot, a team leader or even the authors themselves.

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 number 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 differ in different development teams.

Figure 2.1 and 2.2 illustrates the activity and state diagrams of the generic code review process respectively.



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

Figure 2.1: Activity Diagram for the Code Review Process.

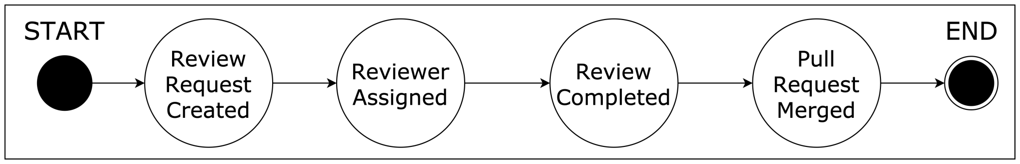


Figure 2.2: State Diagram for the Code Review Process.

2.2 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 [10]. Due to the recent improvements in big data applications, process mining has become a promising subfield of business intelligence and been applied in different domains. It investigates the life cycle of business processes from different aspects by mining event logs [11]. One important type of process mining is business process conformance, checking whether there exists a mismatching between a formally defined process model and real-life event logs progressing this model [11]. In recent years, many different conformance checking studies have been proposed on various industries such as healthcare [12] and manufacturing [13].

Throughout the years, software engineering community has developed and introduced many software development life cycles, models and processes [14]. The main objective of following such processes is to ensure the development of software within the limited resources and limitations (time and budget) [14]. 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. [15] investigate the conformance of software development processes from a Brazilian company with more than 2,000 software 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. [14] introduce a tool-based approach to detect process non-conformances and their future impacts. Poncin et al. [16] and Rubin et al. [17] propose their own software process mining frameworks based on ProM [18], 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. [19] compare the designed and actual processes to support OSS project managers in improving the process flows. Gupta [20, 21] 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 defined and detected [22].

Chapter 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 the qualitative analysis [23].

The overview of research methodology followed in this study is given Figure 3.1. First, a multivocal literature review is conducted to mine an initial set of code review bad practices from both academic and industry perspectives. By combining the outcomes of white and gray literature scanning, an initial set of bad practices in the code review process is achieved. 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 conducted to support our qualitative results *(MLR and semi-structured interviews)* with quantitative analysis.

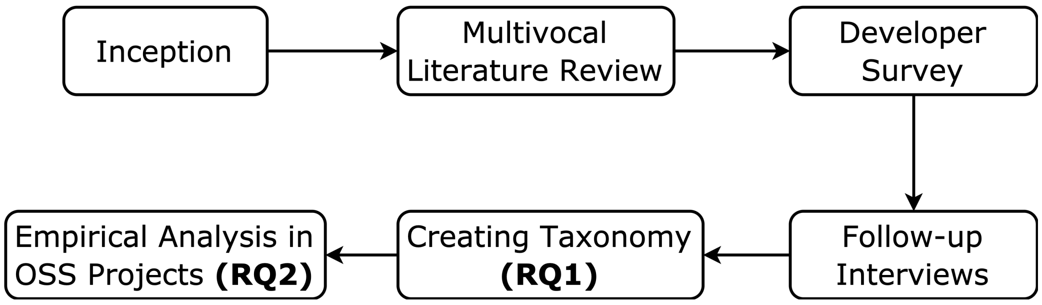

Inception
Multivocal Literature Review
Developer Survey
Follow-up Interviews
Creating Taxonomy (RQ1)
Empirical Analysis in OSS Projects (RQ2)

Figure 3.1: 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, a multivocal literature review is conducted.

3.1.1 White Literature

To scan the white literature, the guidelines of Kitchenham [24] is followed. After developing a review protocol, the relevant studies are searched in Google Scholar and the most popular digital libraries: IEEE Explore, ACM Digital Library and Springer. Before starting the search process, a generic query is defined in the following way:

("code review" OR "code review process") AND ("bad practice" OR "smell*" OR "challenge*")*

The main inclusion criterion for our case is the relevancy of the study to our topic, *i.e. non-ideal practices followed in the code review process.* To ensure this criterion, each study is 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 study. As a result of this step, a list of 17 primary studies is formed.

After digital libraries are searched, each resulting paper is mapped to the related code review smells. The literature sources with their corresponding smells are given in Appendix A.

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. [7]. 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. [7]: (1) Search process, (2) Source selection and (3) Quality assessment of sources.

We run a Google search for the term "code review". Each of resulting 42 pages (411 results) are 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"*). As a selection criterion, we only consider the sources written by a reputable organization or someone linked to a reputable organization without a time restriction since the majority of code review related contents have been published in the last 10 years. Similar to the white literature, a snowballing technique is performed on the resulting sources.

For the quality assessment of resulting sources, the checklist proposed by Garousi et al. is followed [7]. Finally, 18 sources indicating at least one bad practice / smell in the code review process are collected.

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 3.1.

Table 3.1: CR Smells Appeared in White & Gray Literature

Smell Name	White Literature	Gray Literature
Lack of Review	[3, 25–28]	[29–33]
Review Buddies	[34–36]	[29, 33, 37]
Ping-pong	[2, 38, 39]	[37, 40, 41]
Looks Good to Me (LGTM) Reviews	[3, 25, 34, 42]	[29, 40, 43]
Sleeping Review	[2, 44]	[33, 40, 43, 45, 46]
Missing Context	[2, 38, 47, 48]	[29, 33, 43, 46, 49–53]
Large Changesets	[2, 38, 42, 54, 55]	[33, 40, 43, 50, 56, 57]

3.2 Developer Survey & Follow-up Interviews

After completing the multivocal literature review, an extensive developer survey is prepared for software practitioners actively conducting code reviews. The questions of our survey are available online in our replication package[1]. The main objectives of this survey are:

1. To observe whether the definition of code review smells resulted in MLR are agreeable to 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 (inquiring about thresholds for calling an instance as a code review smell).
4. To observe the perception of practitioners on the code review smells.

We contacted 47 experienced developers selected from our personal & professional network to fill out our survey. 32 developers out of 47 fill out the survey

[1]`https://github.com/emredogan7/code-review-smells`

Table 3.2: Demographic Information of the Survey Respondents

Company		**Num. of Employees**	**Num. of Survey Respondents**	**Avg. CR Experience (In years)**
Type	**ID**			
Multinational software company	#1	100,000+	5	14.6
	#2	100,000+	2	6.5
	#3	100,000+	2	21.5
	#4	50,000+	5	15.2
	#5	1,000+	3	6
	#6	1,000+	2	10
Midsize software company	#7	1,000+	6	12.5
	#8	1,000+	1	6
	#9	1,000+	1	4
	#10	200+	2	8.5
Small-sized software company	#11	10-50	1	14
	#12	10-50	1	12
Consultancy	#13	N/A	1	15
	Total respondents:		32	11.7

and three of them agree to perform a follow-up interview. The majority of respondents (19 out of 32) are working for multinational software companies that are considered to be top software companies in the world.

The respondents have an average programming experience of 15.7 years and code review experience of 11.7 years. In the survey, each smell is explained briefly with a real-life example. Then, various questions related to respondents' familiarity with each smell are asked. We also ask 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 3.2.

Then, three respondents are 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 conduct a thematic analysis in order to find developers' opinion on the possible root causes and side effects of each smell.

Table 3.3: Survey Results

	Do you agree with the smell definition?	**How critical is this smell? (Avg. Rating Out of 5)**	**How often do you encounter this smell? (Avg. Rating Out of 5)**
Lack of Review	32/32	4.56	1.66
Review Buddies	31/32	3.59	2.69
Ping-pong	25/32	2.91	2.28
LGTM Reviews	32/32	4.31	2.72
Sleeping Reviews	28/32	3.78	3.00
Missing Context	32/32	3.75	2.34
Large Changesets	32/32	3.72	2.25

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

Chapter 4

Taxonomy of Code Review Smells

The literature review leads us to seven code review smells focusing the bad practices in the code review process from different aspects. Table 4.1 illustrates the definition, root causes and side effects of each code review smell.

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 MLR is illustrated in Table 3.3.

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

Table 4.1: Taxonomy of Code Review Smells

CR Smell	Definition	Root Causes	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).	- Tendency to take the easy way out	- Inefficient 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/inefficient 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.	- The wish of developers to take the easy way out.	- The reviewer does not have enough information about the changeset. - Decrease in the traceability of artifacts
Large Changesets	The changeset is too large to be reviewed.	- Large backlog items - Everything in a single changeset	- Unwilling developers for review - Inefficient reviews

4.1 Synthesizing Literature Reviews & Developer Survey

According to the survey results, majority of the respondents agree with our set of code review bad practices. Relatively lower agreement 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 hours would be called a sleeping review). We adjusted our thresholds (48 hours for sleeping review) according to the survey respondents. Related to the criticality of each smell, except ping-pong smell (2.9/5), the rest of the categories got a score of at least 3.6/5 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 our set of respondents from reputable companies with many years of experience, they were less likely to encounter these smells. (Follow-up interviews and open-ended questions indicated that their company already have 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 sources.

Lack of Code Review: 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)* [30], 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 [29].

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 [29]:

"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 respondent 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."

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 [40] 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."

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 [40]:

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

Sleeping Reviews: Code review speed is a common discussion in the industry. In Google's Engineering Practices documentation, it is stated that [43]:

"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've written. Secondly, the 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."

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 in the following words [49]:

"...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 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..."

Large Changesets: In a blog post of Palantir [50], 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 minutes 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.”

4.2 Final Taxonomy

In the following subsections, each smell is introduced with a detailed explanation, its possible root causes and 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 practices as *lack of code review.*

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.

Side Effects:

This smell leads to missing code reviews that might introduce some future defects in the code.

4.2.2 Review Buddies

Selecting a proper reviewer is an important initial step for effective code reviews [58]. 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].

In the view of this fact, there exists a problematic code reviewer selection when a developer has a tendency to get their changesets reviewed by the same reviewer. We call this type of smell as review buddies.

Root Causes: The main reason behind this smell is the tendency of developers to take the easy way out. 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.

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

4.2.3 Reviewer-Author Ping-pong

According to the defined code review processes in 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*.

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.

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 finds a defect or have some suggestions to the author, they are supposed to state their opinion by providing some feedback through comments. 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 open source community "looks good to me (LGTM)".

Root Causes:

Irrelevant Reviewer: The reviewer is unfamiliar to 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 developer does not pay attention to the review task and just approves the changeset.

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 [59]. Fagan 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 [60]. In Google, code reviews are completed in a short time, with a median of less than 4 hours [61]. Whereas, in the study of Rigby and Bird [62], the median review completion times of Microsoft, AMD, Chrome and Android projects are found to be between 14.7 and 20.8 hours.

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.

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.

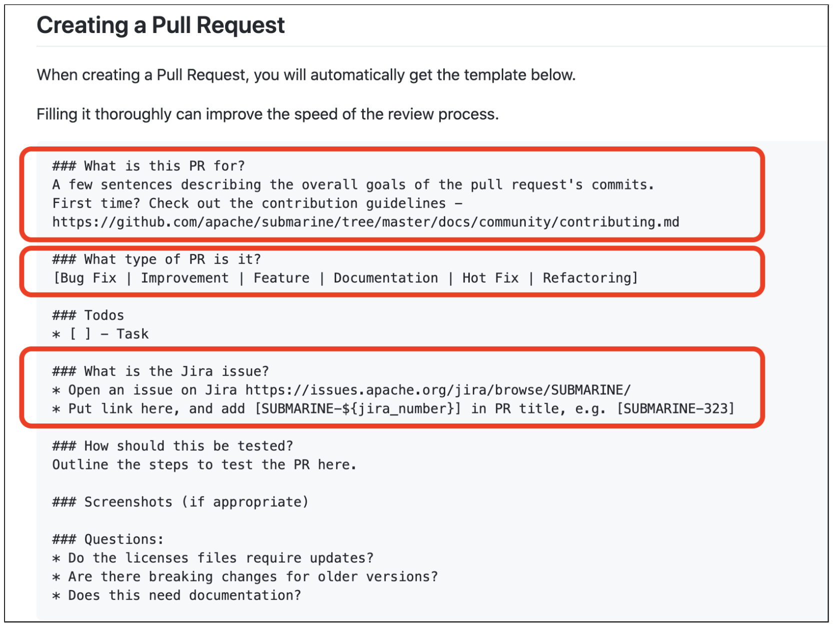
Side Effects:

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 software development and maintenance life cycles [63]. Code review is the inspection of a code changeset which 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?
```

Figure 4.1: Contribution Guideline of the Project Apache Submarine.[1]

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 guideline for submitting a new pull-request to Apache's Submarine repository is given in Figure 4.1. One of the required fields in the given template is the Jira issue related to the commit under review.

Dalipaj et al. [47] also support our claim on the lack of linkage between bug and review repositories of OpenStack project since the developers do not report the related artifact information.

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

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

Root Causes:

The main reason of 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).

Side Effects:

Lack of Traceability: 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 proper explanation in the re-visited changeset.

Lack of Information for Reviewers: The reviewer can not 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 [62]. Large changesets have negative impacts on the review process in different aspects: Rigby et al. [60] find that commits should include small and complete changesets. Bosu et al. [64] and Czerwonka et al. [44] 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. [61] claim that one main reason for fast code reviews at Google is that 90% of code reviews include less 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 to be reviewed.

Root Causes:

Large Backlog Items: If the task is too complicated to realize in a small changeset, then the authors has to create large changesets. To fix such problems, the backlog items should be generated in an atomic manner.

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

Side Effects:

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

Inefficient 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 inefficient reviews introducing possible future bugs.

Chapter 5

Empirical Analysis

This chapter includes the details of experimental 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 an 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 the author to make some additional changes and create a new patch set.

GitHub is a popular Git repository hosting service. Beyond its main purpose as

Table 5.1: 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 5.2: Summary Statistics for Data Collected from Four GitHub Repositories

Project Name	**Total PRs**	**Filtered PRs**	**Start Date**	**End Date**
GitHub Desktop	3,602	2,993	2016-05-11	2020-06-05
Visual Studio Code	7,343	5,206	2015-11-16	2020-06-06
TensorFlow	14,498	9,807	2015-11-09	2020-06-06
Django	13,008	5,578	2012-04-28	2020-06-06
Total	38,451	23,584		

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 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 [65] and made it available online[1]. The summary statistics for Gerrit and GitHub projects are given in Table 5.1 and Table 5.2 respectively.

The empirical analysis is performed on interactive Python notebooks and shared online[2] with instructions to replicate this study.

[1]`https://figshare.com/s/a7691f88aa67dc4bd828`
[2]`https://github.com/emredogan7/code-review-smells`

5.2 Data Cleaning & Preprocessing

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

- The scope of our empirical analysis is limited to the code review processes 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 Table 5.1 and 5.2.

5.3 Quantitative Results

According to the taxonomy detailed in Chapter 4, a detection method for each smell is proposed except for *LGTM Reviews*. The reason to exclude this smell

Table 5.3: Quantitative Results 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	Freq.	Perc.(%)	Freq.	Perc.(%)	Freq.	Perc.(%)	Freq.	Perc.(%)	Freq.	Perc.(%)
Lack of Review	2,987	4.0	28,535	49.6	7,486	45.8	32,621	60.4	71,629	35.3
Ping-pong	21,178	28.3	4,890	8,5	1,966	12.0	2,887	5.3	30,921	15.3
Sleeping Reviews	19,818	26.5	20,004	34.7	2,797	17.1	15,014	27.8	57,633	28.4
Large Changesets	3,564	4.8	6,263	10.9	1,006	6.2	2,508	4.6	13,341	6.6
Missing Context	20,345	27.2	18,883	32.8	4,789	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 5.4: Quantitative Results of Code Review Smells in Four GitHub Projects

	GitHub Desktop		**Visual Studio Code**		**TensorFlow**		**Django**		**Total**	
Total Instances	2,993		5,206		9,807		5,578		23,584	
CR Smells	Freq.	Perc.(%)	Freq.	Perc.(%)	Freq.	Perc.(%)	Freq.	Perc.(%)	Freq.	Perc.(%)
Lack of Review	440	14.7	2,989	57.6	1,273	13.0	3,334	59.9	8,036	34.1
Ping-pong	209	7.0	92	1.8	449	4.6	7	0.1	757	3.2
Sleeping Reviews	1,240	41.4	2,089	40.1	4,690	47.8	1,887	33.8	9,906	42.0
Large Changesets	160	5.3	415	8.0	975	9.9	162	2.9	1,712	7.3
Missing Context	335	11.2	1,277	24.5	4,330	44.2	2,138	38.3	8,080	34.3
Combined	1,868	62.4	4,316	82.9	8,015	81.7	4,990	89.5	19,189	81.4

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 and percentages in eight OSS projects. The resulting statistics for Gerrit and GitHub projects are given in Table 5.3 and 5.4.

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.

Table 5.5: Size Labels of Changesets Introduced in Gerrit

Changed Lines of Code	**Size Label**
[0,10)	XS
[10,50)	S
[50,200)	M
[200,1000)	L
1000+	XL

5.3.1 Lack of Code Review

To detect this smell, the following procedure is followed:

Smell Detection Method:

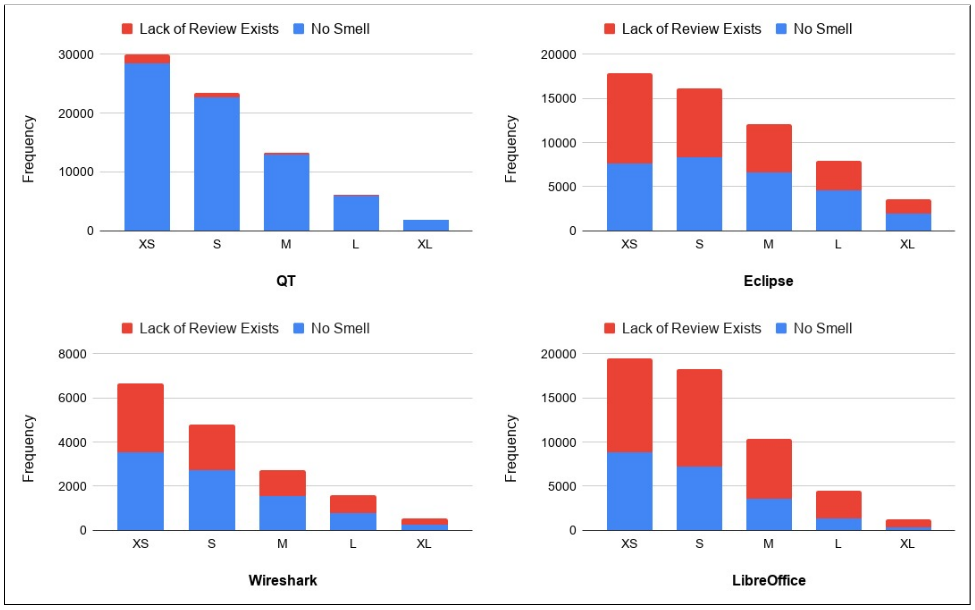
1. If the changeset is merged to the project codebase without a code review, then it is an unreviewed commit.
2. If the one and only reviewer of a changeset is the author of it, then it is a self-reviewed commit.
3. If a review consists of unreviewed or self-reviewed changesets, then it suffers from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

By following these steps, eight projects are examined respectively. The empirical results are given in Table 5.3 and 5.4.

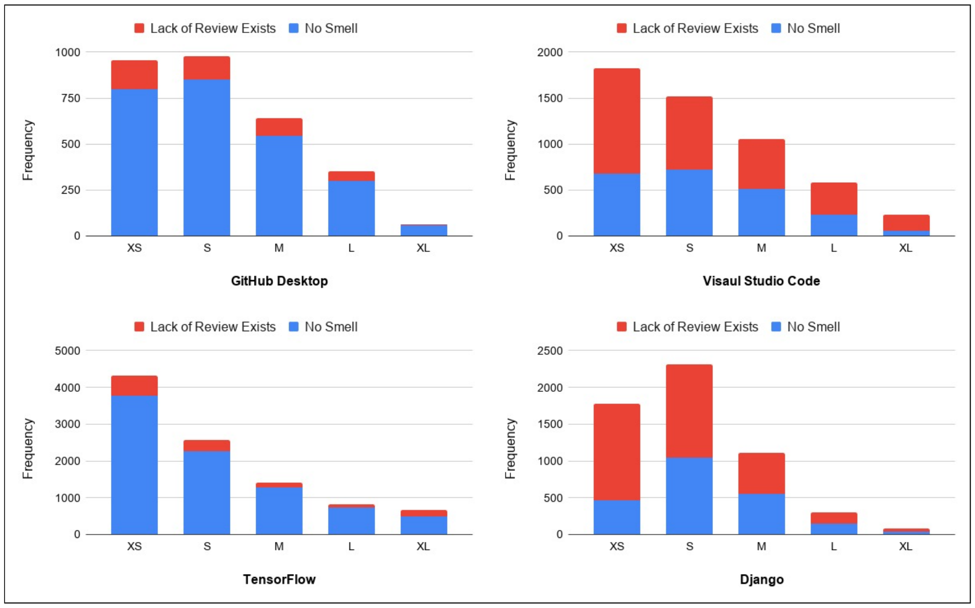
Although Eclipse, Wireshark and LibreOffice projects show similar characteristics (45.8% to 60.4%), QT has a significantly lower smell percentage (4%). Such a major difference leads us to investigate the contribution guidelines & review policies of these four projects. [29, 46, 53, 66]

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

"Do not approve your own changes." [29]



(a) Gerrit Projects



(b) GitHub Projects

Figure 5.1: Frequencies of different-sized changesets with the smell: *Lack of Code Review)* in four Gerrit and GitHub projects

While 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)." [46]

A similar investigation is also conducted on GitHub projects. *Desktop* and *TensorFlow* projects have a 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. Figure 5.1 illustrates the smell frequencies of different sized changesets in Gerrit and GitHub projects. These size intervals (XS through XL) are introduced in Gerrit itself and can be seen in Table 5.5.

The results show that *QT*, *Desktop* and *TensorFlow* projects are not affected by the lack of review smell in a significant manner. In the other projects, the ratios of smelling reviews drop as the commit size increases meaning that small changesets are not reviewed properly.

5.3.2 Review Buddies

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

<u>Smell Detection Method:</u>

1. Self-reviewed and unreviewed commits are eliminated.

2. Commits authored by a developer having less 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. This threshold value is asked of the survey participants and discussed 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 occurrence frequencies 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.

Table 5.6: Review Buddies in Four Gerrit Projects

Project	**Developers Having a Review Buddy**	**Developers Having More Than 50 Contributions**	**Smell Percentage (%)**
QT	39	202	19.3
Eclipse	49	154	31.8
Wireshark	7	33	21.2
LibreOffice	28	79	35.4
Total	123	468	26.3

Table 5.7: Review Buddies in Four GitHub Projects

Project	**Developers Having a Review Buddy**	**Developers Having More Than 50 Contributions**	**Smell Percentage (%)**
GitHub Desktop	0	11	0.0
Visual Studio Code	1	14	7.1
TensorFlow	2	39	5.1
Django	1	12	8.3
Total	4	76	5.3

Since this smell is related to the developers rather than the code review processes, its results are given separately in Table 5.6 and 5.7. It is indicated that more than one fourth of 468 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.

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

5.3.3 Reviewer-Author Ping-pong

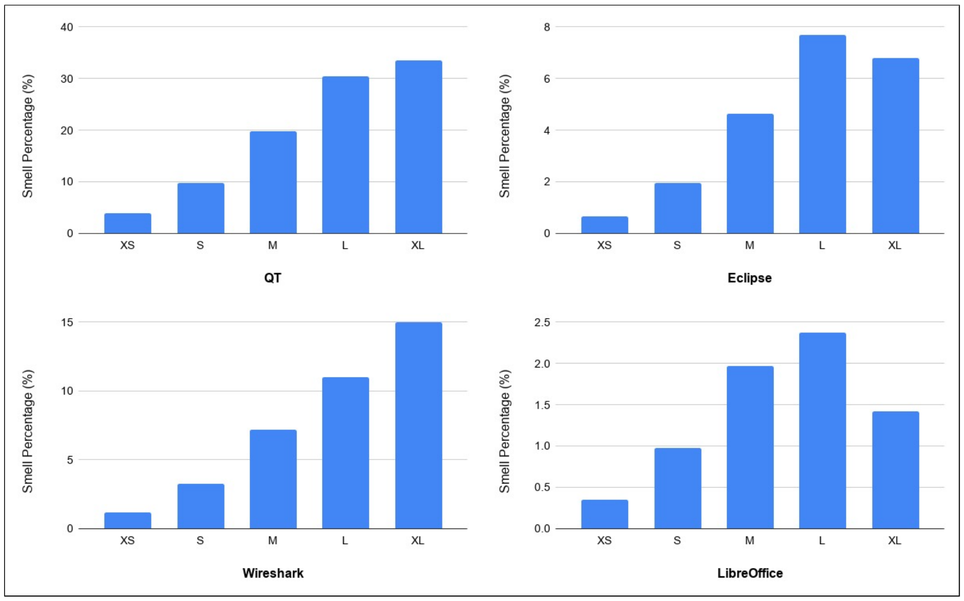
The procedure to detect this smell is given in the following steps:

Smell Detection Method:

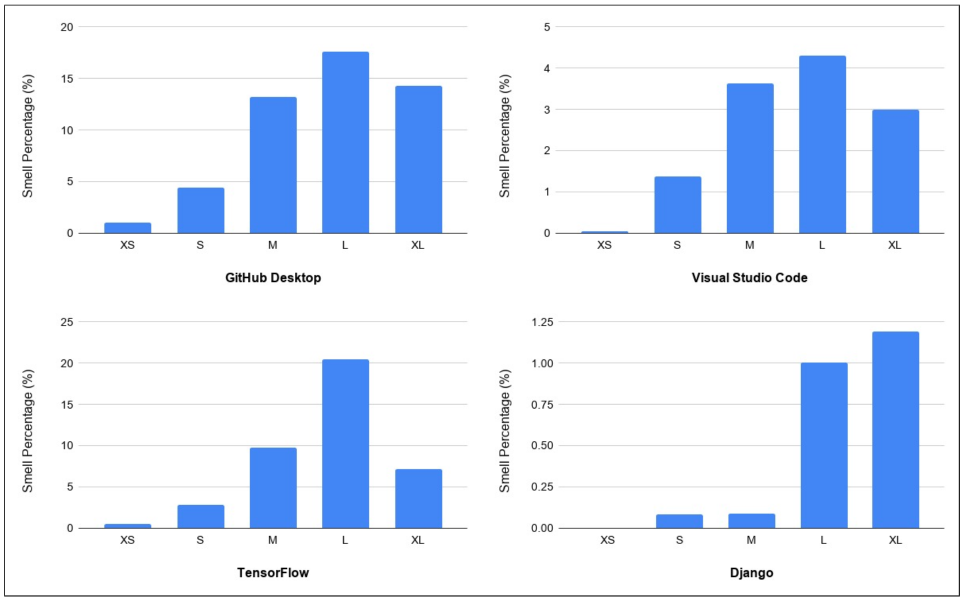
1. If a review process consists of an excessively large number of iterations between the author and reviewer, it is affected by the smell: *reviewer-author ping-pong.*
2. To decide the threshold value for *the excessively large changeset*, the survey participants are asked how many iterations there should be between the author and the reviewer at most. Majority of the respondents (23 out of 32) agree on 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 suffers from the smell: *reviewer-author ping-pong.*

The results in Table 5.3 and 5.4 show that the code review instances in Gerrit projects lead longer author-reviewer iterations than the GitHub projects.

When the smell percentages in different-sized changesets are investigated in Figure 5.2, it is concluded that the number of reviewer-author ping-pong cases increase as the changeset size increases.



(a) Gerrit Projects



(b) GitHub Projects

Figure 5.2: Smell percentages of different-sized changesets with the smell: *Reviewer-Author Ping-pong)* in four Gerrit (a) and four GitHub (b) projects

5.3.4 Sleeping Reviews

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

<u>Smell Detection Method:</u>

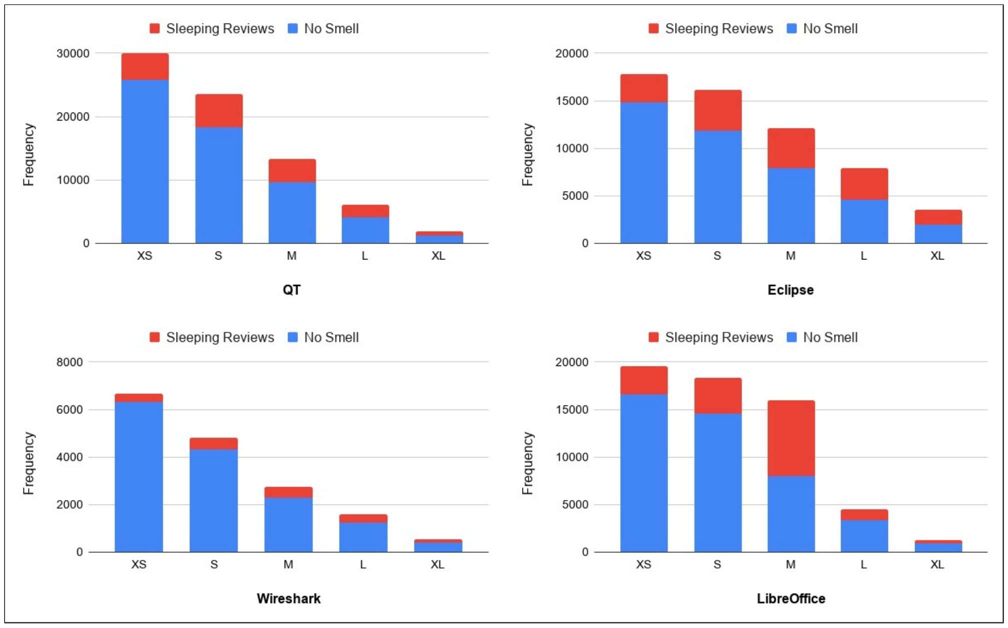
1. The elapsed time between the creation and completion moments of each code review process is calculated and named *review sleeping time (RSTime)*.

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

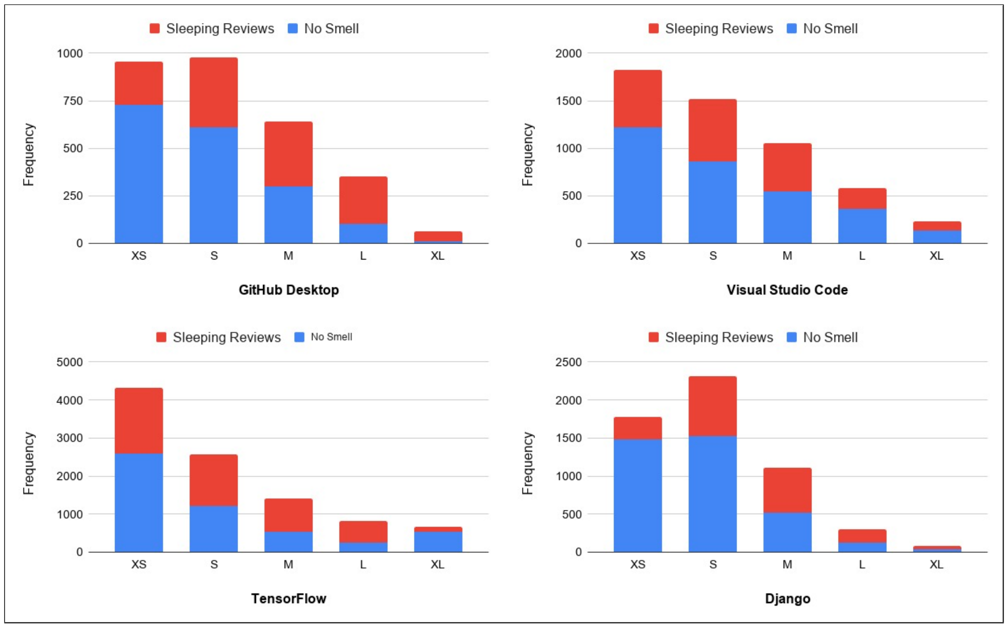
2. The decision of selecting the threshold for a sleeping review is made with respect to the literature review and survey results. In our survey, the question of how long a code review process should take at most is asked of the participants. The majority of the respondents (29 out of 32) claim that a code review process should not exceed two days (48 hours).

3. Relying on the statistics established in white & gray literature and the survey results, a code review process is called a *sleeping review* if it takes more than two days.

Within 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 majority of the open source projects are developed on a volunteer basis and it makes hard to distinguish weekends/holidays from weekdays.



(a) Gerrit Projects



(b) GitHub Projects

Figure 5.3: Frequencies of different-sized changesets with the smell: *Sleeping Reviews)* in four Gerrit (a) and four GitHub (b) projects

The occurrence statistics of this smell are given in Table 5.3 and 5.4. Wireshark seems to have faster code reviews whereas more than one fourth of the reviews in other projects take longer than 48 hours.

We also investigated the relation between the changeset size and sleeping review frequency in each project. The histogram in Figure 5.3 illustrates this relation in each project. It is expected for the reviews of large changesets to take a longer time. However, the long review processes of small changesets indicate some suboptimal review characteristics.

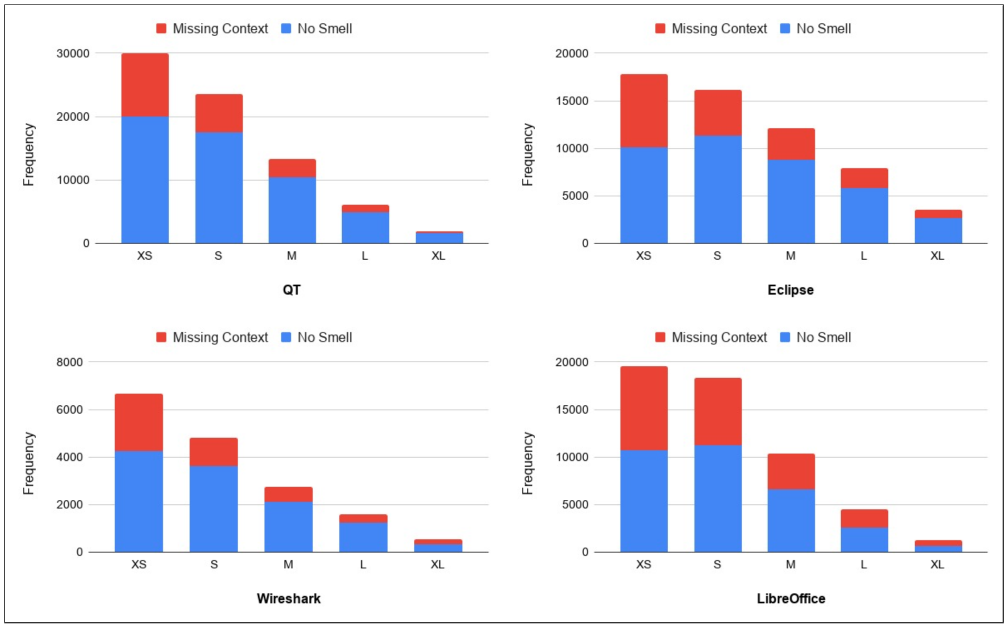
5.3.5 Missing Context in Reviews

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

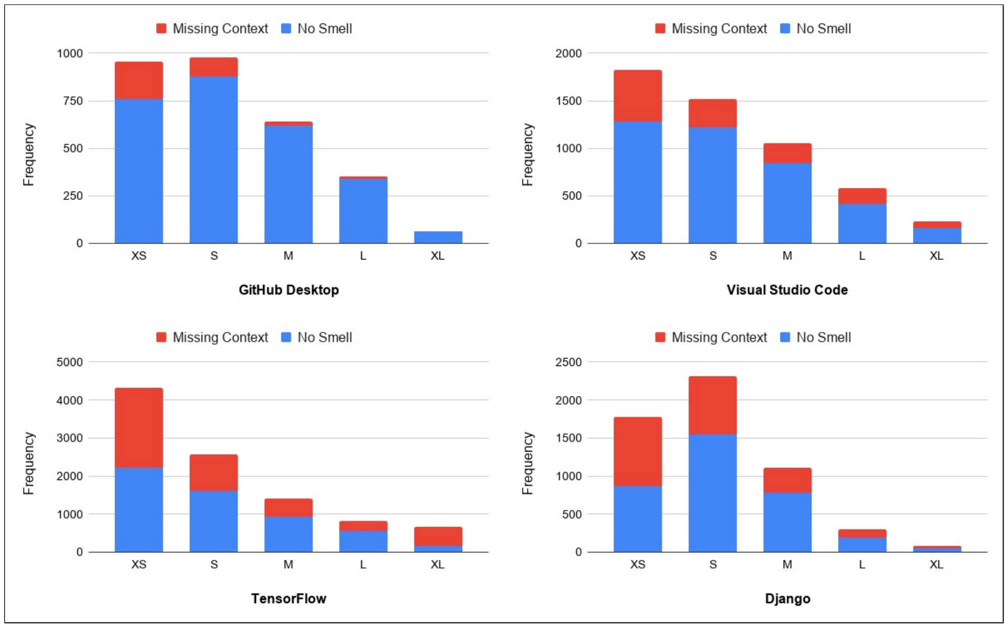
Smell Detection Method:

1. Since each OSS project has its own contribution guideline, 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 guideline.
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.
3. If a review process is not linked to a related issue or 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 copies 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.



(a) Gerrit Projects



(b) GitHub Projects

Figure 5.4: Frequencies of different-sized changesets with the smell: *Misssing Context in Code Reviews)* in four Gerrit (a) and four GitHub (b) projects

The obtained results for this smell are illustrated in Table 5.3 and 5.4. It is shown that almost one code review process out of three suffers from the lack of a proper changeset description.

When the smell occurrence ratios in different-sized changesets are investigated in Figure 5.4, it is clearly seen that this bad practice is more common in the small changesets among both Gerrit and GitHub projects.

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 suffers from *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 (23 out of 32) agreed on 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 Table 5.3 and 5.4. It is illustrated that all of the projects suffer from this smell with a percentage range betweeen 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 frequencies and percentages of the code review processes having at least one code review smell are obtained. The bottom lines of Table 5.3 and 5.4 illustrate that 71.1%

of code reviews in Gerrit and 81.4% of GitHub PRs suffer from at least one smell defined in our taxonomy.

Chapter 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 does not suffer from 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 code review task is dispersed among a larger number of contributors resulting a smaller number of developers with a review buddy.

Reviewer-Author Ping-pong: In average, code reviews in Gerrit take larger numbers of iterations compared with the GitHub projects (15.3% and 3.2%). 28.3% of the code reviews in QT project suffers from this smell where Visual Studio Code and Django projects has the best results among all.

Sleeping Reviews: This smell depends on the project structure rather than the code review platform. Each project 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 [29]. Other seven projects suffer from at least one code review smell with a range of 62.4% to 89.5%.

6.2 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 experiments indicate that all of the code review smells introduced in our taxonomy exist in different ratios. The implications of this thesis 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 exist). Existence of such guidelines does not guarantee to avoid all smells but can decrease the smell percentages. For example, 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 [29] 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*[1] 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. [67] introduced bad smells in the software architectures. Rompaey et al. [68] defined the symptoms of poorly designed tests as *test smells*. Zampetti et al. [69] 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.

[1]`https://pullreminders.com/`

Chapter 7

Threats to Validity

This chapter discusses the threats to internal and external validity of our study.

Internal validity is concerned with the causal relations investigated within the study [70]. To minimize the risk of any subjective activity during white & gray literature review, the web searches are conducted in the private mode of browsers.

Regarding the developer survey, there are two 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 reputable organizations.

Our experiments are designed to observe the occurrence ratio of each smell by keeping all other parameters fixed. Also, for our results not to be affected by the distribution of data, the definition and the detection method of each smell are determined in the planning phase and not changed during the experiments. To increase the replicability of our study, we shared the datasets[1] and the source code[2] online. Some other validity threats regarding our study setup and dataset

[1]`https://figshare.com/s/a7691f88aa67dc4bd828`

[2]`https://github.com/emredogan7/code-review-smells`

can be listed in the following way:

- **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 modified with respect to the Gerrit and GitHub specific features.

- **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 opinion through the survey, these thresholds are still subject to discussion and could be configured depending on the project.

  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. 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 [70]. 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 conducted survey with 32 experienced professionals also support the importance and existence of code reviews smells in practice.

Chapter 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.
- 23.3% of developers in four projects have a review buddy.
- 14.0% of code review instances take more than three review iterations.
- 29.8% 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 suffer from at least one code review smell.

Below is the list of possible extensions to this study and new future directions:

- **Impact Analysis of Code Review Smells:** In our study, each code review smell is evaluated by getting domain experts' opinion 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.

- **Smell Detection Tools:** After the term *code smell* is introduced by Kent Beck [6], several smell detection tools have been proposed in order to enhance software maintainability by automatically detecting code smells [71]. 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.).*

- **Extension of This Taxonomy:** As explained in Chapter 4, *LGTM Reviews* smell is excluded within our experiments. Empirical analysis 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 a future work, our taxonomy can be extended with these smells and their corresponding empirical analysis.

Bibliography

[1] M. Fagan, "Design and code inspections to reduce errors in program development," in *Software pioneers*, pp. 575–607, Springer, 2002.

[2] L. MacLeod, M. Greiler, M.-A. Storey, C. Bird, and J. Czerwonka, "Code reviewing in the trenches: Challenges and best practices," *IEEE Software*, vol. 35, no. 4, pp. 34–42, 2017.

[3] S. McIntosh, Y. Kamei, B. Adams, and 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*, pp. 192–201, 2014.

[4] C. Thompson and 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*, pp. 83–92, 2017.

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

[6] M. Fowler, *Refactoring: improving the design of existing code.* Addison-Wesley Professional, 2018.

[7] V. Garousi, M. Felderer, and M. V. Mäntylä, "Guidelines for including grey literature and conducting multivocal literature reviews in software engineering," *Information and Software Technology*, vol. 106, pp. 101–121, 2019.

[8] M. Beller, A. Bacchelli, A. Zaidman, and 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*, pp. 202–211, 2014.

[9] L. G. Votta Jr, "Does every inspection need a meeting?," in *Proceedings of the 1st ACM SIGSOFT symposium on Foundations of software engineering*, pp. 107–114, 1993.

[10] W. Van Der Aalst, "Data science in action," in *Process mining*, pp. 3–23, Springer, 2016.

[11] A. R. C. Maita, L. C. Martins, C. R. Lopez Paz, L. Rafferty, P. C. Hung, S. M. Peres, and M. Fantinato, "A systematic mapping study of process mining," *Enterprise Information Systems*, vol. 12, no. 5, pp. 505–549, 2018.

[12] X. Lu, R. S. Mans, D. Fahland, and 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)*, pp. 1–8, IEEE, 2014.

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

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

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

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

[17] V. Rubin, C. W. Günther, W. M. Van Der Aalst, E. Kindler, B. F. Van Dongen, and W. Schäfer, "Process mining framework for software processes," in *International conference on software process*, pp. 169–181, Springer, 2007.

[18] B. F. Van Dongen, A. K. A. de Medeiros, H. Verbeek, A. Weijters, and 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*, pp. 444–454, Springer, 2005.

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

[20] 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*, pp. 658–661, 2014.

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

[22] M. Gupta, A. Sureka, and 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*, pp. 122–131, 2014.

[23] M. Di Penta and 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)*, pp. 499–500, IEEE, 2017.

[24] B. Kitchenham and S. Charters, "Guidelines for performing systematic literature reviews in software engineering," 2007.

[25] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan, "An empirical study of the impact of modern code review practices on software quality," *Empirical Software Engineering*, vol. 21, no. 5, pp. 2146–2189, 2016.

[26] P. Thongtanunam, S. McIntosh, A. E. Hassan, and H. Iida, "Review participation in modern code review," *Empirical Software Engineering*, vol. 22, no. 2, pp. 768–817, 2017.

[27] O. Kononenko, O. Baysal, L. Guerrouj, Y. Cao, and M. W. Godfrey, "Investigating code review quality: Do people and participation matter?," in *2015 IEEE international conference on software maintenance and evolution (ICSME)*, pp. 111–120, IEEE, 2015.

[28] G. Bavota and 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)*, pp. 81–90, IEEE, 2015.

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

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

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

[32] "7 code review best practices and dynamics to implement (part 1) - dzone agile." `https://dzone.com/articles/7-code-review-best-practices-and-dynamics-to-imple`. (Accessed on 08/21/2020).

[33] "10 faulty behaviors of code review - itake unconference - speaker deck." `https://speakerdeck.com/lemiorhan/10-faulty-behaviors-of-code-review-itake-unconference`. (Accessed on 08/21/2020).

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

[35] D. German, G. Robles, G. Poo-Caamaño, X. Yang, H. Iida, and 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)*, pp. 523–534, IEEE, 2018.

[36] A. Bosu, J. C. Carver, C. Bird, J. Orbeck, and C. Chockley, "Process aspects and social dynamics of contemporary code review: Insights from open source development and industrial practice at microsoft," *IEEE Transactions on Software Engineering*, vol. 43, no. 1, pp. 56–75, 2016.

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

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

[39] C. D. Egelman, E. Murphy-Hill, E. Kammer, M. M. Hodges, C. Green, C. Jaspan, and 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.

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

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

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

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

[44] J. Czerwonka, M. Greiler, and 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*, vol. 2, pp. 27–28, IEEE, 2015.

[45] "7 code review best practices and dynamics — pluralsight." `https://www.pluralsight.com/blog/platform/code-review-best-practices`. (Accessed on 08/21/2020).

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

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

[48] A. Ram, A. A. Sawant, M. Castelluccio, and 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*, pp. 201–212, 2018.

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

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

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

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

[53] “Development/submittingpatches - the wireshark wiki.” https://wiki.wireshark.org/Development/SubmittingPatches. (Accessed on 08/26/2020).

[54] P. Weißgerber, D. Neu, and S. Diehl, “Small patches get in!,” in *Proceedings of the 2008 international working conference on Mining software repositories*, pp. 67–76, 2008.

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

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

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

[58] E. Doğan, E. Tüzün, K. A. Tecimer, and 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)*, pp. 1–6, IEEE, 2019.

[59] M. Fagan, “A history of software inspections,” in *Software pioneers*, pp. 562–573, Springer, 2002.

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

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

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

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

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

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

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

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

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

[69] F. Zampetti, C. Vassallo, S. Panichella, G. Canfora, H. Gall, and M. Di Penta, “An empirical characterization of bad practices in continuous integration,” *Empirical Software Engineering*, vol. 25, no. 2, pp. 1095–1135, 2020.

[70] P. Runeson and M. Höst, “Guidelines for conducting and reporting case study research in software engineering,” *Empirical software engineering*, vol. 14, no. 2, p. 131, 2009.

[71] E. Fernandes, J. Oliveira, G. Vale, T. Paiva, and 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*, pp. 1–12, 2016.

Appendix A

Literature Sources

As the result of the multivocal literature review, 17 academic studies and 18 gray literature sources are shared with the corresponding smells in Table A.1.

Table A.1: 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 [25]	White	✓			✓			
Four eyes are better than two: On the impact of code reviews on software quality [28]	White	✓						
Review participation in modern code review [26]	White	✓						
Investigating code review quality: Do people and participation matter? [27]	White	✓						
Individual, social and personnel factors influencing modern code review process [34]	White		✓		✓			
Was my contribution fairly reviewed? [35]	White		✓					
Process aspects and social dynamics of contemporary code review [36]	White		✓					
Code reviewing in the trenches: Challenges and best practices [2]	White			✓		✓	✓	✓
Confusion in code reviews: Reasons, impacts, and coping strategies [38]	White			✓			✓	✓
Pushback: Characterizing and detecting negative interpersonal interactions in code review [39]	White			✓				
A study of the quality-impacting practices of modern code review at Sony mobile [42]	White				✓			✓
Code reviews do not find bugs [44]	White					✓		
Software engineering artifact in software development process-linkage between issues and code review processes [47]	White						✓	
What makes a code change easier to review: an empirical investigation on code change reviewability [48]	White						✓	
Small patches get in! [54]	White							✓
The influence of non-technical factors on code review [55]	White							✓
QT Review Policy [29]	Gray	✓	✓		✓		✓	
[vtk-developers] No self reviews on Gerrit please (e-mail) [30]	Gray	✓						
Why code reviews matter [31]	Gray	✓						
7 Code Review Best Practices and Dynamics to Implement [32]	Gray	✓						
10 Faulty Behaviors of Code Review [33]	Gray	✓	✓			✓	✓	✓
Introducing Code Review and Collaboration [37]	Gray		✓	✓				
How We Do Code Review [40]	Gray			✓	✓	✓		✓
Code Review Guidelines-GitLab [41]	Gray			✓				
Google's Engineering Practices Documentation [43]	Gray				✓	✓	✓	✓
7 Code Review Best Practices And Dynamics [45]	Gray					✓		
Contribution Guidelines-LibreOffice [46]	Gray					✓	✓	
Linus Torvalds: 'I Do No Coding Any More' [49]	Gray						✓	
Code Review Best Practices-Palantir [50]	Gray						✓	✓
How to Make Good Code Reviews Better [51]	Gray						✓	
Pull request best practices [52]	Gray						✓	
Contribution Guidelines-Wireshark [53]	Gray						✓	
What is Code Review? [56]	Gray							✓
Code Reviews at Google [57]	Gray							✓

Appendix B

Code and Reproducibility

The data used in the experiments is shared online at Figshare [1].

Questions of the developer survey and the source code of this study are openly available at the GitHub repository [2].

[1]`https://figshare.com/s/a7691f88aa67dc4bd828`
[2]`https://github.com/emredogan7/code-review-smells`