Noname manuscript No.
(will be inserted by the editor)

Code Smells Detection via Modern Code Review: A Study of the OpenStack and Qt Communities

Xiaofeng Han · Amjed Tahir · Peng Liang · Steve Counsell · Kelly Blincoe · Bing Li · Yajing Luo

Received: date / Accepted: date

Abstract Code review plays an important role in software quality control. A typical review process involves a careful check of a piece of code in an attempt to detect and locate defects and other quality issues/violations. One type of issue that may impact the quality of software is code smells - i.e., bad coding practices that may lead to defects or maintenance issues. Yet, little is known about the extent to which code smells are identified during modern code review. To investigate the concept behind code smells identified in modern code review and what actions reviewers suggest and developers take in response to the identified smells, we conducted an empirical study of code smells in code reviews by analyzing reviews from four large open source projects from the OpenStack (Nova and Neutron) and Qt (Qt Base and Qt Creator) communities. We manually checked a total of 25,415 code review comments obtained by keywords search and random selection; this resulted in the identification of 1,539 smell-related reviews which then allowed the study of the causes of code smells, actions taken against identified smells, time taken to fix identified smells, and reasons why developers ignored fixing identified smells. Our analysis found that 1) code smells were not commonly identified in code reviews, 2) smells were usually caused by *violation of coding conventions*, 3) reviewers usually provided constructive feedback, including fixing (refactoring) recommen-

Xiaofeng Han · Peng Liang (✉) · Bing Li · Yajing Luo
School of Computer Science, Wuhan University, Wuhan, China
Hubei Luojia Laboratory, Wuhan, China
E-mail: {hanxiaofeng, liangp, bingli, luoyajing}@whu.edu.cn

Amjed Tahir
School of Mathematical and Computational Sciences, Massey University, Palmerston North, New Zealand
E-mail: a.tahir@massey.ac.nz

Steve Counsell
Department of Computer Science, Brunel University London, London, United Kingdom
E-mail: steve.counsell@brunel.ac.uk

Kelly Blincoe
Department of Electrical, Computer, and Software Engineering, University of Auckland, Auckland, New Zealand
E-mail: k.blincoe@auckland.ac.nz

dations to help developers remove smells, 4) developers generally followed those recommendations and actioned the changes, 5) once identified by reviewers, it usually takes developers less than one week to fix the smells, and 6) the main reason why developers chose to ignore the identified smells is that it is *not worth fixing the smell*. Our results suggest the following: 1) developers should closely follow coding conventions in their projects to avoid introducing code smells, 2) *review-based* detection of code smells is perceived to be a trustworthy approach by developers, mainly because reviews are context-sensitive (as reviewers are more aware of the context of the code given that they are part of the project's development team), and 3) program context needs to be fully considered in order to make a decision of whether to fix the identified code smell immediately.

Keywords Modern Code Review · Code Smell · Mining Software Repositories · Empirical Study

1 Introduction

Code smells are defined as symptoms of possible code or design problems (Fowler, 1999), which may potentially have a negative impact on software quality, such as maintainability (Palomba et al., 2018), code readability (Abbes et al., 2011), testability (Tahir et al., 2016), and defect-proneness (Khomh et al., 2009).

A large number of studies have focused on smell detection and removal techniques (Moha et al., 2009; Tsantalis and Chatzigeorgiou, 2009). There are also a number of open source (and widely used in industrial settings) static analysis tools for smell detection; these include tools such as PMD[1], SonarQube[2], and Designite[3]. Those tools are general purpose and use a threshold approach for identifying certain smells. However, previous work (Tahir et al., 2020; Yamashita and Moonen, 2013) has shown that the program context and domain are important in identifying smells. This may also include other factors like developer experience and previous involvement in the project, which makes it difficult for program analysis tools to correctly identify smells since this information is rarely taken into account. Existing smell detection tools are also known to produce false positives (Fontana et al., 2016; Sharma and Spinellis, 2018). Therefore, manual detection of smells could be considered more valuable than current automatic approaches.

Code review is a process which aims to verify the quality of software by detecting defects and other issues in the code and to ensure that the code is readable, understandable and maintainable before they are merged into the code base. It has been linked to improved quality (Baker Jr, 1997), reduced defects (McIntosh et al., 2016), reduced anti-patterns (Morales et al., 2015) and the identification of vulnerabilities (Meneely et al., 2014). Compared to smell detection tools, code reviews are usually performed by developers belonging to the same project (McConnell, 2004), so it is possible that reviewers will take full account of program context and thus better identify code smells. In modern code review (MCR), code changes are reviewed through some code review platforms, such as Gerrit[4]. There is a common

[1] `https://pmd.github.io`
[2] `https://www.sonarqube.org`
[3] `https://www.designite-tools.com`
[4] `https://www.gerritcodereview.com`

practice of keeping the changes as small as possible to facilitate the review. But it is still important to see if reviewers identify any code smells in the changes being added to the code base. We are interested in this human-driven detection of code smells since we want to see how different this is from automatic detection of code smells given that context matters. However, little is known about the extent to which code smells are identified during modern code review and whether developers (the code authors) take any action when a piece of code is deemed "smelly" by reviewers.

Therefore, we set out to study the concept behind code smells identified in MCR and track down actions taken after reviews were carried out. To this end, we mined code review discussions from four most active projects from the OpenStack[5] community (Nova[6] and Neutron[7]) and the Qt[8] community (Qt Base[9] and Qt Creator[10]), which use Gerrit as their code review platform. We then conducted a comprehensive quantitative and qualitative analysis to study how common it was for reviewers to identify code smells during code review, why the code smells were introduced, what actions they recommended for those smells, how long it took developers to change the code based on those recommendations and why developers ignored some of the identified smells. In total, we analysed 1,539 smell-related code reviews obtained by manually checking 25,415 review comments to achieve our goal. Our results suggest that:

1. Code smells are not widely identified in modern code review.
2. Following coding conventions can help reduce the introduction of code smells.
3. Reviewers usually provide useful suggestions to help developers better fix the identified smells, while developers commonly accept reviewer recommendations regarding the identified smells and tend to refactor their code based on those recommendations.
4. Review-based detection of code smells is seen as a trustworthy mechanism by developers.
5. Program context needs to be taken into full account to determine whether to fix the identified code smells immediately.

In this paper, we extended our earlier work on studying code smells in code reviews (Han et al., 2021) through the following additions:

1. We extended our dataset by including the code review data from two large projects of the Qt community.
2. We explored specific refactoring actions suggested by reviewers.
3. We investigated two additional research questions (RQ4 and RQ5 in Section 3.1) discussing the resolution time of smells and also the reasons why developers chose to ignore the identified smells.

The paper is structured as follows: related work is presented in Section 2. The study design and data extraction methods are then explained in Section 3 and

[5] https://www.openstack.org
[6] https://wiki.openstack.org/wiki/Nova
[7] https://wiki.openstack.org/wiki/Neutron
[8] https://www.qt.io/
[9] https://github.com/qt/qtbase
[10] https://www.qt.io/product/development-tools

the results are presented in Section 4; this is followed by a discussion in Section 5 before threats to the validity in Section 6; finally, conclusions and future work in Section 7.

2 Related Work

2.1 Studies on Code Smells

A growing number of studies have investigated the impact of code smells on software quality, including defects (Hall et al., 2014; Khomh et al., 2009), maintenance (Sjøberg et al., 2013) and program comprehension (Abbes et al., 2011). Other studies have looked at the impact of code smells on software quality using a group of developers working on a specific project (Palomba et al., 2015; Sjøberg et al., 2013; Soh et al., 2016).

Tufano et al. (2015) mined version histories of 200 open source projects to study when code smells were introduced and the main reason behind their interaction. It was found that smells appeared in general as a result of maintenance and evolution activities. Sjøberg et al. (2013) investigated the relationship between the presence of code smells and maintenance effort through a set of control experiments. Their study did not find significant evidence that the presence of smells led to increased maintenance effort. Previous studies also include work investigating the impact of different forms of smells on software quality, such as architectural smells (Garcia et al., 2009; Martini et al., 2018), test smells (Bavota et al., 2015; Tahir et al., 2016) and spreadsheet smells (Dou et al., 2014).

A number of previous studies have investigated developer perception of code smells and their impact in practice. A survey on developer perception of code smells conducted by Palomba et al. (2014) found that developer experience and system knowledge are critical factors in the identification of code smells. Yamashita and Moonen (2013) reported that developers are moderately concerned about code smells in their code. A recent study by Taibi et al. (2017) replicated the two previous studies (Palomba et al., 2014; Yamashita and Moonen, 2013) and found that the majority of developers always considered smells to be harmful; however, it was found that developers perceived smells as critical in theory, but not as much in practice. Tahir et al. (2020) mined posts from Stack Exchange sites to explore how the topics of code smells and anti-patterns were discussed amongst developers. Their study found that developers widely used online forums to ask for general assessments of code smells or anti-patterns instead of asking for particular refactoring solutions.

2.2 Code Reviews in Software Development

Code review is an integral part in modern software development. In recent years, empirical studies on code reviews have investigated the potential code review factors that affect software quality. For example, McIntosh et al. (2014) investigated the impact of code review coverage and participation on software quality in the Qt, VTK, and ITK projects. The authors used the incidence rates of post-release defects as an indicator and found that poorly reviewed code (e.g., with low review

coverage and participation) had a negative impact on software quality. Uchôa et al. (2021) investigated whether and how technical (e.g., number of times a file has been changed and types of change) and social (e.g., number of prior code changes submitted by the code owner and centrality of the code owner on the collaboration graph) metrics can be used to predict design impactful changes by analyzing more than 50k code reviews of seven real-world systems.

Some studies have focused on the impact of code review on software quality. A study by Kemerer and Paulk (2009) investigated the impact of review rate on software quality. The authors found that the *Personal Software Process* review rate was a significant factor affecting defect removal effectiveness, even after accounting for developer ability and other significant process variables. Several studies (Kononenko et al., 2015; McIntosh et al., 2014, 2016) have investigated the impact of modern code review on software quality. Other studies have also investigated the impact of code reviews on different aspects of software quality, such as vulnerabilities (Bosu et al., 2014), design decisions (Zanaty et al., 2018), anti-patterns (Morales et al., 2015) and code smells (Nanthaamornphong and Chaisutanon, 2016; Pascarella et al., 2020).

Many recent studies of code review are based on pull requests (PR). Wessel et al. (2020) conducted an empirical study on the effects of adopting bots to support the code review process on pull requests. They found that the adoption of code review bots increased the monthly number of merged pull requests with less communication between maintainers and contributors and lead projects to reject fewer pull requests. Coelho et al. (2021) investigated technical aspects characterizing refactoring-inducing PRs based on data mined from GitHub and refactorings detected by RefactoringMiner. They found that PRs that induced refactoring edits have different characteristics from those that do not. Besides, their qualitative analysis indicates that at least one refactoring edit was induced by code review in more than half of refactoring-inducing PRs they studied. Cassee et al. (2020) present an exploratory empirical study investigating the effects of Continuous Integration (CI) on open source code reviews. They found that the number of comments per code review decreases after the adoption of CI, while the number of changes made during a code review remains constant.

Panichella and Zaugg (2020) investigated the approaches and tools that are needed to facilitate code review activities from a developer point of view. They found that developers performed additional activities or tasks (e.g., the need to fix licensing and security issues) during code review with the availability of new emerging development technologies and practices, thus additional types of feedback and novel approaches and tools (e.g., for automatically detecting and fixing documentation issues) are wanted by developers. However, code review is basically a human task involving technical, personal and social aspects. Chouchen et al. (2021) present the concept of Modern Code Review Anti-patterns (MCRA) and identify five common MCR anti-patterns. They conducted a study by analyzing 100 reviews randomly selected from the OpenStack project. Their preliminary results show that these anti-patterns are indeed prevalent in MCR, affecting 67% of code reviews.

Nanthaamornphong and Chaisutanon (2016) examined review comments from code reviewers and described the need for an empirical analysis of the relationship between code smells and peer code review. Their preliminary analysis of review comments from OpenStack and WikiMedia projects indicated that code review

processes identified a number of code smells. However, the study only provided preliminary results and did not investigate the causes of, or resolution strategies for, these smells. A more recent study by Pascarella et al. (2020) found that code reviews helped in reducing the severity of code smells in source code, but this was mainly a side effect to other changes unrelated to the smells themselves.

3 Methodology

We detail the methodology followed in this study in this section. We explain the methods used to collect, analyse and report our results for the research questions answered in this study.

3.1 Research Questions

The main goal of this study is to investigate how code smells are addressed during the course of a modern code review process. Specifically, we analyse code reviews for the purpose of understanding the nature of code smells in code reviews from a code reviewer and developer point of view. This goal is decomposed into the following five research questions (RQs):

RQ1: Which code smells are the most frequently identified by code reviewers?
Rationale: This question aims to find out the frequency with which code smells are identified by code reviewers and what particular code smells are repeatedly detected/reported by reviewers. Such information can help in improving developers' awareness of these frequently identified code smells and also help tool designers to focus on smells of more interest to developers.

RQ2: What are the common causes for code smells that are identified during code reviews?
Rationale: This question investigates the main reasons behind the identified smells as explained by the reviewers or developers. Previous research has shown that context is important in identifying code smells (Sae-Lim et al., 2018; Tahir et al., 2020). When conducting a review, reviewers can express and explain why they think the code under review may contain a smell. Developers can also reply to reviewers and explain what they think of the smell(s), and, if they agree with the reviewers' assessment, how they introduce the smell(s). Understanding the common causes of code smells identified manually by reviewers will shed some light on the effectiveness of manual detection of smells and help developers better understand the nature of identified smells and context in which those smells are being labelled.

RQ3: How do reviewers and developers treat the identified code smells?
Rationale: This question investigates the actions suggested by reviewers and those taken by developers on the identified smells. When a smell is identified, reviewers can provide suggestions to resolve the smell and developers can then decide on whether to fix or ignore the code with the smell. In addition to this, we also

investigate the concrete refactoring actions (e.g., move/extract method) suggested by reviewers. This question is further divided into three sub-questions (from the perspective of the reviewer, developer and the relationship between their actions):

RQ3.1: What actions do reviewers *suggest* to deal with the identified smells?

RQ3.2: What actions do developers *take* to resolve the identified smells?

RQ3.3: What is the *relationship* between the actions suggested by reviewers and those taken by developers?

RQ4: How long does it take to resolve code smells by developers after they have been identified by reviewers?

Rationale: With this question, we want to investigate the influence of different code smell categories on the fix time. In addition, combined with the results of RQ3.1, we also want to know the influence of reviewers' suggestions on the fix time of code smells. This can help in understanding the nature of each of those smells, and how difficult (using time as an indicator of difficulty) it can be to implement such fixes.

RQ5: What are the common causes for not resolving code smells that have been identified in code?

Rationale: In the case where code smells are not resolved, technical debt is introduced by developers. Consequently, with this question, we want to know what makes the developers choose to ignore the smells. Understanding this can further help to remove smells and pay back technical debt.

3.2 Research Setting

We conducted our study using the projects from two large and active open-source communities: OpenStack and Qt. OpenStack is a set of software tools for building and managing cloud computing platforms. It is considered one of the largest open source communities. OpenStack projects contain around 13 million lines of code, contributed to by around 12 thousand developers[11]. Qt is an open source cross-platform application and UI framework developed by the Digia operation. Contributions form different large communities are also welcomed by Qt.

We deemed these two communities to be appropriate for our study, since they are large in size and both have long investment in their code review process. We then selected two of the most active projects (based on the number of patch submissions (Hirao et al., 2020)) in each of those communities as our subject projects: Nova (a fabric controller) and Neutron (a network connectivity platform) from the OpenStack Community and Qt Base (core UI functionality) and Qt Creator (Qt IDE) from the Qt community.

The two OpenStack projects (Nova and Neutron) are mainly written in Python, while the Qt projects (Qt Base and Qt Creator) are mostly written in C++. All these projects use Gerrit[12], a web-based modern code review platform built on top of Git. The Gerrit review workflow is explained next.

[11] As of March 2022: `https://www.openhub.net/p/openstack`

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

Gerrit is designed for modern code review and provides a detailed code review workflow. First, a developer (author) makes a change to the code and submits the code (patch) to the Gerrit server so that it can be reviewed. Then, verification bots check the code using static analysers and run automated tests. A reviewer (usually other developers that have not been involved in writing the code under review) will then conduct a formal review of the code and provide comments. The original author can reply to the reviewer's comments and action the required changes by producing a new revision of the patch. This process is repeated until the change is merged to the code base or finally abandoned.

3.3 Mining Code Review Repositories

Fig. 1 outlines our data extraction and mining process. We mined the code review data via the RESTful API provided by Gerrit, which returns the results in a JSON format. We used a Python script to automatically mine the review data in the studied period and store the data in a local database. Details of the four projects are shown in Table 1.

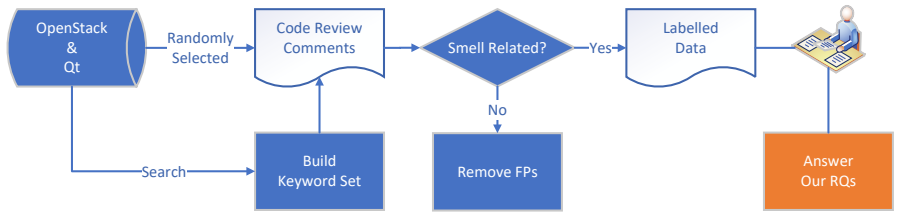

Fig. 1: An overview of our data mining and analyzing process

Table 1: An overview of the subject projects

Project	Review Period	#Code Changes	#Comments
Nova	Jan 2014 - Dec 2018	22,762	156,882
Neutron		15,256	152,429
Qt Base		27,340	104,272
Qt Creator		28,229	79,087
Total		93,587	492,670

In total, we mined 93,587 code changes and 492,670 code review comments between Jan. 2014 and Dec. 2018 from the four projects. Upon checking the data, we found that there are comments published by bots in Qt Base and Qt Creator projects. Since our goal is to investigate manual detection of code smells in code review, we decided to remove all bots generated comments (51,837 comments in total). By doing so, we ended up with a total of 93,587 code changes and 440,833

review comments (309,311 from the OpenStack community and 131,522 from the Qt community) for further analysis.

3.4 Building the Keyword Set

To locate code review comments that include code smell discussions, we used several variations of terms referring to code smells or anti-patterns, including "code smell", "bad smell", "bad pattern", "anti-pattern" and "technical debt". In addition, considering that reviewers may point out a specific code smell by its name (e.g., dead code) rather than using general terms, we also included a list of specific code smell terms obtained from Tahir et al. (2018), that extracted these smell terms from several relevant studies on this topic, including the first work on code smells by Fowler (1999) and the systematic review by Zhang et al. (2011). The list of these specific smell terms used in our study are shown in Table 2.

Table 2: Specific code smell terms included in our data mining process

Specific Code Smell Terms			
Accidental Complexity	Anti Singleton	Bad Naming	Blob Class
Circular Dependency	Coding by Exception	Complex Class	Complex Conditionals
Data Class	Data Clumps	Dead Code	Divergent Change
Duplicated Code	Error Hiding	Feature Envy	Functional Decomposition
God Class	God Method	Inappropriate Intimacy	Incomplete Library Class
ISP Violation	Large Class	Lazy Class	Long Method
Long Parameter List	Message Chain	Middle Man	Misplaced Class
Parallel Inheritance Hierarchies	Primitive Obsession	Refused Bequest	Shotgun Surgery
Similar Subclasses	Softcode	Spaghetti Code	Speculative Generality
Suboptimal Information Hiding	Swiss Army Knife	Temporary Field	Use Deprecated Components

Since the effectiveness of the keyword-based mining approach relies on the set of keywords that are used in the search, we followed the systematic approach used by Bosu et al. (2014) to identify the keywords included in our search. This includes the following steps[13]:

1. Build an initial keyword set (as described above).
2. Build a corpus by searching for review comments that contain at least one keyword of our initial keyword set (e.g., "dead" or "duplicated") in the code review data we collected in Section 3.3.
3. Process the identified review comments which contain at least one keyword of our initial keyword set and then apply the identifier splitting rules (i.e., "isDone" becomes "is Done" or "is_done" becomes "is done").
4. Create a list of tokens for each document in the corpus.
5. Clean the corpus by removing stopwords, punctuation and numbers and then convert all the words to lowercase.
6. Apply the Porter stemming algorithm (Porter, 2001) to obtain the stem of each token.
7. Create a Document-Term matrix (Tan et al., 2016) from the corpus.
8. Find the additional words that co-occurred frequently with each of our initial keywords (co-occurrence probability of 0.05 in the same document).

[13] implemented using the NLTK package: `https://www.nltk.org`

After performing these eight steps, we found that no additional keywords co-occurred with each of our initial keywords, based on the co-occurrence probability of 0.05 in the same document. Therefore, we believe that our initial keyword set is sufficient to support the keyword-based mining method. The initial set of keywords (which is the same as the final set of keywords) is shown in Table 3.

Table 3: The initial set of keywords included in our data mining process

Code Smell Term	**Keywords**
Code Smell	smell, smelly
Bad Smell	bad, smell, smelly
Anti-Pattern	anti, pattern, bad
Bad Pattern	bad, pattern
Technical Debt	technical, debt
Accidental Complexity	accidental, complexity, complex
Anti Singleton	anti, singleton
Bad Naming	bad, naming
Blob Class	blob
Circular Dependency	circular, circularity, dependency, dependent
Coding by Exception	exception
Complex Class	complex, complexity
Complex Conditionals	complex, complexity, conditional, condition
Data Class	data class
Data Clumps	clump
Dead Code	dead, death, unused, useless
Divergent Change	divergent, divergence
Duplicated Code	duplicated, duplicate, duplication, clone
Error Hiding	hiding, hide
Feature Envy	envy
Functional Decomposition	decompose, decomposition
God Class	god, brain
God Method	god, brain
Inappropriate Intimacy	inappropriate, intimacy
Incomplete Library Class	incomplete, library
ISP Violation	ISP, violate, violation
Large Class	large, big
Lazy Class	lazy
Long Method	long
Long Parameter List	long, parameter list
Message Chain	chain
Middle Man	middle
Misplaced Class	misplace, misplaced
Parallel Inheritance Hierarchies	parallel, inheritance
Primitive Obsession	obsession
Refused Bequest	refuse, refused, bequest
Shotgun Surgery	shotgun, surgery
Similar Subclasses	similar, subclass
Softcode	softcode
Spaghetti Code	spaghetti
Speculative Generality	speculative, generality
Suboptimal Information Hiding	suboptimal, hiding, hide
Swiss Army Knife	swiss, army, knife
Temporary Field	temporary, temporal
Use Deprecated Components	deprecated, deprecate, component

3.5 Identifying Smell-related Reviews in Keywords-searched Review Comments

We identified smell-related code reviews in four steps, as follows:

In **step one**, we developed a script to search for review comments that contained at least one of the keywords identified in Section 3.4. The search returned a total of 23,292 review comments from the four projects.

In **step two**, two of the authors independently and manually checked the review comments obtained in **step one** without considering any other information to exclude comments that were *clearly* unrelated to code smells. When both coders decided a review comment was *clearly* not related to code smells, we excluded it from any future analysis. The Cohen's Kappa coefficient value (Cohen, 1960) is 0.83, which indicates a near perfect agreement between the two coders. The number of votes is shown in Table 4. As a result of this step, the number of remaining review comments became 4,761.

To illustrate this process, consider the following two review comments that contain the keyword "dead". In the first example, the reviewer commented that: "*why not to put the port on dead vlan first?*"[14]. Although this comment contains the keyword "dead", both coders agreed that it was unrelated to code smells and the comment was therefore excluded. In the second example, the reviewer commented: "*remove dead code*"[15], which was regarded as related to *dead code* by the two coders and was included in the analysis.

In **step three**, the same two coders worked together to manually analyze the remaining (4,761) review comments using the related information of each review comment, including the code review discussions and associated source code to determine whether the code reviewers identified any smells in the review comments. We considered a comment to be related to code smells only when both coders agreed. The agreement between the two coders was calculated using the Cohen's Kappa coefficient (Cohen, 1960), which is 0.84 (almost perfect agreement). The number of votes for TT (True-True), TF (True-False), FT (False-True), FF (False-False) from the two coders is shown in Table 4. When the coders were unsure or disagreed about the outcomes, a third author was then involved in the discussion until a consensus was reached. This resulted in a reduction in the number of review comments to 1,592.

Table 4: The number of votes for TT, TF, FT, FF from the two coders in step two and three

Step two			**Step three**		
Coder 2 \ **Coder 1**	**Maybe**	**No**	**Coder 2** \ **Coder 1**	**Yes**	**No**
Maybe	3576	482	**Yes**	1435	151
No	703	18531	**No**	187	2988

[14] `http://alturl.com/gqn7u`
[15] `http://alturl.com/2kcko`

To better explain our selection process, consider the two examples in Fig. 2. In the top example[16], the reviewer suggested adding another argument to a method to eliminate code duplication. Then the developer replied: "*Done*", which implies an acknowledgment of the code duplication. We considered this as a clear smell-related review and the review comment was retained for further analysis. In contrast, in the bottom example[17], we observed that the comment was just used to explain the meaning of the "DRY" principle, but did not indicate that the code contained duplication according to the context. Thus, this comment was excluded from analysis.

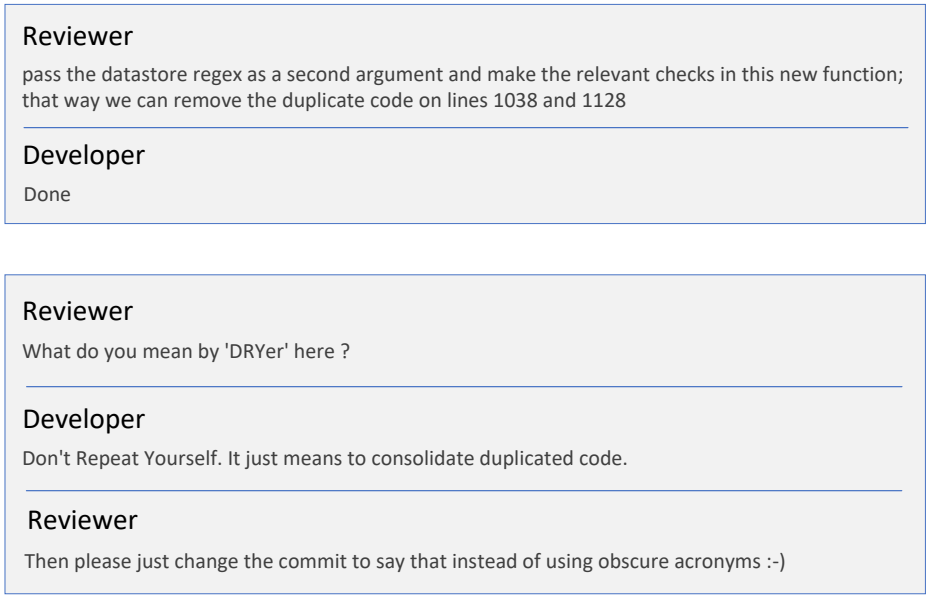

Fig. 2: Review comments related to *duplicated code*: the top review is smell-related, while the bottom one is not

Finally, in **step four**, we recorded the related information of each review comment in an external text file for further analysis, which contained: 1) a URL to the code change, 2) the type of the identified code smell, 3) the discussion between reviewers and developers and 4) a URL to the source code. We ended up with a total of 1,502 smell-related reviews (we note that several review comments appearing in the same discussion were merged). An example of an extracted source file is shown below:

[16] `http://alturl.com/4s775`
[17] `http://alturl.com/786zn`

Code Change URL: `http://alturl.com/2ne85`
Code Smell: *Dead Code*
Code Smell Discussions:
1) Reviewer: "Looks like copy-paste of above and, more importantly, dead code."
2) Developer: "yes, sorry for that."
Source Code URL: `http://alturl.com/yai68`

3.6 Identifying Smell-related Reviews in Randomly-selected Review Comments

Knowing that reviewers and developers may not use the same keywords as we used in Section 3.5 when detecting and discussing code smells during code review, we supplemented our keyword-based mining approach by including a randomly selected set of review comments from the rest of the review comments (291,229 in the OpenStack projects and 126,312 in the Qt projects) that did not contain any of the keywords used in Section 3.4. Based on 95% confidence level and 3% margin of error (Israel, 1992), we ended up with an additional 1,064 review comments from the OpenStack projects and 1,059 review comments from the Qt projects. We then followed the same process of manual analysis (i.e., from **step two** to **step four** as described in Section 3.5) to identify smell-related reviews in these randomly selected review comments. Finally, we identified a total of 37 smell-related reviews from the randomly selected review comments.

In addition to the 1,502 smell-related reviews obtained by keywords search in Section 3.5, we finally obtained a total of 1,539 smell-related reviews for further analysis. Fig. 3 shows the size (in the form of inserted and deleted lines) of code changes related to the identified smell-related reviews. For inserted lines, 55% of code changes have no more than 200 lines of newly added code. Only 21% of the code changes have more than 500 insertions. As for deleted lines, 92% of code changes only deleted no more than 200 lines. This suggests that the size of code changes was relatively small, generally.

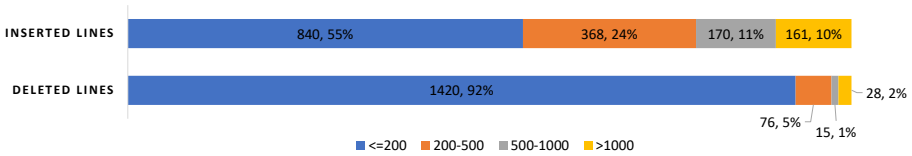

Fig. 3: The size of code changes related to the identified smell-related reviews

3.7 Manual Analysis and Classification

3.7.1 RQ1: Which code smells are the most frequently identified by code reviewers?

In Sections 3.5 and 3.6, we explained how we identified and recorded the smell type noted in each review when analyzing the review comments. When a reviewer used general terms (such as "smelly" or "anti-pattern") to describe the identified smell, we classified the type in these reviews as "general". The others were classified as specific smells, based on the keyword included and the description provided (e.g., *duplicated code*).

3.7.2 RQ2: What are the common causes for code smells that are identified during code reviews?

For RQ2, we adopted Thematic Analysis (Braun and Clarke, 2006) to find the causes for the identified code smells in Sections 3.5 and 3.6. We used MAXQDA[18] - a software package for qualitative research - to code the related information of the identified code smells. Firstly, we coded the collected smell-related reviews by highlighting sections of the text related to the causes of the code smell in the review. When no cause was found, we used "cause not provided/unknown". Next, we looked over all the codes that we created to identify common patterns and generated themes. We then reviewed the generated themes by returning to the dataset and comparing our themes against it. Finally, we named and defined each theme. We also undertook further analysis from the perspective of smell types, to investigate the main causes leading to the introduction of a specific smell type.

This process was performed by the same two coders as in Section 3.5 and Section 3.6. A third author was involved in cases of disagreement by the two coders.

3.7.3 RQ3: How do reviewers and developers treat the identified code smells?

For RQ3, we manually checked the code reviews obtained by following the process described in Section 3.5 and Section 3.6 to identify the actions suggested by reviewers and taken by developers.

For RQ3.1, we placed the actions recommended by reviewers into three categories, as proposed in Tahir et al. (2018):

1. **Fix**: recommendations are made to refactor the code smells.
2. **Capture**: detecting that there may be a code smell, but no direct refactoring recommendations are given.
3. **Ignore**: recommendations are to ignore the identified smells.

When reviewers provided *fix* actions, we considered this as a refactoring action. We then further investigated the concrete refactoring actions provided by reviewers based on the classification in Fowler (1999), which provides 7 categories with 72 specific refactorings. For example, when you have a code fragment that can be grouped together, you can use **Extract Method** (i.e., move this code to a separate new method (or function) and replace the old code with a call to the method).

[18] `https://www.maxqda.com/`

Another example is related to generalization. If you have two classes with similar features, you can use **Extract Superclass** to refactor your code. That is, create a superclass and move the common features to the superclass. When no specific refactoring action could be extracted from one review, we chose to exclude it from our analysis.

For RQ3.2, we investigated how developers responded to reviewers that identified code smells in their code. We conducted this analysis following a three step approach: We first checked the developer's response to the reviewer in the discussion (Gerrit provides a discussion platform for both reviewers and developers). Second, we investigated the associated source code file(s) of the patch before the review was conducted and the changes in the source code made after the review. Finally, if the developers neither responded to the reviewers nor modified the source code, we then checked the status of the corresponding code change (i.e., merged or abandoned).

We considered the identified code smells to be solved in these two cases: 1) changes were made in the source code file(s) and 2) the corresponding code change was finally abandoned (i.e., when the code change was abandoned, the code change was not be merged into the code base. In other words, the code smell no longer existed).

There were cases where developers would not fix the smell immediately and said that they would fix the identified smell in the future (i.e., in a later code change). In such a case, it is difficult to judge whether the identified smell was finally fixed. Therefore, we regarded this situation as *unknown*.

For RQ3.3, based on the results of RQ3.1 and RQ3.2, we categorized the relationship between the actions recommended by reviewers and those taken by developers into the following three categories:

1. A developer *agreed* with the reviewer's recommendations.
2. A developer *disagreed* with the reviewer's recommendations, or
3. A developer *did not respond* to the reviewer's comments.

These three categories were then mapped into three actions:

1. **Fixed the smell**: Refactoring was done and the smell was successfully removed.
2. **Ignored the smell**: No changes were performed to the source code and the smell was finally ignored.
3. **Unknown**: Explained above.

In the below example, the reviewer just pointed out an instance of a *dead code* smell. We categorised the suggestion of the reviewer as "Capture". Subsequently, the developer replied "Done" to the reviewer, which meant that the developer had resolved the smell. We thought that, in such a case, the developer had *agreed* with the reviewer's recommendation and fixed the smell.

Link: `http://alturl.com/7u8yq`
Reviewer: "This is dead code since you're overwriting it next."
Developer: "Done"

3.7.4 RQ4: How long does it take to resolve code smells by developers after they have been identified by reviewers?

When an identified code smell was fixed by developers, we checked the time taken for the fix. We extracted two types of time information related to the identified code smells:

1. **Identification Time**: we regarded the time when the reviewer published the smell-related review comment as the identification time of the smell.
2. **Resolution Time**: the time taken until a smell is resolved/refactored. This time can be divided into two categories:
   (a) when the smell was fixed in a later patch, we regarded *the time of uploading the patch* as the resolution time.
   (b) when the smell was not fixed in a later patch, but the code change that contained the smell was later abandoned, we regarded *the time of abandoning the change* as the resolution time.

To provide a better understanding of these two types, Fig. 4 shows an example from the Qt Creator project[19]. This example shows a comparison of source code between two patches: patch 3 (left) and patch 4 (right). In this example, the reviewer identified duplication (i.e., *duplicated code*). We regarded the published time of this comment (i.e., Jul 28, 2014, 17:47:32 UTC+08:00) as the identification time of this smell. In patch 4, we see that the developer made a change as the reviewer suggested, implying that the smell was fixed. So we considered the upload time of patch 4 (i.e., Jul 28, 2014, 18:42:38 UTC+08:00) as the resolution time of this smell. The interval between these two time points (55 minutes and 6 seconds) is seen as the time taken to fix the code smell.

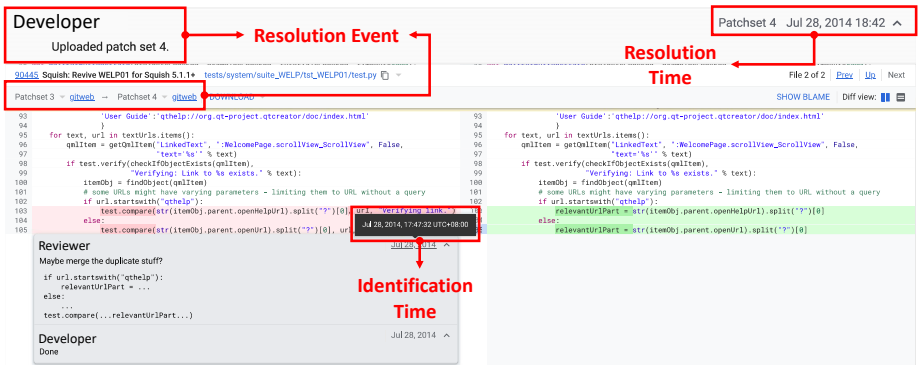

Fig. 4: Example for the identification time and resolution time of a smell

In four reviews, the time interval between the resolution time and the identification time is very long, exceeding one year. In this case, the code change where the identified smell was located took a long time to be abandoned. We believe

[19] `http://alturl.com/b2dkt`

that this case is abnormal (i.e., outliers) and including these reviews would have affected our results; consequently we excluded these four reviews.

There are also two reviews in which the resolution time of the identified smells could not be determined. Fig. 5 shows an example from the Nova project[20]. In this example, the developer replied to the reviewer that they had removed most of the duplicate code (i.e., the smell was fixed). However, we could not find out in which code change this code smell was fixed. That is, we could not get the resolution time of this smell; consequently, we excluded this review from our analysis.

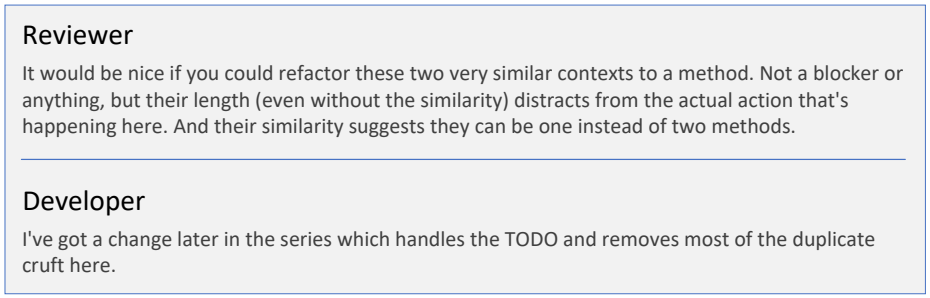

Fig. 5: An example of a smell where the resolution time cannot be found

There are also two reviews in which the reviewer added the smell-related comment after the code change had been abandoned. In this case, the identification time was earlier than the resolution time, which is not suitable for our analysis and consequently we excluded these two reviews too.

Finally, we decided to keep only the smell categories that had over 100 instances. Some code smell categories (i.e., *long method*, *circular dependency*, *speculative generality*, *swiss army knife* and general smell) were rarely identified and consequently these smell categories may have limited statistical significance. We decided to exclude them (35 reviews) from answering this RQ.

Moreover, to investigate the relationship between the time for fixing smells and reviewer suggestions, we divided the selected reviews into two groups: 1) reviews in which reviewers provided specific refactoring actions and 2) reviews in which reviewers just captured the smells or provided general actions. We then calculated and compared the minimum, quartile, maximum, and median time in these two groups.

3.7.5 RQ5: What are the common causes for not resolving code smells that have been identified in code?

For RQ5, we further investigated the reasons why developers explicitly disagreed with reviewers' assessment of smells (i.e., when developers challenge the reviewers' assessments and then decide to ignore the identified smells). We adopted Thematic Analysis (Braun and Clarke, 2006) to identify the causes of developers ignoring code smells identified by reviewers by studying the content in the reply section of

[20] `http://alturl.com/odfjv`

the review. We followed the same process in this RQ as the one used for answering RQ2. For these cases, we then further checked the final status of the code changes to investigate what happened after developers disagreed with reviewers and decided not to fix the identified code smells.

Note that all of the manual analysis and classification (i.e., identifying smell-related code reviews and their classifications in various aspects) was conducted by at least two authors. A third author was involved in case of disagreement. In total, the manual analysis process took around 65 days (full-time) work of the coders. To facilitate replication, we provide the full data (coded) together with scripts used to collect the dataset in our replication package (Han et al., 2022).

4 Results

In this section, we present the results of our five research questions (RQs). We also provided a replication package online which is complementary for understanding the results and replicating this study (Han et al., 2022).

4.1 RQ1: Which code smells are the most frequently identified by code reviewers?

We show the distribution of code smells identified in the code reviews from the OpenStack projects in Figure 6. In general, we identified 1,184 smell-related reviews in OpenStack. Of all the code smells we identified, *duplicated code* is the most frequently identified smells, with exactly 617 (52%) instances. The smells of *bad naming* and *dead code* are also frequently identified, as they are discussed in 304 (26%) and 218 (18%) code reviews, respectively. There are 30 (2%) code reviews which identified *long method*, while other smells such as *circular dependency* and *swiss army knife* are discussed in only 4 code reviews. The rest of code reviews (11, 1%) use general terms (e.g., code smell) to describe the identified smells, which are called *general smell terms* in this work. Note that the results of distribution of smell-related reviews from the OpenStack projects are slightly different from the results in our previous work (Han et al., 2021) because we found that some code smells were mentioned by developers and were not identified through code review, which were consequently removed from our analysis.

The distribution of code smells identified in the Qt projects is shown in Fig. 6. In general, we identified 355 smell-related reviews in Qt. Unlike OpenStack, *bad naming* is the most frequently identified smell, appearing in 146 (41%) reviews. *Dead code* and *duplicated code* follow closely, identified in 113 (32%) and 92 (26%) reviews, respectively. *Long method*, *circular dependency*, and *speculative generality* are discussed in only 2, 1 and 1 reviews, respectively. Another finding is that no *general smell term* is identified in the code reviews of Qt projects.

RQ1 Summary: According to the percentage of smell-related review comments (less than 1% of total review comments), only a small number of code smells were extracted in the code review data we analysed. Of the identified smells, ***duplicated code***, ***bad naming***, and ***dead code*** are the most frequently identified smells in code reviews.

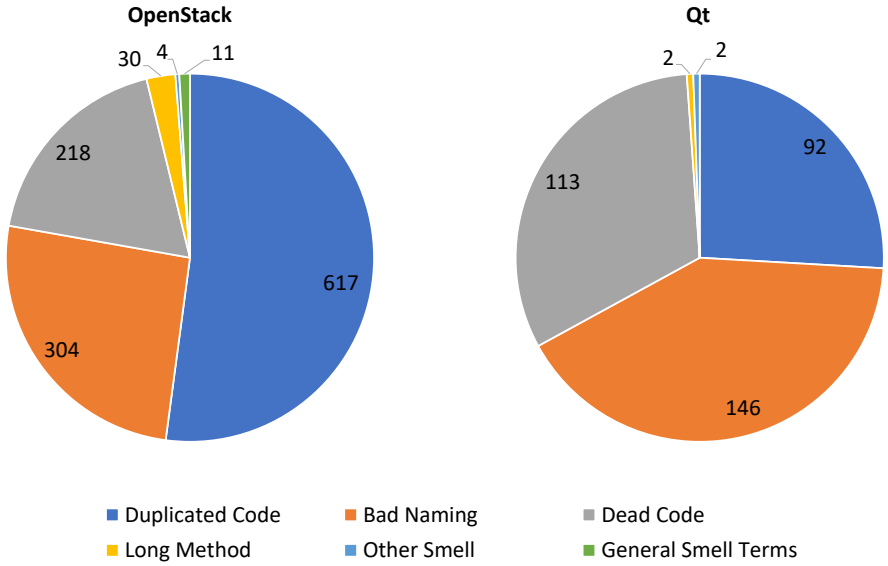

Fig. 6: Distribution of smell-related reviews from the OpenStack and Qt projects

4.2 RQ2: What are the common causes for code smells that are identified during code reviews?

For RQ2, we used Thematic Analysis to identify the common causes for the identified code smells as noted by code reviewers or developers. We then identified four key causes:

- **Violation of coding conventions**: certain violations of coding conventions (e.g., naming convention) are the cause for the smells. (Example: "*moreThanOneIp (CamelCase) is not our naming convention*" [21]).
- **Lack of familiarity with existing code**: developers introduced the smells due to the unfamiliarity with the functionality or structure of the existing code. (Example: "*this useless line because* `None` *will be returned by default*" [22]).
- **Unintentional mistakes of developers**: developers forgot to fix the smells or introduced the smells by mistake. (Example: "*You can see I renamed all of the other test methods and forgot about this one*" [23]).
- **Design choices**: the smells were considered to be caused by the design choice of developers. (Example: "*...If that's the case something is smelly (too coupled)...*" [24]).

We firstly found that the majority of reviews (1,081, 70%) did not provide any explanation for the identified smells - in most cases, the reviewer(s) simply pointed

[21] `http://alturl.com/azijc`
[22] `http://alturl.com/h2bpc`
[23] `http://alturl.com/cisgv`
[24] `http://alturl.com/y2ndw`

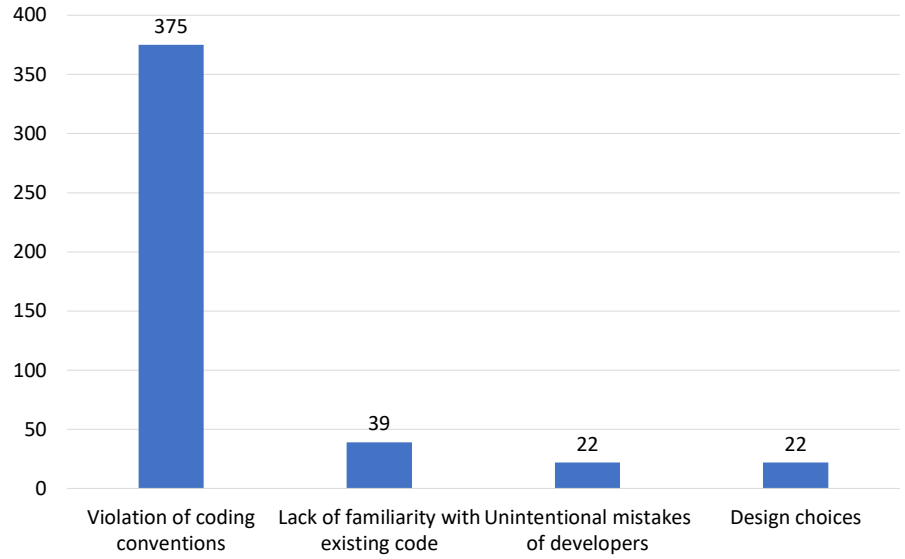

Fig. 7: Causes for the identified smells (we note that there are 1,071 reviews where the reason was not provided)

out the problems, but did not provide any further reasoning for their judgements. The detailed result is shown in Fig. 7.

Of the remaining 458 reviews in which the causes of the smells are provided, 375 (82%) of the reviews indicate that **violation of coding conventions** is the main reason for the smell. For example, a reviewer suggested that the developer should adhere to the naming standard of 'test_[method under test]_[detail of what is being tested]' which indicated a *bad naming* smell, as shown below:

> **Link:** `http://alturl.com/zw5e6`
> **Reviewer:** "Please adhere to the naming standard of 'test_[method under test]_[detail of what is being tested]' to ensure that future maintainers will have an easier time associating tests and the methods they target."

In addition, 39 (8%) of the reviews indicate that the smells are caused by developers' **lack of familiarity with existing code**. An example of such a case is shown below. In this case, the reviewer pointed out that the exception did not raise, so the exception handling code became a *dead code* smell that should be removed. This could imply that the developer was not aware that the specific exception was not raised.

> **Link:** `http://alturl.com/ccjy3`
> **Reviewer:** "on block_device.BlockDeviceDict.from_api(), exception.InvalidBDMVolumeNotBootable does not raise. so it is necessary to remove the exception here."

Twenty-two reviews (5%) attribute **unintentional mistakes of developers** (such as copy and paste) to be the cause of the smells, similar to the example shown below:

> **Link:** `http://alturl.com/zwz2x`
> **Reviewer:** "I think you forgot to remove this."
> **Developer:** "Darn, yes bad copy / paste. Will fix it."

Twenty-two reviews (5%) indicate that **design choices** was the cause of the identified smells. This means that the developers made a poor design choice which introduced the smell. Below is an example in which the reviewer pointed out that the code may indicate that the developer improperly decomposed some test methods.

> **Link:** `http://alturl.com/9ctor`
> **Reviewer:** "The fact that this is now a one liner feels like code smell. I'm not sure, but it may indicate improper decomposition of some of these test methods."
> **Developer:** "I think I see where you're coming from with this. I'm going to have another look."

More specifically, from the perspective of code smell types, the distribution of causes for different smell types are shown in Table 5.

Table 5: The distribution of causes for different smell types

Code Smell	Cause	Count	%
Duplicated Code	cause not provided	683	96.3%
	lack of familiarity with existing code	19	2.7%
	unintentional mistakes of developers	7	1.0%
Bad Naming	violation of coding conventions	368	81.8%
	cause not provided	79	17.6%
	unintentional mistakes of developers	2	0.4%
	lack of familiarity with existing code	1	0.2%
Dead Code	cause not provided	284	85.8%
	lack of familiarity with existing code	19	5.7%
	design choices	15	4.5%
	unintentional mistakes of developers	13	4.0%
Long Method	cause not provided	29	90.6%
	design choices	3	9.4%
Circular Dependency	cause not provided	4	100%
Swiss Army Knife	cause not provided	1	100%
Speculative Generality	cause not provided	1	100%
General Smell	violation of coding conventions	7	63.6%
	design choices	4	36.4%

For *duplicated code*, the majority of reviews (683 out of 709, 96.3%) did not provide any cause for the identified smells. When causes are provided, **lack of**

familiarity with existing code is the main cause for this smell type, accounting for 2.7%. The situation of *dead code* is similar to that of *duplicated code*. In 85.8% of the reviews, no further explanation was provided. When the cause was provided, **lack of familiarity with existing code**, **design choices** and **unintentional mistakes of developers** account for almost the same proportion, 5.7%, 4.5%, and 4.0%, respectively. On the other hand, *bad naming* was different to *duplicated code* and *dead code* smells. The reviewers usually provided an explanation for why they think a bad naming smell existed and **violation of coding conventions** is the main noted cause for *bad naming*. In only 17.6% of reviews was the cause of the *bad naming* smell not provided. For the other 49 smells, 71% of the reviews did not provide the cause of the identified smell. Both **design choices** and **violation of coding conventions** are mentioned in 7 reviews.

RQ2 Summary: In general, over half of the reviews did not provide an explanation of the causes of the smells. In terms of the causes, **violation of coding conventions** is the main cause for the smell as noted by reviewers and developers. Specifically, **violation of coding conventions** is the main cause of the *bad naming* smell. For the other smells, **lack of familiarity with existing code** and **unintentional mistakes of developers** are the main noted causes of smells.

4.3 RQ3: How do reviewers and developers treat the identified code smells?

4.3.1 RQ3.1: What actions do reviewers suggest to deal with the identified smells?

Table 6: Actions recommended by reviewers to resolve smells in the code

Reviewer's recommendation	**Count**
Fix (without recommending any specific implementation)	718
Fix (provided specific implementation)	405
Capture (just noted the smell)	366
Ignore (no side effects)	50

The results of this RQ are shown in Table 6. In the majority of reviews (1,123, 73%), reviewers recommended *fix* resolving the identified code smells. These fixes included either general directions (such as the name of a refactoring technique to be used) or specific actions (pointing to specific changes to the code base that could remove the smell). 405 (36%) of these fixes provided example code snippets to help developers better refactor the smells. An example review where the reviewer suggested a general *fix* action is shown below.

Link: `http://alturl.com/3r3pu`
Reviewer: "remove dead code"
Developer: "Done"

Next is an example of a review that suggested a *fix* recommendation with specific implementation. In this example, the reviewer suggested removing duplicate code from a test case and also provided a working example of how to apply **Extract Method** (i.e., the process of moving part of the code inside a method/function to a separate new method and replacing the existing code with a call to the newly created method) refactoring to define a new test method.

Link: `http://alturl.com/c3g69`
Reviewer: "I think you can do function that remove duplicated code, something like that following..."

```
def _compare(self, exp_real):
    for exp, real in exp_real:
        self.assertEqual(exp['count'], real.count)
        self.assertEqual(exp['alias_name'], real.alias_name)
        self.assertEqual(exp['spec'], real.spec)
```

366 reviews (24%) fall under the *capture* category. In those reviews, reviewers just pointed out the presence of the smells, but did not provide any refactoring suggestions. In a small number of reviews (50, 3%), reviewers suggested ignoring the code smells found in the code reviews. In such a case, reviewers indicated that they could tolerate the identified code smell or there was no need to fix the smell at that point. We further analysed the types of the identified smells when reviewers suggested an *ignore* action. The detailed results are shown in Table 7. Of the code smells that reviewers suggested ignoring, *duplicated code* makes up the majority (68%). *Bad naming* follows, accounting for 22%. The remaining code smells (i.e., *dead code*, *circular dependency* and *long method*) only appear in 5 (10%) reviews.

Table 7: Types of code smells that reviewers suggested *ignore* action

Code Smell	**Count**	**%**
Duplicated Code	34	68%
Bad Naming	11	22%
Dead Code	3	6%
Long Method	1	2%
Circular Dependency	1	2%

We then investigated the specific refactoring actions provided by reviewers in cases where they recommended a *fix* action. Of the 1,123 reviews where reviewers provided *fix* suggestions, there are 754 (67%) reviews where no specific refactoring actions are provided by reviewers. For smells that are straightforward to resolve, such as *dead code* and *bad naming*, reviewers usually just suggested removing the smell without providing concrete refactoring actions in these reviews. In the remainder of the reviews, the distribution of the specific refactoring actions recommended by reviewers is shown in Table 8.

From the table, we can see that reviewers usually suggested concrete refactoring actions for *duplicated code*, such as **Extract Method** and **Consolidate Duplicate Conditional Fragments**. It also indicates that there are multiple types of refactoring actions to solve *duplicated code*. Furthermore, **Extract Method** is the

Table 8: Specific refactoring actions recommended by reviewers to resolve the identified smells

Code smell	Reviewer's recommendation	Count
Duplicated Code	Extract Method	261
	Consolidate Duplicate Conditional Fragments	34
	Extract Superclass	12
	Consolidate Conditional Expression	12
	Parameterize Method	11
	Pull Up Method	5
	Extract Variable	4
	Substitute Algorithm	2
	Extract Subclass	2
	Add Parameter	1
Long Method	Extract Method	19
	Consolidate Duplicate Conditional Fragments	2
Swiss Army Knife	Extract Method	1
General Smell	Extract Method	3

most frequently suggested refactoring action, especially for fixing *duplicated code* and *long method*. Below we present the detailed results of *duplicated code* and *long method* refactoring actions:

Specific refactoring actions for fixing duplicated code

In cases where reviewers named specific refactoring actions (344 reviews), **Extract Method** is the most frequently suggested (in 261 reviews, 76%) refactoring. Below is an example of where the reviewer suggested extracting the duplicated code to a new private method.

> **Link:** `http://alturl.com/482nn`
> **Reviewer:** "Instead of duplicating the function body, use a _data() function to supply test rows such as (cipher, presence of the ephemeral key)"

Consolidate Duplicate Conditional Fragments (i.e., move the code which can be found in all branches of a conditional outside of the conditional) and **Consolidate Conditional Expression** (i.e., consolidate all conditionals that lead to the same result or action in a single expression) are recommended in 34 (10%) and 12 (3%) reviews, respectively to solve duplication in conditional statements. Below is an example from Qt Base project, where the reviewer provided specific code snippets by consolidating duplicate conditional fragments.

Link: `http://alturl.com/on6ts`
Reviewer: "You can remove the duplicate rect.setSize() by re-ordering these conditions."

```
if (rect.width() < ...) {
    if (rect.isEmpty() && (touch->device...)
        diameter = ...
    rect.setSize
}
```

We note that **Extract Superclass** (i.e., create a shared superclass for the classes with common fields and methods and move all the identical fields and methods to the superclass), **Pull Up Method** (i.e., make the methods that perform similar work in subclasses identical and then move them to the relevant superclass) and **Extract Subclass** (i.e., create a subclass and use it in cases where a class has features that are used only in certain cases) were suggested in 12, 5, and 2 reviews, respectively. These three refactoring actions were used to deal with generalisation in classes (e.g., two or more classes with common fields and methods that can be grouped together and the original classes extend the newly created superclass). Below is an example from the Nova project:

Link: `http://alturl.com/z6svv`
Reviewer1: "A lot of these methods appear to be duplicated from LibvirtISCSIVolumeDriver. Maybe just use it as the base class for LibvirtISERVolumeDriver?"
Reviewer2: "I agree with 'Reviewer1'. This is a very good point."
Developer: "I'm working on that.. thanks."

Parameterize Method (i.e., combine methods that perform similar actions that are different only in their internal values, numbers or operations by using a parameter that will pass the necessary special value) and **Add Parameter** (create a new parameter to pass the necessary data for the method which does not have enough data to perform certain actions) were suggested in 11 and 1 reviews, respectively. When multiple methods perform similar actions differing only in their internal values, numbers or operations, these refactoring actions can be used (similar to code duplication) to combine these methods by using a parameter (argument) passed as a value to the method. An example of the **Parameterize Method** refactoring in the Neutron project is shown below:

Link: `http://alturl.com/o3r38`
Reviewer: "The same comment applies to the below. The methods below are very similar and it would be better to define a common method which takes "resource" name as an argument."
Developer: "Done"

There are four reviews in which the reviewers suggest **Extract Variable** to remove the *duplicated code*. In another two reviews, the *duplicated code* was caused by bad (algorithm) implementation and the reviewers suggested using **Substitute**

Algorithm as a solution.

Specific refactoring actions for fixing long method

Of the 21 reviews where the reviewers provided specific refactoring actions, **Extract Method** was also suggested as a way of extracting the *long method* (appearing in 19 reviews). Below is an example of a review where the reviewer suggested splitting the code to separate (more manageable) methods.

> **Link:** `http://alturl.com/33dvx`
> **Reviewer:** "Aside: this function is crazy long and could probably benefit from some refactoring into smaller helper functions."
> **Developers:** "Agreed, I'll make a note to revisit during R."

There are two reviews in which the reviewers suggested **Consolidating Duplicate Conditional Fragments** to refactor a *long method*.

4.3.2 RQ3.2: What actions do developers take to resolve the identified smells?

Table 9 provides details of the number of reviews that identified code smells versus the number of fixes of the identified code smells.

Table 9: Developers' actions to code smells identified in code reviews

Code smell	#Reviews	#Fixed by developers	% of fixes
Duplicated Code	709	561	79%
Bad Naming	450	400	89%
Dead Code	331	307	93%
Long Method	32	24	75%
Circular Dependency	4	2	50%
Swiss Army Knife	1	1	100%
Speculative Generality	1	1	100%
General Smell	11	8	73%
Total	**1,539**	**1,304**	**85%**

Of the 1,539 code smells identified in the reviews, the majority (1,304, 85%) were refactored by developers after the review (i.e., changes were made to the patch). As per the results of RQ1, *duplicated code*, *bad naming*, and *dead code* were the most frequently identified smells by reviewers. Subsequently, those smells were also widely resolved by developers. 561 (79%) *duplicated code*, 400 (89%) *bad naming* and 307 (93%) *dead code* smells were refactored by developers after they were identified in reviews. The proportion of other smells being fixed are 73% (36/49).

Below is an example of a review with a recommendation by the reviewer to remove *dead code* in Line 132 of the original file (i.e., remove the `pass` statement); the developer then agreed with the reviewer's recommendation and deleted the unused code. Fig. 8 shows the code before review (Fig. 8a) and after the action taken by the developer (Fig. 8b).

Link: `http://alturl.com/szswu`
Reviewer: "you can remove 'pass', it's commonly considered as dead code by coverage tool"
Developer: "Done"

(a) method before review (b) after change made by the developer

Fig. 8: An example of a *remove dead code* operation after review (the change is highlighted in Line 132 (a))

There are 28 (2%) reviews in which the developers indicated that they would fix the identified smells in a later change or commit. In this case, it is difficult to determine whether the identified smells are fixed or not. We categorized these cases as *Unknown*. Below is an example where the developer promised to remove the duplication in another patch.

Link: `http://alturl.com/dqev8`
Reviewer: "nit: I'm pretty sure we have this same pattern in several places in this driver code, we should create a libvirt.utils helper method for this at some point, i.e. is_parallels(vm_mode=None)."
Developer: "Ok, I'll do it in another patch"

The remaining 207 (13%) reviews do not lead to any changes in the code, indicating that developers may have chosen to ignore such recommendations. This could be a case where the developers thought that those smells were not as harmful as suggested by the reviewers, or that there were other issues requiring more urgent attention, resulting in those smells being counted as technical debt in the code (Li et al., 2015).

4.3.3 RQ3.3: What is the relationship between the actions suggested by reviewers and those taken by developers?

For answering this RQ, a visual map of reviewer recommendations and resulting developer actions is shown in Fig. 9.

In 971 (63%) of the obtained reviews, developers agreed with the reviewer's suggestion and took exactly the same actions (either *fix* or *ignore*) as suggested by the reviewers. Of those cases, there are 23 cases where developers agreed with reviewers on ignoring the smells (i.e., a smell had been identified, but the reviewer

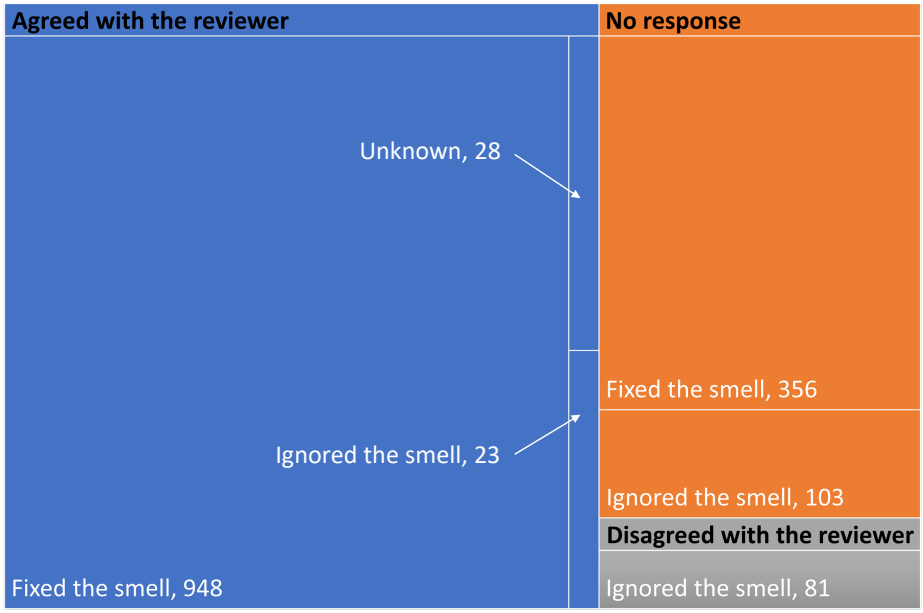

Fig. 9: A treemap of the relationship between developers' actions in response to reviewers' recommendations regarding code smells identified in the code

may have thought that the impact of the smell was minor and *there is no need to fix the smell now*). The example below shows a case where a reviewer pointed out that they could accept duplicated code if there was a reasonable justification and the developer gave their explanation and ignored the smell.

> **Link:** `http://alturl.com/s59so`
> **Reviewer:** "...I just don't like duplicated code but if there is a reasonable justification for this I can be sold cheaply and easily."
> **Developer:** "we need `create_vm` here to support a lot of the other testing in this method. I agree it's duplicate code, but it's needed here too and this one is more complex that (sic) the `test_config` one...."

There are 28 (2%) reviews where the developers agreed with reviewers but did not make any changes to the code immediately; however, the developers promised to fix the identified smells in a later patch or commit. Below is an example in which the developer promised to remove the duplication in a follow-up change.

> **Link:** `http://alturl.com/pzmzz`
> **Reviewer:** "Can't we use the same enum class for both instead of keeping in sync? (for example in follow-up patch)"
> **Developer:** "Yes, I will remove the duplication in a follow up change."

In 356 (23%) reviews, even when developers did not respond to reviewers directly in the review system, they still made the required changes to the source code

files. We note that there are other 81 (5%) reviews where developers had different opinions from reviewers and decided to ignore the recommendations to refactor the code and remove the smell. In those cases, the developers themselves decided that the smell was either not as critical as perceived by the reviewers, or there were time or project constraints preventing them from implementing the changes, which are typically self-admitted technical debt (Potdar and Shihab, 2014). An example review is shown below:

> **Link:** `http://alturl.com/pzmzz`
> **Reviewer:** "This method has a lot duplicated code of '_apply_instance_name_template'. The differ in the use of 'index' and the CONF parameters. With a bit refactoring only one method would be necessary I guess."
> **Developer:** "I thought to make / leave this separate in case one wants to configure the multi_instance_name_template different to that of single instance."

Similarly, there are also 103 (7%) reviews in which developers neither replied to reviewers nor modified the source code. For those cases, we suppose that the developers did not find the recommendations regarding how to deal with the specific smells in the code helpful and therefore decided not to perform any changes. In all of those cases, no further explanation/reasons were provided by the developers on why they ignored these recommended changes.

RQ3 Summary: In most reviews, reviewers provide fixing (refactoring) recommendations (e.g., in the form of code snippets) to help developers remove the identified smells. Developers generally follow those recommendations and perform the suggested refactoring operations, which then appear in the patches committed after the review.

4.4 RQ4: How long does it take to resolve code smells by developers after they have been identified by reviewers?

According to the result of RQ3.2, a total of 1,304 (85%) code smells identified in the reviews were fixed by developers. Of these, we removed cases where the time taken for the fix was more than one year (4 reviews), the resolution time could not be determined (2 reviews), and the identification time was earlier than the resolution time (2 reviews, as explained in Section 3.7.4). We also note that some smell categories are very few in number from the 1,304 code reviews (e.g., *long method* and *circular dependency*) and we chose to exclude them (35 reviews) from the analysis of this RQ because their resolution time may have limited statistical significance. Thus, the main smell categories we investigated in this RQ are *duplicated code*, *bad naming*, and *dead code* smells. Finally, we analysed 1,261 code reviews to answer RQ4. Table 10 shows minimum, quartile, maximum, and mean time for fixing different categories of code smells.

As shown in Table 10, we found that the minimum time taken for fixing *duplicated code*, *bad naming*, and *dead code* could be very short - only around one or two minutes. However, it may also take a very long time to fix these code smells

Table 10: Time taken for fixing different categories of code smells

	Duplicated Code	**Bad Naming**	**Dead Code**
Count	555	401	305
Minimum Time (second)	69s	127s	63s
Lower Quartile (hour)	7.9h	3.4h	2.1h
Median Time (hour)	23.0h	20.5h	19.1h
Higher Quartile (day)	5.0d	3.8d	3.7d
Maximum Time (day)	352.0d	325.3d	291.2d
Mean Time (day)	12.2d	9.2d	7.8d

(around a year in some cases). According to the quartile time (i.e., lower quartile, median, and higher quartile time) for fixing these three categories of code smells, *duplicated code* was fixed slower than the other two smells. We also conducted a survival analysis using the Kaplan–Meier curves (Kaplan and Meier, 1958). The survival curve is shown in Fig 10, which also supports our finding above.

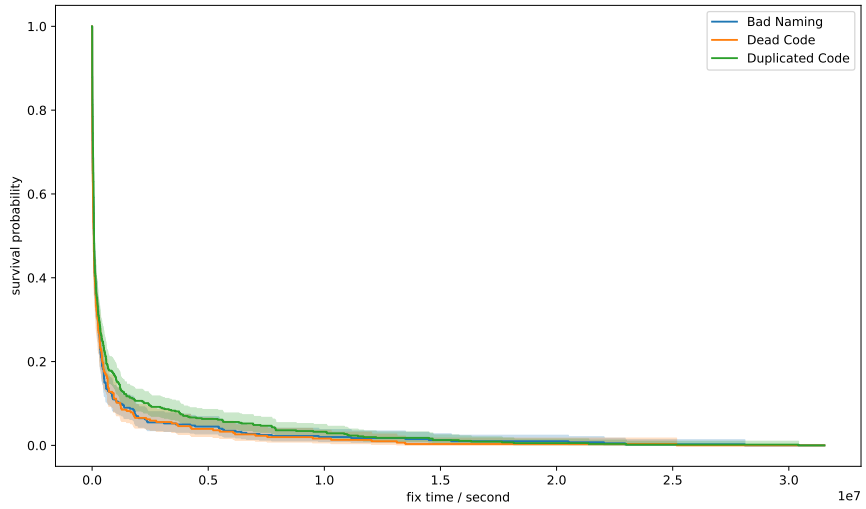

Fig. 10: The survival curve of different smells identified in code reviews

Further, we used a Kruskal-Wallis Test (Kruskal and Wallis, 1952) to determine whether or not there was a statistically significant difference in fix time between these three groups. We used SciPy[25] (fundamental algorithms for scientific computing in Python) to perform the Kruskal-Wallis Test. In this case, the test statistic is 13.3041 and the corresponding *p-value* is 0.0013. Since the *p-value* is less than 0.05, we can conclude that the category of code smell lead to a statistically significant difference in fix time.

[25] `https://scipy.org/`

Fig. 11 shows the distribution of time taken for fixing those code smells. From this figure, we observe that 1,045 smells (83%) were fixed within one week. More than half of smells (674, 53%) were fixed within one day, and 371 (29%) smells were fixed in more than one day but within one week. 125 (10%) smells took developers 2-4 weeks to fix, and only 91 (7%) smells took more than one month to fix.

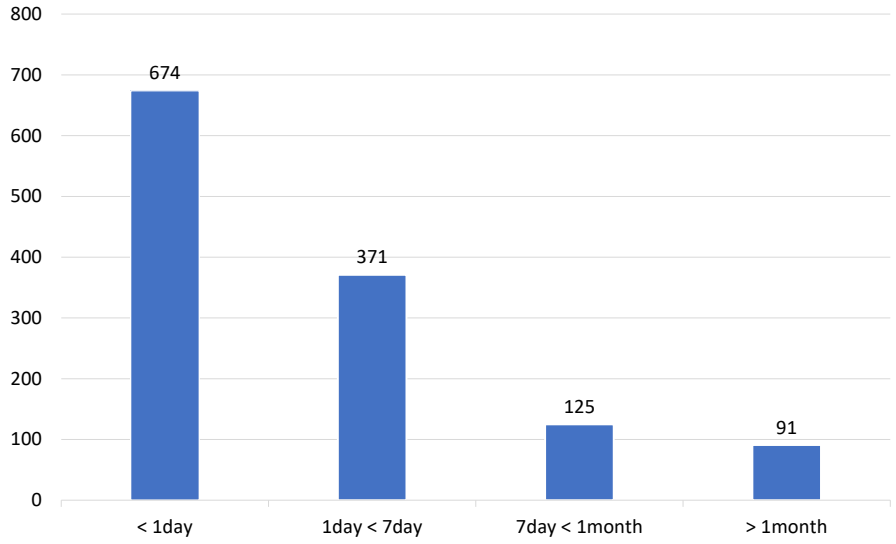

Fig. 11: Distribution of time for fixing smells identified in code reviews

We also calculated the minimum, quartile, maximum, and mean time for fixing code smells based on whether the specific refactoring actions were provided or not. The results are shown in Table 11. We can see that the time taken to fix the identified smells is almost the same under these two circumstances. We formed a null hypothesis that there was no statistical difference in time taken to fix a smell whether specific refactoring suggestions have been provided or not. We then performed a Mann-Whitney Test (Mann and Whitney, 1947) to compare the differences between those two groups. The test statistic is 160548.5 and the corresponding *p-value* is 0.7800 (i.e., *p-value*>0.05); it can therefore be concluded that there is no statistically significant difference between fix time in these two categories. Based on the results, we can see that whether reviewers provided specific refactoring suggestions or not has little effect on the fix time of the identified code smells.

RQ4 Summary: Among the studied smells, *duplicated code* smells took more time to fix compared to *bad naming* and *dead code* smells. Moreover, 83% of the smells were fixed by developers within one week from being identified in code reviews.

Table 11: Time taken for fixing smells (classification according to whether the reviewers provide specific refactoring actions or not)

	Specific refactoring actions provided	**No specific actions provided**
Count	360	901
Minimum Time (second)	69s	63s
Lower Quartile (hour)	4.5h	4.6h
Median Time (hour)	21.6h	21.1h
Higher Quartile (day)	3.9d	4.2d
Maximum Time (day)	266.1d	352.0d
Mean Time (day)	8.2d	10.9d

4.5 RQ5: What are the common causes for not resolving code smells that have been identified in code?

According to the results of RQ3.2, there are 81 (5%) reviews in which developers disagreed with reviewers and chose to ignore the identified code smells. We excluded one review as we were not able to access the URL link provided by the developer. Fig. 12 shows this review, in which the developer only replied with a URL to the reviewer, but to which we had no access. We then inspected the later patches and the final status of the code change. We found that the developer made no change in later patches and the code change was finally merged to the code base. Considering this, we treated this as a case of an *ignore* of the reviewer's suggestion. We could not find the reason why the developer ignored the smell and consequently we excluded this review from our analysis.

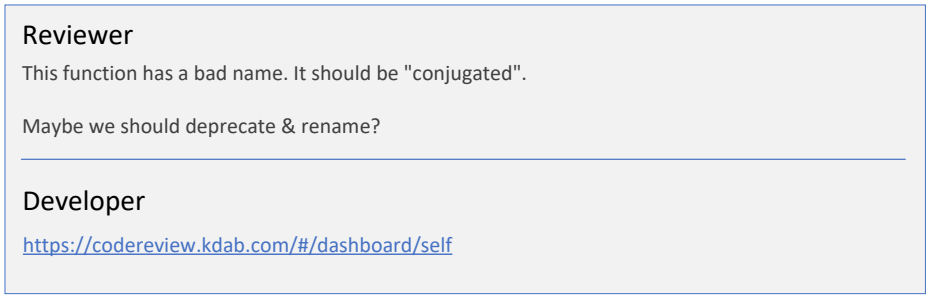

Fig. 12: The review in which we could not find the reason why the developer ignored the smell

As shown in Table 12, we found that the main reason for why developers ignored identified smells was simply that it was **not worth fixing the smell** (in 28 reviews). In this case, the developers thought that there were more important things to consider, or fixing the smells would bring little value or add more complexity. An example of such a case is shown below. In this case, the developer thought that removing the duplication would make specific comparisons more expensive and the developer chose to ignore the smell.

Table 12: Distribution of causes for ignoring smells identified in code reviews

The cause	Count	%
Not worth fixing the smell	28	35%
Difference in opinion between developers and reviewers	20	25%
Limited by developers' knowledge	9	11%
Keep consistent with other code	8	10%
For future consideration	7	9%
Cause other errors when fixing the smell	4	5%
Improve code readability	4	5%

Link: `http://alturl.com/76z9f`
Reviewer: "How about implementing int compare() which returns <0, 0 or >0, and then implement the other comparisons in terms of that? Should result in less code duplication as there currently is with operator<and operator== ..."
Developer: "I don't see how this reduces code duplication. I can see how a compare would allow one function to do all comparisons but it would make specific comparisons more expensive."

20 reviews attribute **difference in opinion between developers and reviewers** to be the reason why developers ignored the smells. This means that although the reviewers identified the code smell, the developer thought it was acceptable and chose not to fix it. In the below example, the reviewer suggested removing the *duplicated code* while the developer thought that there was no problem and ignored the smell.

Link: `http://alturl.com/up6a5`
Reviewer: "It's kind of crazy to have to duplicate this kind of logic in the tests. The first thing I'd like to suggest is extracting event name determination into its own method so it can be easily mocked out in unrelated tests."
Developer: "I'd prefer to just leave this for now if that's okay"

In nine reviews, the reason why developers ignored the identified smell was **limited by developers' knowledge**. This means that developers could not find a better way to fix the identified smell or they did not have enough information to fix it. Here is an example about *bad naming* where the developer could not find a better name and left it as it was.

Link: `http://alturl.com/z753q`
Reviewer: "naming might be confusing. You could also include Result::MessageIntermediate and think of a better name for this group... (just to save an unnecessary roundtrip: no suggestions.)"
Developer: "got no better naming idea as well.. leaving this for now as is, guess this will change when the whole Message* stuff gets re-done."

In eight reviews, the reason why developers ignored the smell is that they chose to **keep consistent with other code**. The following example shows that the developer just used the original name, although the reviewer thought that it was poor naming.

Link: `http://alturl.com/onn6f`
Reviewer: "Gosh, what an awful naming convention..."
Developer: "I agree but I just used the original one"

Seven reviews indicate that developers chose to ignore smells **for future consideration**. Developers indicated that they preferred to keep the status quo to help future changes. There are also cases in which developers indicated that the smells would be fixed once some certain features came online. Below is an example in which the developer believed that small duplication would help to make things cleaner for subsequent changes.

Link: `http://alturl.com/bevos`
Reviewer: "This really needs to be extracted into a common method. We will suck at maintaining the API if we have this level of duplication."
Developer: "I think we could remove duplication for force param here, but leave things as is for other params, e.g. common get_args method will add microversion checks for block_migration param, see lines 83-97, which I'd like to leave for microversion 2.34, because it's already supports 'auto' for block_migration. So this small dup will help to make things cleaner for subsequent changes."

Four reviews indicated that dealing with a particular smell could **cause other errors when fixing the smell**. The following example shows that removing the identified *dead code* (i.e., the `return` statement) would produce errors in release mode.

Link: `http://alturl.com/au5pu`
Reviewer: "No dead code, please."
Developer: "It's dead code but if I omit the "return 0" statement I get errors in release mode about missing return value."

In the remaining 4 reviews, the developers noted that the existence of smells would have a positive impact and it would, in general, **improve code readability**, as shown in the example below.

Link: `http://alturl.com/uzo6c`
Reviewer: "nit: duplicates to lines 444-449; could be refactored into an attribute."
Developer: "True, but that would make the test a little less readable IMO. In this case I think the duplication is worth it"

We also checked the status of the code changes where developers disagreed with reviewers and chose to ignore the identified smells. The detailed results are show in Table 13. Although the identified smells were ignored by developers, 61 (75.3%) code changes were still merged to the primary codebase while only 20 (24.7%) code changes were finally abandoned or deferred.

Table 13: The status of code change where developers disagreed with reviewers and ignored the smells

Code Change Status	Count	%
Merged	61	75.3%
Abandoned	19	23.5%
Deferred	1	1.2%

RQ5 Summary: Developers disagreed with reviewers in only a small number of code reviews (81, 5%). In terms of the reasons for disagreement, **not worth fixing the smell** was the main reason for ignoring the identified smells. Although developers disagreed with reviewers and ignored the smells, 75.3% of these changes were still merged into the codebase.

5 Discussion

5.1 RQ1: The most frequently identified smells

In general, most of the smells are not extracted from the code review data. One potential reason is that code smells do not appear frequently during the development of the four selected projects in this study, or, simply, that the reviewers were unaware of the presence of certain code smells. Another potential reason is that reviewers were aware of code smells, but that they did not consider them very harmful. One avenue of further work would be to run smell detection tools and compare the smells identified by these tools with the smells identified manually by reviewers to better understand how many code smells are being missed by reviewers. Interviews with the code reviewers can also provide further understanding of the reasons behind the low number of code smells being discussed in code review.

The results of RQ1 imply that *duplicated code*, *bad naming* and *dead code* are, by far, the most frequently identified code smells in code reviews. Results regarding *duplicated code* are in line with previous findings which indicate that this smell is frequently discussed among developers in online forums (Tahir et al., 2020) and is also the smell that developers are most concerned about (Yamashita and Moonen,

2013). However, *dead code* and *bad naming* were not found to be ranked highly in previous studies (Yamashita and Moonen, 2013). The different results are due to the different context and domain, critical to identifying smells, as shown by previous studies (Tahir et al., 2020; Yamashita and Moonen, 2013). The results reported in these two previous studies (Tahir et al., 2020; Yamashita and Moonen, 2013) are based on a more generic investigation of code smells among online Q&A forum users and developers. The context of some of these code smells is not fully taken into account, even if the developers provide specific scenarios to explain their views. In contrast, the results reported in this study are project-centric and the context of the identified code smells during code reviews is known to reviewers and developers involved in the identification and removal of the smells. This is also supported by our further investigation of the refactoring actions applied to source code once smells are identified (discussed in RQ3 and RQ5).

A study by Palomba et al. (2018) has shown that the most diffuse code smells are those characterized by long and/or complex code (e.g., *complex class*), which is different from our results of RQ1. One potential reason for this is the contrast in code smells considered in the two studies. In the study of Palomba et al. (2018), they did not consider *duplicated code*, *dead code* and *bad naming*, but focused on a larger granularity of code smells, i.e., at the class and method level. Another potential reason is that, in modern code review, code changes are kept as small as possible to facilitate the review process and this may contain very few class level design issues, e.g., *complex class*. Code smells at a lower level of granularity (e.g., *dead code*) are easier to detect on the fly (especially with conditional statements), while complexity related smells can be hard to detect without the use of detection tools. The reasons for the difference are also an interesting aspect that should be explored in future research.

In addition, different styles of code reviews may affect smell detection. In this work, we focused on the modern code review process that reviews code changes. Compared with other styles of code reviews (e.g., reviewing code instead of changes to the code base), changes by default are going to be smaller in a modern code review setting, leading to detection of smells with a smaller granularity. Other styles of code review may consider the whole project and it is more likely to identify smells at higher levels, such as project and component level code smells.

5.2 RQ2: The causes for identified smells

In general, we identified four types of common causes (see Fig. 7) for code smells in code reviews (RQ2). Among these, **violation of coding conventions** was the major cause of code smells identified in reviews. Conventions are important in reducing the cost of software maintenance, while the existence of smells can increase this cost. We conjecture that this is because developers may not be familiar with the coding conventions of their community and the system they implemented. More specifically, **violation of coding conventions** is the main cause for the *bad naming* smell. Usually, communities or companies will have a specific naming convention, which can help improve the readability of code. The main cause for *duplicated code* and *dead code* is **lack of familiarity with existing code**. For example, *duplicated code* and *dead code* may occur because developers are unaware of existing functionality. This implies that a developer's unfamiliarity with coding conventions or

existing code can inadvertently lead to smells or other problems and this can have a negative impact on software quality.

Another main observation is that more than half of reviewers (in review comments where they indicated that there was a code smell) simply pointed out the smells in the code, but did not provide any further explanation as to why they considered that as a smell. One explanation for this is that the identified smells are simple or self-explanatory (e.g., *duplicated code*, *dead code*). Therefore, it is not expected that the reviewers needed to provide any further explanation for these smells. Although the point of code review is to identify shortcomings (including potential code smells) in contributed code, understanding the causes of code smells can help practitioners better understand how the code smell was introduced and then take corresponding remedial measures.

5.3 RQ3: The relationship between what reviewers suggest and the actions taken by developers

The results of RQ3 show that reviewers usually provide useful recommendations (sometimes in the form of code snippets) when they identify smells in the code and developers usually follow these suggestions. Given the constructive nature of most reviews, developers tend to agree with the review-based smell detection mechanism (i.e., where a reviewer detects and reports a smell) and, in most cases, they perform the recommended actions (i.e., refactoring their code) to remove the smell. We believe that this is because reviewers can take more information into account as the program context and domain are important in identifying smells (Sae-Lim et al., 2018; Tahir et al., 2020; Yamashita and Moonen, 2013).

The result of RQ3.1 shows that reviewers usually provide general refactoring instructions (i.e., remove or refactor the smells) without specific suggestions (i.e., how to refactor the smells) and these types of smells are usually easy to fix. For example, *dead code* is usually resolved by simply removing the unused or unreachable code. For *bad naming*, it is usually a matter of coming up with a different name and changing a small part of source code. Moreover, compared with the recommendations for *dead code* and *bad naming*, there are more types of specific refactoring actions suggested by reviewers for resolving *duplicated code*. One possible reason could be that *duplicated code* is more difficult to repair by contrast and consequently reviewers would propose more detailed actions to help developers remove those smells. Another finding is that **Extract Method** is the most frequently suggested refactoring action, which is consistent with what Fowler states in his seminal refactoring text (Fowler, 1999): "*Extract Method is one of the most common refactorings I do. I look at a method that is too long or look at code that needs a comment to understand its purpose. I then turn that fragment of code into its own method*".

There were some case when reviewers suggested ignoring identified smells. In these cases, we found that *duplicated code* made up the majority. In general, the tolerance of *duplicated code* varies from one reviewer to another. When reviewers identified *duplicated code*, but the number of lines of duplicated code did not reach their threshold, reviewers often indicated that the relevant *duplicated code* could be ignored.

5.4 RQ4: The time taken for fixing the smells

From the perspective of code smell categories, it usually takes more time to fix *duplicated code* than *dead code* and *bad naming*. We believe that this is related to the nature of those code smells. Usually, code duplication involves multiple parts of the source code rather than a single part, which makes it more difficult to fix than *dead code* and *bad naming*. Another finding is that the longest time taken for fixing *duplicated code*, *bad naming*, or *dead code* was around 300 days. We posit that this is partially related to the way developers work on patches and abandon code changes. When the developer uploads a new patch, it may not be used to solve the identified code smell and to solve other problems also. In some reviews, we regarded the time of abandoning the code change where the smell locates as the resolution time of the identified code smell. This could also prolong the resolution time of the code smell we obtained because the code change is usually abandoned after a long time without any update.

Moreover, from the perspective of the distribution of the time taken to fix smells, most of the identified smells were fixed within one week from the time they were first identified in the reviews. We suspect that this finding may be related to the nature of the code smell and the reviewers' recommendations. *Dead code* and *bad naming* are usually easier to fix as we explained in Section 5.3. Additionally, we found that 7% of smells took more than one month to fix. We then further checked related information on these smells, i.e., the code review discussions, but found that no reasons were provided for such a delay in most cases. The developers just uploaded the patches or abandoned the code changes after a significant time period without providing any reasoning. Only in one review[26], a developer indicate that it was not the right time to fix the identified code smell.

5.5 RQ5: Reasons for ignoring the identified smells

Although not as frequently occurring, there are cases where changes recommended by reviewers were ignored (see Figure 9). The result for RQ5 shows that **not worth fixing the smell** is the main reason why developers ignored removing the smells from the code. In other words, it is assumed that fixing the smells would either add more complexity to the code, or just bring little value as a result. The context of the smell (such as the time of fixing it, the complexity it brings, whether there is something more important, etc.) should be taken into full consideration and the value that the fix will bring should also be assessed.

Another reason is **difference in opinion between developers and reviewers**, consistent with what Fowler noted (Fowler, 1999) that "*no set of metrics rivals informed human intuition*". This situation is partially due to the different understanding or experience of reviewers and developers about the severity of identified code smells. When a reviewer identifies a code smell to be resolved, a developer may not agree that the code smell needs to be fixed; equally, that it is an issue which can be fixed later in the same way that technical debt is accrued (Li et al., 2015).

We also found that although developers ignored smells, most of the code changes were still merged into the codebase. One potential reason for this is that

[26] `http://alturl.com/rrxo7`

the proposal of a code change is not specifically used to fix the identified code smell. The introduction of a code smell is usually a side product and the existence of a code smell may have little influence on the merging of code change. It also means that the program context of a code smell has a great impact on how harmful the smell is and whether or not it needs to be fixed immediately.

5.6 Implications

There are a number of implications of the work contained in this paper. First, although we built the initial set of keywords with 5 general code smell terms and 40 specific code smell terms, most of the smells are not extracted from the code review data (e.g., *long parameter list*, *temporary field*, and *lazy class*). Gerrit is designed to review code changes and smells might not be mentioned during the code review process if they are not deemed severe enough by the reviewers. Another potential reason is that code smells considered as problematic in academic research may not be considered as a pressing problem in industry. Thus, more research should be conducted with practitioners to explore existing code smells and to understand the driving force behind industry efforts on code smell detection and elimination. This will help to guide the design of next-generation code smell detection tools.

Second, **violation of coding conventions** is the main cause of code smells identified in code reviews. It implies that a developer's lack of familiarity with the coding conventions in their company or organization could have a significantly negative impact on software quality. To reduce code smells, project leaders need to adopt code analysis tools and also help educate their developers to become familiar with the coding conventions adopted in the system; we note that some tools can also be used to automatically check for compliance with code conventions.

Third, in smell-related reviews, reviewers usually give useful suggestions to help developers better fix the identified code smells and developers generally tend to accept those suggestions. Review-based detection of smells is seen as a trustworthy mechanism by developers. Although code analysis tools (both static analyzers and dynamic (coverage-based) tools) are able to find some of those smells, their large outputs restrict their usefulness. Most tools are context and domain-insensitive, making their results less useful due to potential false positives produced by these tools (Fontana et al., 2016).

Fourth, it usually takes developers less than one week to fix an identified smell. According to the results of RQ4, providing detailed recommendations has little influence on the fix time of code smells. Fixing those smells depends on many factors, most importantly program context. For relatively less complex code smells (e.g., *duplicated code*), reviewers may merely point out the existence of those smells rather than spending time making more detailed suggestions for their removal.

Fifth, there are cases where developers disagreed with reviewers and ignored identified smells (see Figure 9). Of these, **not worth fixing the smell** is the main cause when developers chose to ignore identified smells. This could imply that developers do not tend to make any changes to existing code where fixing code smells takes significant effort (a typical technical debt scenario (Li et al., 2015). That is, context seems to matter in deciding whether a smell is bad or not (Sharma and Spinellis, 2018; Tahir et al., 2020). There have been some recent attempts to develop smell-detection tools that take developers-context into account

(Pecorelli et al., 2020; Sae-Lim et al., 2018). However, contextual factors such as project structure and developer experience are much harder to capture with tools. Code reviewers are much better positioned to understand and account for those contextual factors (as they are involved in the project) and their assessment of smells might be trusted more by developers than that of automated detection tools.

Finally, to increase the reliability of detecting code smells, it may need a two-step detection mechanism; first, static analysis tools to identify smells (as they are faster than human assessment and also more scalable) and second for reviewers to go through those smell instances. They should then decide, based on the additional contextual factors, which of those smells should be removed and at what cost. One potential problem with such an approach is that most tools would probably produce large sets of outputs, making it impractical for reviewers working on a large code base; improving the accuracy of smell-detection tools is vital for its application in such a context.

6 Threats to Validity

Given the empirical nature of our study, potential threats can affect the study results. We classify and discuss these threats by following the recommendations suggested in Wohlin et al. (2012).

External Validity: Our study considered two major projects from the OpenStack community (Nova and Neutron) and two major projects from the Qt community (Qt Base and Qt Creator), since those projects have invested a significant effort in their code review process (see Section 3.2). OpenStack is a set of software tools for building and managing cloud computing platforms and Qt is an open source cross-platform application and UI framework. The projects from the OpenStack community are mainly written in Python, while the projects from the Qt community are mainly written in C++. Different domains and programming languages can help improve the external validity and make the study results and findings more generalizable to other systems. We believe that our results and findings could help researchers and developers understand the importance of the manual detection of code smells better. Moreover, including code review discussions from other communities will supplement our findings and this may lead to more general conclusions.

Internal Validity: The main threat to internal validity is related to the quality of the selected projects. It is possible that the projects we selected do not provide a good representation of the types of code smells we included in our study. To address this threat, we selected two large projects from the OpenStack community and two large projects from the Qt community with Gerrit as their code review tool. Their investment in code review processes and commitment to perform code review to their entire code base make them good candidates for our study. Another threat to the internal validity is the nature of modern code review used in Gerrit. In such code review platforms, code changes (instead of a code snapshot or the whole codebase) are reviewed, where the changes are usually micro to small changes. The practice with such a review system is to review small additions or modifications to the codebase. By default, this will limit higher, more abstract project-level smells and issues from being detected.

Construct Validity: A large part of the study depends on manual analysis of the data, which could affect the construct validity due to personal oversight and bias. In order to reduce its impact, each step in the manual analysis (i.e., identifying smell-related code reviews and their classifications in various aspects) was conducted by at least two authors and a third author was involved in case of disagreement. The selection of the keywords used to identify the reviews which contain smell discussions is another threat to construct validity since reviewers and developers may use terms other than those that we used in our mining query. To minimize the impact of this threat, we first combined a list of code smell terms that developers and researchers frequently used, as reported in several previous studies (Fowler, 1999; Tahir et al., 2018; Zhang et al., 2011). Then, we identified the keywords by following the systematic approach used by Bosu et al. (2014) to minimize the impact of missing keywords due to misspelling or other textual issues. Moreover, we randomly selected a collection of code review comments that did not contain any of our keywords to supplement our dataset, reducing the threat to construct validity.

Reliability: Before starting our full scale study, we conducted a pilot run to check the suitability of the data source. The execution of all the steps in our study, including the process of data mining, data filtering and manual analysis was discussed and confirmed by at least two of the authors. We also provided the replication package of this work online (Han et al., 2022) for replication purposes, which partially increases the reliability of the study results.

7 Conclusions

In this work, we conducted an empirical analysis of code smells identified in modern code review (MCR). Although there are many studies focusing on code smells or code reviews, little is known about the extent to which code smells are identified and resolved during MCR. To this end, we statistically analysed the code review comments from four most active projects of the OpenStack community (Nova and Neutron) and the Qt community (Qt Base and Qt Creator). More specifically, we manually analysed the *types*, *causes*, *actions*, and *fixing interval* of/towards the identified code smells.

According to our results, code smells are not commonly identified in code reviews and when identified, *duplicated code*, *bad naming* and *dead code* are, by far, the most frequently identified smells. When smells are identified, most reviewers provide constructive suggestions to help developers fix the code and developers are willing to fix the smells through suggested refactoring operations; it usually takes developers less than one week to fix the identified smells. We also found that code smells were often introduced as a result of developers violating coding conventions. Although not as frequent, there are also cases where developers disagree with the reviewers and ignore identified smells. The main cause for it is that developers think it is not worth fixing the smells (i.e., bring little value or introduce more complexity when fixing).

Based on our findings, we make the following suggestions for both researchers and practitioners:

1. Developers should follow the coding conventions in their projects to reduce code smell incidents; some tools can also be used to automatically check for compliance with code conventions.
2. Code smell detection via code reviews is seen as a trustworthy approach by developers (given their constructive nature) and smell-removal recommendations made by reviewers appear more actionable by developers.
3. Program context is important in the identification of code smells and also should be taken into account in order to determine whether to fix the identified code smells immediately.

In the next step, we plan to extend this work by studying code smells in code reviews in a larger set of projects from different communities, including from industrial projects. We also plan to obtain a further understanding of the practitioners' attitude towards code smells in code reviews by conducting a survey with both reviewers and developers and thus take a closer look at what the academic and industrial communities think about code smells via code review.

Acknowledgements

This work is supported by the National Natural Science Foundation of China (NSFC) under Grant No. 62172311 and the Special Fund of Hubei Luojia Laboratory.

References

Abbes M, Khomh F, Gueheneuc YG, Antoniol G (2011) An empirical study of the impact of two antipatterns blob and spaghetti code on program comprehension. In: Proceedings of the 15th European Conference on Software Maintenance and Reengineering (CSMR), IEEE, pp 181–190

Baker Jr RA (1997) Code reviews enhance software quality. In: Proceedings of the 19th International Conference on Software Engineering (ICSE), ACM, pp 570–571

Bavota G, Qusef A, Oliveto R, De Lucia A, Binkley D (2015) Are test smells really harmful? an empirical study. Empirical Software Engineering 20(4):1052–1094

Bosu A, Carver JC, Hafiz M, Hilley P, Janni D (2014) Identifying the characteristics of vulnerable code changes: An empirical study. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE), ACM, p 257–268

Braun V, Clarke V (2006) Using thematic analysis in psychology. Qualitative Research in Psychology 3(2):77–101

Cassee N, Vasilescu B, Serebrenik A (2020) The silent helper: The impact of continuous integration on code reviews. In: Proceedings of the 27th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, pp 423–434

Chouchen M, Ouni A, Kula RG, Wang D, Thongtanunam P, Mkaouer MW, Matsumoto K (2021) Anti-patterns in modern code review: Symptoms and prevalence. In: Proceedings of the 28th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, pp 531–535

Coelho F, Tsantalis N, Massoni T, Alves ELG (2021) An empirical study on refactoring-inducing pull requests. In: Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), ACM, pp 1–12

Cohen J (1960) A coefficient of agreement for nominal scales. Educational and Psychological Measurement 20(1):37–46

Dou W, Cheung SC, Wei J (2014) Is spreadsheet ambiguity harmful? detecting and repairing spreadsheet smells due to ambiguous computation. In: Proceedings of the 36th International Conference on Software Engineering (ICSE), ACM, pp 848–858

Fontana FA, Dietrich J, Walter B, Yamashita A, Zanoni M (2016) Anti-pattern and code smell false positives: Preliminary conceptualisation and classification. In: Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), IEEE, pp 609–613

Fowler M (1999) Refactoring: Improving the Design of Existing Code. Addison-Wesley

Garcia J, Popescu D, Edwards G, Medvidovic N (2009) Identifying architectural bad smells. In: Proceedings of the 13th European Conference on Software Maintenance and Reengineering (CSMR), IEEE, pp 255–258

Hall T, Zhang M, Bowes D, Sun Y (2014) Some code smells have a significant but small effect on faults. ACM Transactions on Software Engineering and Methodology 23(4):1–39

Han X, Tahir A, Liang P, Counsell S, Luo Y (2021) Understanding code smell detection via code review: A study of the openstack community. In: Proceedings of the 29th IEEE/ACM International Conference on Program Comprehension (ICPC), IEEE, pp 323–334

Han X, Tahir A, Liang P, Counsell S, Blincoe K, Li B, Luo Y (2022) Replication package for the paper: Code smells detection via modern code review: A study of the OpenStack and Qt communities. URL `https://doi.org/10.5281/zenodo.5588454`

Hirao T, McIntosh S, Ihara A, Matsumoto K (2020) Code reviews with divergent review scores: An empirical study of the openstack and qt communities. IEEE Transactions on Software Engineering DOI 10.1109/TSE.2020.2977907

Israel GD (1992) Determining sample size. Fact Sheet PEOD-6, Florida Cooperative Extension Service, Institute of Food and Agricultural Sciences, University of Florida, Florida, U.S.A

Kaplan EL, Meier P (1958) Nonparametric estimation from incomplete observations. Journal of the American statistical association 53(282):457–481

Kemerer CF, Paulk MC (2009) The impact of design and code reviews on software quality: An empirical study based on psp data. IEEE Transactions on Software Engineering 35(4):534–550

Khomh F, Di Penta M, Gueheneuc YG (2009) An exploratory study of the impact of code smells on software change-proneness. In: Proceedings of the 16th Working Conference on Reverse Engineering (WCRE), IEEE, pp 75–84

Kononenko O, Baysal O, Guerrouj L, Cao Y, Godfrey MW (2015) Investigating code review quality: Do people and participation matter? In: Proceedings of the 31th IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, pp 111–120

Kruskal WH, Wallis WA (1952) Use of ranks in one-criterion variance analysis. Journal of the American statistical Association 47(260):583–621

Li Z, Avgeriou P, Liang P (2015) A systematic mapping study on technical debt and its management. Journal of Systems and Software 101:193–220

Mann HB, Whitney DR (1947) On a test of whether one of two random variables is stochastically larger than the other. The annals of mathematical statistics pp 50–60

Martini A, Fontana FA, Biaggi A, Roveda R (2018) Identifying and prioritizing architectural debt through architectural smells: A case study in a large software company. In: Proceedings of the 12th European Conference on Software Architecture (ECSA), Springer, pp 320–335

McConnell S (2004) Code Complete. Pearson Education

McIntosh S, Kamei Y, Adams B, Hassan AE (2014) 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 (MSR), ACM, p 192–201

McIntosh S, Kamei Y, Adams B, Hassan AE (2016) An empirical study of the impact of modern code review practices on software quality. Empirical Software Engineering 21(5):2146–2189

Meneely A, Tejeda ACR, Spates B, Trudeau S, Neuberger D, Whitlock K, Ketant C, Davis K (2014) An empirical investigation of socio-technical code review metrics and security vulnerabilities. In: Proceedings of the 6th International Workshop on Social Software Engineering (SSE), ACM, pp 37–44

Moha N, Gueheneuc YG, Duchien L, Le Meur AF (2009) Decor: A method for the specification and detection of code and design smells. IEEE Transactions on Software Engineering 36(1):20–36

Morales R, McIntosh S, Khomh F (2015) Do code review practices impact design quality? a case study of the Qt, VTK, and ITK projects. In: Proceedings of the 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER), IEEE, pp 171–180

Nanthaamornphong A, Chaisutanon A (2016) Empirical evaluation of code smells in open source projects: preliminary results. In: Proceedings of the 1st International Workshop on Software Refactoring (IWoR), ACM, pp 5–8

Palomba F, Bavota G, Di Penta M, Oliveto R, De Lucia A (2014) Do they really smell bad? a study on developers' perception of bad code smells. In: Proceedings of the 30th International Conference on Software Maintenance and Evolution (ICSME), IEEE, pp 101–110

Palomba F, Bavota G, Di Penta M, Oliveto R, Poshyvanyk D, De Lucia A (2015) Mining version histories for detecting code smells. IEEE Transactions on Software Engineering 41(5):462–489

Palomba F, Bavota G, Di Penta M, Fasano F, Oliveto R, De Lucia A (2018) On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. Empirical Software Engineering 23(3):1188–1221

Panichella S, Zaugg N (2020) An empirical investigation of relevant changes and automation needs in modern code review. Empirical Software Engineering 25(6):4833–4872

Pascarella L, Spadini D, Palomba F, Bacchelli A (2020) On the effect of code review on code smells. In: Proceedings of the 27th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE

Pecorelli F, Palomba F, Khomh F, De Lucia A (2020) Developer-driven code smell prioritization. In: Proceedings of the 17th Working Conference on Mining Software Repositories (MSR), ACM, pp 220–231

Porter MF (2001) Snowball: A language for stemming algorithms. Open Source Initiative OSI

Potdar A, Shihab E (2014) An exploratory study on self-admitted technical debt. In: Proceedings of the 30th IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, pp 91–100

Sae-Lim N, Hayashi S, Saeki M (2018) Context-based approach to prioritize code smells for prefactoring. Journal of Software: Evolution and Process 30(6):1–24

Sharma T, Spinellis D (2018) A survey on software smells. Journal of Systems and Software 138:158–173

Sjøberg DI, Yamashita A, Anda BC, Mockus A, Dybå T (2013) Quantifying the effect of code smells on maintenance effort. IEEE Transactions on Software Engineering 39(8):1144–1156

Soh Z, Yamashita A, Khomh F, Guéhéneuc YG (2016) Do code smells impact the effort of different maintenance programming activities? In: Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), IEEE, pp 393–402

Tahir A, Counsell S, MacDonell SG (2016) An empirical study into the relationship between class features and test smells. In: Proceedings of the 23rd Asia-Pacific Software Engineering Conference (APSEC), IEEE, pp 137–144

Tahir A, Yamashita A, Licorish S, Dietrich J, Counsell S (2018) Can you tell me if it smells? a study on how developers discuss code smells and anti-patterns in stack overflow. In: Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering (EASE), ACM, pp 68–78

Tahir A, Dietrich J, Counsell S, Licorish S, Yamashita A (2020) A large scale study on how developers discuss code smells and anti-pattern in stack exchange sites. Information and Software Technology 125:106333

Taibi D, Janes A, Lenarduzzi V (2017) How developers perceive smells in source code: A replicated study. Information and Software Technology 92:223–235

Tan PN, Steinbach M, Kumar V (2016) Introduction to Data Mining. Pearson Education India

Tsantalis N, Chatzigeorgiou A (2009) Identification of move method refactoring opportunities. IEEE Transactions on Software Engineering 35(3):347–367

Tufano M, Palomba F, Bavota G, Oliveto R, Di Penta M, De Lucia A, Poshyvanyk D (2015) When and why your code starts to smell bad. In: Proceedings of the IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE), IEEE, vol 1, pp 403–414

Uchôa A, Barbosa C, Coutinho D, Oizumi W, Assunção WKG, Vergilio SR, Pereira JA, Oliveira A, Garcia A (2021) Predicting design impactful changes in modern code review: A large-scale empirical study. In: Proceedings of the 18th IEEE/ACM International Conference on Mining Software Repositories (MSR), IEEE, pp 471–482

Wessel M, Serebrenik A, Wiese I, Steinmacher I, Gerosa MA (2020) Effects of adopting code review bots on pull requests to oss projects. In: Proceedings of the 36th IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, pp 1–11

Wohlin C, Runeson P, Höst M, Ohlsson MC, Regnell B, Wesslén A (2012) Experimentation in Software Engineering. Springer

Yamashita A, Moonen L (2013) Do developers care about code smells? an exploratory survey. In: Proceedings of the 20th Working Conference on Reverse Engineering (WCRE), IEEE, pp 242–251

Zanaty FE, Hirao T, McIntosh S, Ihara A, Matsumoto K (2018) An empirical study of design discussions in code review. In: Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), ACM, pp 1–10

Zhang M, Hall T, Baddoo N (2011) Code bad smells: A review of current knowledge. Journal of Software Maintenance and Evolution: Research and Practice 23(3):179–202