# On The Effect Of Code Review On Code Smells

Luca Pascarella,[1] Davide Spadini,[1] Fabio Palomba,[2] Alberto Bacchelli[3]
[1]Delft University of Technology, The Netherlands — [2]University of Salerno, Italy — [3]University of Zurich, Switzerland
l.pascarella@tudelft.nl, d.spadini@tudelft.nl, fpalomba@unisa.it, bacchelli@ifi.uzh.ch

***Abstract*—Code smells are symptoms of poor design quality. Since code review is a process that also aims at improving code quality, we investigate whether and how code review actually influences the severity of code smells. In this study, we analyze more than 21,000 code reviews belonging to seven Java open-source projects; we find that active and participated code reviews have a significant influence on the likelihood of reducing the severity of code smells. This seems to confirm the expectations around code review's influence on code quality. However, when manually investigating a 365 cases in which the severity of code smells was reduced with a review, we found that the reduction was typically a *side effect* of changes that reviewers requested on matters unrelated to code smells.**

***Index Terms*—Code Smells; Code Review; Empirical Studies.**

## I. Introduction

Code smells are sub-optimal implementation decisions taken and applied by developers during software evolution [1]. Our research community has found compelling empirical evidence of the negative impact of code smells on software maintainability [2]–[5], program comprehensibility [6], and development effort [7], as well as on the problems that developers encounter when dealing with these smells [8]–[13].

Nevertheless, our community has also provided evidence that code smells are far from being the primary concerns for practitioners [14]–[17] and developers perform refactoring activities targeted to reduce smells [11], [18]–[20].

Morales *et al.* reported an exception to this trend [21]: They found initial evidence suggesting that *developers are concerned with code smells during code review*, as opposed to the other phases previously investigated by other researchers. In a case study of the Qt, VTK, and ITK projects, Morales *et al.* found that those Java classes that receive better code review treatment (*e.g.*, most of the changes to these classes are reviewed and more reviewers are involved) are less prone to the occurrence of code smells. Indeed, this finding is in line with previous research that has shown that developers during code reviews are not only worried about finding defects but also about general code improvements and design quality [22]. Interestingly, the existence of relationship between good/more code review and less code smells could indicate an important context around which researchers on code smells can focus their attention, for example to have the most impact with their tools on detecting and refactoring smells.

The method used by Morales *et al.* [21], however, did not allow the authors to establish any causal link between code review and code smells. Despite the authors carefully considered confounding factors in the statistical models (*e.g.*, size, complexity, ownership), it is impossible to exclude that other factors could be the actual cause of the measured relationship.

In this study, we continue on this line of work and propose a study whose goal is to investigate *the effect of code review on code smells*. We use a novel research method to reach stronger conclusions about the causality of the link between code review and code smells. In particular: (1) we consider a fine-granularity (*i.e.*, file-level, as opposed to component level) (2) compare the code smells in each file before it enters a code review and after it exits from it (as opposed to looking at aggregate values, *e.g.*, on code review coverage), and (3) manually investigate cases in which the severity of code smells does decrease in a code review.

In particular, in this study we (1) collect data of 21,879 code reviews pertaining to seven Java software projects, (2) measure whether—with each code review—the severity of any of the six selected code smells in the files under review decreases, (3) use statistical analysis to investigate the relationship between any severity decrement to the activity in the corresponding code review, (4) assess whether specific code smells are more often decreasing in severity within a code review, and (5) manually analyze 365 instances in which the severity of a code smell decreased during code review.

Our results (1) indicate that active and participated code reviews have a higher chance of reducing code smells, (2) fail to confirm a relationship between what developers reported as more critical code smells in previous studies [16], [17] and code review activity, and (3) reveal that most of the changes in code smells are not due to a direct comment on design issues, but rather to side effect of code changes requested by the reviewers on matters unrelated to code smells.

Overall, this work includes the following contributions:

1) Empirical evidence, based on a large-scale study, that if a code review is active and participated, it has the potential of reducing the impact of code smells on reviewed files;
2) Empirical evidence on the types of code smells whose severity is most likely to be reduced because of code review, which includes the discussion of contrasting results with respect to previous findings achieved in the field of code smell perception and refactoring;
3) A classification clarifying the causes that led to code smell severity decrements, whose results highlight that only rarely reviewers actively stimulate the discussion on how to make the design of a class more functional within the system;
4) An online appendix [23], including all data and scripts used to conduct our study.

## II. Background and related work

In this section, we report the background information on the topic of interest and connected related work.

### A. Related Work

Our work relates to both code review and code smells. In the following we contextualize our work with respect to literature around these two aspects.

**Code Review.** The research community has extensively investigated how code review activities, particularly participation and coverage of the process, influence source code quality. Abelein *et al.* [24] demonstrated the impact of user participation and involvement on system success. Thongtanunam *et al.* [25], [26] found that reviewing participation and expertise can decrease defect-proneness, while Rigby *et al.* [27] reported that review participation is the most influential factor influencing code review efficiency. Moreover, studies indicated that patches should be reviewed by at least two developers to maximize fault identification [28]–[31].

Bavota and Russo [32] found that the patches with a low number of reviewers tend to have a higher chance of inducing new defects. Furthermore, McIntosh *et al.* [33], [34] measured the extent to which the proportion of reviewed patches and the amount of participation in a module impact software quality, finding that a low percentage has a negative impact on code quality. Nevertheless, these findings were later partially contradicted by Sadowski *et al.* [35] in the case of Google. Ram *et al.* [36] further suggested that reviewers should be involved in the inspection of small patches to be more efficient, while Pascarella *et al.* [37] identified a set of information needs that would help developers while reviewing code.

Mäntylä and Lassenius [38] conducted a study aimed at investigating what types of defects are discovered in code review, finding that, in addition to functional defects, code reviews find many evolvability defects, (*i.e.*, issues related to the design and non-functional quality of the code). Later, Beller *et al.* [39], as well as Bacchelli and Bird [22], confirmed similar results in different contexts.

**Practitioners And Code Smells.** In parallel, researchers have investigated whether and how developers perceive code smells as an issue. Arcoverde *et al.* [40] studied how practitioners react to the presence of code smells in the source code. Their primary finding indicates that code smells tend to live in the source code for a long time because developers avoid refactoring to prevent unexpected consequences. In an industrial environment, Yamashita and Moonen [15] reported the results of an empirical study aimed at evaluating the code smell severity perceived by developers of commercial software. The authors found that 32% of the interviewed developers overlook code smells; therefore, the removal is not in their priority. Confirming this trend, Sjøberg *et al.* [7] found that code smells pose only secondary design problems, thus justifying how developers tend to ignore their presence. Opposite findings were reported by Palomba *et al.* [12], who found that the previous observation was not entirely correct when considering a subclass of textual code smells only. Developers tend to diagnose better design problems characterized by textually-detectable code smells rather than those of structurally-detectable ones. These results are aligned with a previous study where Palomba *et al.* [16] surveyed a pool of practitioners belonging to three different open source systems and reported that developers consider code smells associated with complex and lengthy source code harmful for maintainability. At the same time, whether the surveyed developers perceived the other code smells as a problem strongly depended on the severity of the problem.

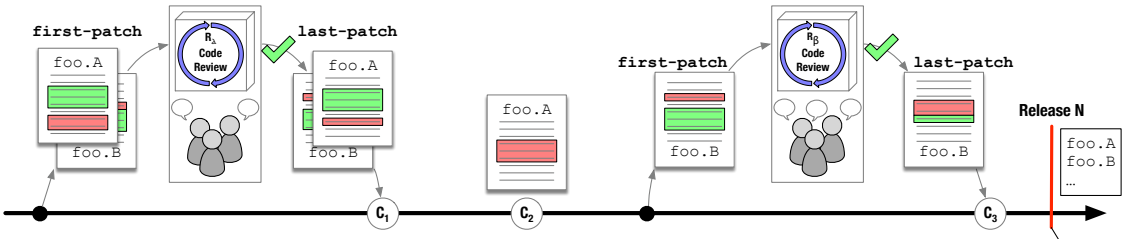
### B. Comparison with Morales et al. *[21]*

Morales *et al.* [21] are the first who investigated the relation between code review and code smells. They found that *software components with limited review coverage and participation are more prone to the occurrence of code smells* compared to components whose review process is more active.

Our study has been greatly inspired by that of Morales *et al.* [21], yet it aims to determine a more reliable causal relationship between code review and code smells, by using a novel method. In the following we detail the differences between our study and the one by Morales *et al.* [21], by considering the scenario in a software system's evolution depicted in Figure 1.

**A scenario of a software system's evolution.** In Figure 1, the horizontal line represents an interval in the evolution of a software system: This interval depicts three commits ($C_1, C_2, C_3$), two code reviews ($R_\alpha$ and $R_\beta$) and a software system's release ($N$, represented by the vertical red line). First, a developer modifies two files (`A` and `B`) that belong to the same component (`foo`) based on the current state of the main branch and submits the patch (`first-patch`) for review ($R_\alpha$). The review includes three reviewers who provide two comments over the course of an unspecified number of iterations. During the review, the patch is changed until the reviewers are eventually satisfied with the final state of the patch (`last-patch`) and allow the author to merge it into the main branch of the system (commit $C_1$). In commit $C_2$, a developer commits changes to `foo.A` to the main branch directly, without a review. Finally, a developer takes the current state of the main branch (*i.e.*, as it is after $C_2$) and modifies `foo.B`; then, the changed file (`first-patch`) is sent to review ($R_\beta$); and the final version of the file after the review loop (`last-patch`) is committed ($C_3$) to the main branch.

**Approach by Morales *et al.* [21].** Considering the aforementioned scenario, the approach by Morales *et al.* considers one of the available releases ($N$) and takes all the released source code files (in our case, we focus on `foo.A` and `foo.B`). Subsequently, the authors compute the code smells for these released files, aggregate the information at component (*i.e.*, package) level, and consider the result as their *dependent variable*. Furthermore, the authors compute control variables in the form of product (*e.g.*, lines of code, at the moment of the release) and process (*e.g.*, ownership and churn,




| Morales et al. [21] | Product metrics | | Process metrics | | Human factors | | Review Coverage | | Review Participation | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| component | Size | ... | Total Churn | ... | Total Authors | ... | % reviewed changes | % reviewed churns | % self-approved | % changes w/o comments | avg. review time | ... | # of code smells |
| foo | 70 | ... | +20 | ... | +20 | ... | 1/2 | 4/6 | 1/2 | 0/2 | 35 | ... | 2 |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |

Control variables
Independent variables
Dependent variable

| | Product metrics | | | Process metrics | | | Review Participation | | | Review File Coverage | | | Review Activity | | code smells severity | | isDecreased |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| review-file | Size | | ... | Churn | | ... | Self-Approved | Review Duration | ... | %comments | %words | ... | Comments | Words | before | after | |
| | before | after | | before | after | | | | | | | | | | | | |
| $R_\lambda$ – foo.A | 45 | 60 | ... | +15 | +20 | ... | No | 35 | ... | 1/3 | 10/80 | ... | 1 | 10 | 0.4 | 0.6 | False |
| $R_\lambda$ – foo.B | 100 | 115 | ... | +20 | +20 | ... | No | 35 | ... | 2/3 | 70/80 | ... | 2 | 70 | 0.5 | 0.5 | False |
| $R_\beta$ – foo.B | 115 | 130 | ... | +10 | +25 | ... | Yes | 10 | ... | 2/2 | 15/15 | ... | 2 | 15 | 0.7 | 0.3 | True |

Figure 1: A comparative explanation of how Morales *et al.* differ from our study.

over the history) metrics [41]. Finally, Morales *et al.* compute review metrics regarding coverage (*e.g.*, the proportion of changes that have been reviewed) and participation (*e.g.*, the proportion of changes without discussion) as independent variables. Following this approach, they consider the whole set of changes and commits that affect the evolution of the project between two releases—including the ones where developers did not perform a code review—to compute code review metrics. Although they balance this effect out with two independent variables (*i.e.*, *proportion of reviewed changes* and *proportion of reviewed churn*), this methodology allowed the authors to only obtain a coarse-grained overview of the relationship between code review practices and software design quality. For instance, in the scenario of Figure 1, changes that do not pass through code review (*e.g.*, `foo.A` in $C_2$) may increase/decrease the number of code smells independently from code review activities.

**Our proposed alternative.** In our study, we propose a novel method to better capture the causal link between code review and code smells. First of all, we consider classes (*i.e.*, files) instead of components (*i.e.*, directories), and we only considered code changes that have been inspected in code review. Then, we compute code smell severity before and after *each* code review (*i.e.*, we compute code metrics considering only changes in the `first-patch` and the `last-patch`). As depicted in Figure 1, our dependent variable is named **isDecreased**: it indicates whether the severity of a smell has decreased after the code review and is computed looking at the difference between the severity of the smell before and after a code review. Changes like the one leading to $C_2$ are not considered by our approach, because they have no direct relationship with any code review activity.

## III. Methodology

Our *goal* to investigate the causal link between code review and code smells, with the *purpose* of understanding whether the inspection activities performed by developers do reduce the severity of code smells. The *perspective* is of both researchers and practitioners, who are interested in gaining a deeper understanding of code review's effects.

### A. Research Questions

First, we investigate to what extent the severity of code smells is reduced by the code review process:

> **$RQ_1$.** How does code review influence the severity of code smells overall?

Then, we analyze whether some types of code smells exhibit a higher likelihood to be reduced with a code review. Since past studies found that developers perceive certain code smells as more relevant [16], [17], we particularly investigate whether these smells are removed with a higher likelihood.

> **$RQ_2$.** How does code review influence the severity of the different types of code smells?

Finally, we aim to qualitatively understand why the code smell severity changes with a code review (*e.g.*, do the reviewers explicitly request the author to manage code smells?):

> **$RQ_3$.** Why does the severity of code smell(s) decrease during code review?

Table I: Dataset exploited in our study.

| System | Time Span | Reviews | Reviewed Files |
|---|---|---|---|
| egit | 10-09 to 11-17 | 5,336 | 18,647 |
| jgit | 09-09 to 11-17 | 5,382 | 23,037 |
| linuxtools | 06-12 to 11-17 | 4,129 | 23,088 |
| platform.ui | 02-13 to 11-17 | 4,756 | 52,441 |
| java-client | 01-12 to 11-17 | 916 | 4,479 |
| jvm-core | 04-14 to 11-17 | 841 | 4,217 |
| spymemcached | 05-10 to 07-17 | 519 | 2,777 |
| | **Overall** | **21,879** | **128,691** |

### B. Context of the Study

To collect data for our study, we leverage CROP (Code Review Open Platform), *i.e.*, a publicly available platform and dataset developed by Paixao *et al.* [42]. This dataset is ideal for our research goals, because it (1) stores all review details, (2) contains a large amount of data on projects with different size and scope and developed by different communities, and (3) is fully compliant with the European Union General Data Protection Regulation [43] (randomly generated data replace real developers' names and email addresses).

**Subject systems.** Among the 11 software systems available in CROP, we focus on those (7) written in Java. In fact, our research community has more widespread and better validated code smells detection strategies for Java classes than for other languages (*e.g.*, those defined by Lanza and Marinescu [44]). The first two columns of Table I report information on the subject software systems, including their names and the time span we for which we have available data.

**Subject code reviews.** The dataset available in CROP includes the whole suite of patch-sets with the related discussions for each code review in the considered systems. Also, it gives easy access to the first-patch and last-patch sets (Figure 1) that represent a twofold snapshot of the source code before and after the code review. Having this information at disposal is a non-trivial advantage for our study, because we can measure the severity of code smells before and after each code review, thus we are able to address our research questions. The last two columns of Table I report the considered code review data, accounting for a total of 21,879 code reviews inspecting 128,691 files.

**Subject code smells.** We consider six different types of code smells, which are reported in Table II. We selected these code smells because: (1) they have different characteristics (*e.g.*, complexity vs. design compliance), thus allowing us to analyze a wider range of design issues that frequently occur in real software systems [2]; and (2) they have already been considered by previous work concerning their relevance for developers [15]–[17] and, therefore, can be used to assess the role of developers' perception within code review activities.

### C. Detection Of Code Smells And Their Severity

A key measure in our study is the severity's variation of the code smells in classes before they enter the review process and after they exit from it. In other words, we are interested in deriving relative measures such as the coefficient of variation for the severity between two events rather than obtaining an absolute value [46]. To that end, we first need to detect the instances of the six considered code smells. We exploit a metric-based code smell identification strategy [47] and, particularly, detection rules inspired to those defined by Lanza and Marinescu [44]: we compute object-oriented metrics [48] characterizing the classes that were subject to code review under different angles such as size, complexity, and coupling, then define thresholds that discriminate code artifacts affected (or not) by specific types of code smells.

After the code smell identification phase, we compute the code smell severity variation ($\Delta severity$) by comparing the extent to which the smelly instance exceeds the given threshold before and after a code review, as in the following formula:

$$severity(smell, class) = \frac{metricsValue_{smell}}{threshold(smell)}$$
$$\Delta severity = severity_{after\_review} - severity_{before\_review} \quad (1)$$

Negative values of $\Delta severity$ indicate that the severity of a code smell decreased as a result of the code review process, while positive values mean that it increased.

**Validation.** The detection mechanism applied can output some false positives, *i.e.*, classes erroneously classified as code smells: should the number of false positives be high, the results of our study would be compromised. To mitigate this issue, we initially select the detection rules used in previous literature [49]; this also includes the thresholds adopted to discriminate between smelly and non-smelly artifacts [50]–[53]. After selecting the detection rules, we assess the reliability of the code smell detection process by manually analyzing a subset of the code smell instances given as output of the detection. In this first iteration, we realized that the thresholds were too relaxed (resulting in too many false positives). To solve this problem, we decide to have more strict rules, *i.e.*, we increased the thresholds for the cyclomatic complexity, or LOC, etc. The optimal fine-tuning allowed us to capture slight variations of $\Delta severity$ leaving out as many as false positive. Our appendix reports the entire list of our thresholds [23].

After the first thresholds tuning, we evaluate the precision of the detector on a statistically significant sample composed of 365 code smell instances (95% statistically significant stratified sample with a 5% confidence interval) identified in the subject software projects. The validation is conducted manually by two authors of this study, who checked the source code of each class to confirm/refuse the presence of a design problem. The precision of the approach on our dataset is 92%: based on this result, we deemed the accuracy of the detector sufficiently high to conduct our study.

### D. $\mathbf{RQ}_1$. Methodology

To address $\mathbf{RQ}_1$, we investigate whether the decrement in the severity of the code smells in the classes submitted for code review vs. the same classes after the code review is influenced by metrics related to engagement and activity in

Table II: Code smells considered in our study. The values in the column "Perceived" corresponds to: Y=Yes; N=No.

| Code smell | Description | Perceived |
|---|---|---|
| Complex Class [45] | Classes exhibiting a high cyclomatic complexity, being therefore hard to test and understand [2], [6] | Y [16], [17] |
| Functional Decomposition [1] | Classes where object-oriented constructs are poorly used, declaring and implementing few methods [1]. | N [17] |
| God Class [1] | Classes that are poorly cohesive, usually characterized by large size and implementing several functionalities [1]. | Y [12], [16] |
| Inappropriate Intimacy [1] | Classes having a too high coupling with another class of the system [1]. | N [16], [17] |
| Lazy Class [1] | Classes generally composed of few lines of code and that have few relationships with other classes of the system [1]. According to Fowler [1], this represents a problem because the time on maintaining these classes might be not worthy. | N [16] |
| Spaghetti Code [1] | Classes without a structure that declare long methods without parameters [1]. | Y [16], [17] |

Table III: Independent and control variables used in our study.

| Type | Metric | Description | Motivation |
|---|---|---|---|
| | | Control variables | |
| Product | Complexity$_{before,after}$ | The Weighted Method per Class metric computed on the version of the class before and after the review. | Classes with high complexity are potential candidates to be refactored. |
| | Size$_{before,after}$ | The Lines of Code metric computed on the version of the class before and after the review. | Large classes are hard to maintain and can be more prone to be subject of refactoring [11]. |
| | PatchSize | Total number of files being subject of review. | Large patches can be more prone to be analyzed with respect to how the involved classes are designed. |
| Process | Churn$_{before,after}$ | Sum of the lines added and removed in the version of the class before and after the review. | Classes having code smells are more change-prone [2], [4]. |
| | PatchChurn | Sum of the lines added or removed in all the classes being subject of review. | Large classes are hard to maintain and can be more prone to be subject of refactoring. |
| | | Independent variables | |
| Participation | Reviewers | Number of reviewers involved in the code review. | Classes having design issues may need more reviewers to be assessed. |
| | SelfApproved | Whether the submitted patch is only approved for integration by the original author. | The original author already believes that the code is ready for integration, hence the patch has essentially not been reviewed. |
| | ReviewDuration | Time spent to perform the code review. | Classes having design issues may enforce developers to spend more time discussing refactoring options. |
| File coverage | PercentageComments | Proportion of comments on the class under review to total number of comments. | Classes receiving less attention (such as comments) may display design issues. |
| | PercentageWords | Proportion of words in the comments on the class to total number of words in all the comments. | Classes receiving less attention (such as number of words) may display design issues. |
| Activity | Comments | Number of comments made by reviewers on the class under review. | Classes having design issues can create more discussion among the reviewers on how to refactor them. |
| | Words | Number of words in the comments made by reviewers on the class under review. | Classes having design issues can create deeper discussion among the reviewers on how to refactor them. |

the code review, while controlling for other characteristics. The following subsections report the methodology adopted to define and compute dependent, independent, and control factors as well as the statistical modeling we exploited. The granularity of our model is class-level: We compute each variable for every class that enters a review and exits from it (*e.g.*, as depicted in the bottom table in Figure 1).

**Dependent Variable.** Our dependent variable is a binary variable *'isDecreased'* that takes the value 'True' if the severity of at least one code smell (existing in the file before the review) is decreased after the review and 'False' otherwise.

**Control Variables.** Some factors may affect the outcome if not adequately controlled. Since we were inspired by the study of Morales *et al.* [21], whenever we could we have used control variables as similar as possible to the ones used in the study by Morales *et al.* [21]. The first five rows of Table III describe the families of variables we use as *control variables* in our statistical model, (*i.e.*, *'Product'* and *'Process'* metrics [41]).

**Independent Variables.** Not all code reviews are done with the same care. Indeed, past research has defined proxy metrics to define engagement and participation in code review and has shown that higher values in these metrics related to better code review outcome [33]. In this study, we proceed similarly: we selected as independent variables a set of metrics to capture the engagement as well as activity during code review and we expect that code review with higher values for these features will lead to a stronger effect on code smell reduction. The second part of the rows in Table III describes the metrics that we consider as independent variables. They belong to the three categories of metrics related to code review activity and engagement: *'Participation'*, *'File coverage'*, and *'Activity'*. The variables in the *'Participation'* category are calculated at review level (*e.g.*, they are equals among all the files in the same code review), while the variables in *'File coverage'* and *'Activity'* are calculated at file level.

**Statistical Modeling.** Before building our models, we ensure that our explanatory variables are not highly correlated with one another using Spearman rank correlation tests ($\rho$). We

choose a rank correlation instead of other types of correlation (*e.g.*, Pearson) because rank correlation is resilient to data that is not normally distributed. We consider a pair of variables highly correlated when $|\rho| > 0.7$, and only include one of the pair in the model, favoring the simplest.

Once computed the variables of our study, we run *binary logistic regression* [54] of the dependent variable, where $logit(\pi_{i,j})$ represents the explained binary value (*'isDecreased'*) for a file $i$ in a review $j$, $\beta_0$ represents the log odds of being a code smell severity reduced in a review, while parameters $\beta_1 \cdot reviewers_j$, $\beta_2 \cdot comments_{i,j}$, $\beta_3 \cdot words_{i,j}$, etc. represent the differentials in the log odds of being a code smell severity reduced for a file reviewed in a review with characteristics $reviewers_{j-mean}$, $comments_{i,j-mean}$, $words_{i,j-mean}$, etc. Moreover, we run multilevel models to take into considerations that the reviewed files are nested into different projects (level 1), as neglecting a multilevel structure may lead to biased coefficients [55]. More formally:

$$\begin{aligned} logit(\pi_{i,j}) &= \beta_0 + \beta_1 \cdot reviewers_j + \\ &+\beta_2 \cdot comments_{i,j} + \beta_3 \cdot words_{i,j} + \\ &+...(other\ vars\ and\ \beta\ omitted) + (1|project) \end{aligned} \quad (2)$$

On the basis of the statistically significant codes given by the logistic regression model, we address $\mathbf{RQ}_1$ by verifying the significance of our independent variables, *i.e.*, the code review coverage, participation, and activity metrics.

### E. $RQ_2$. Methodology

In the second research question, we aim at understanding the influence of code review on specific types of code smells. We also investigate whether the types of code smell previously shown to be perceived as design problems by developers [16], [17] are treated differently. In the context of $\mathbf{RQ}_2$, we exclude all the classes in which more than one code smell was detected. This is done to account for the observation-bias that might arise when performing analyses on classes affected by more smells at the same time: indeed, the perception of a specific type of smell can be biased if another design flaw co-occurs [6], [15], [56].

We first report the number of code smells for each type (*e.g.*, *Complex Class*) in our dataset: this helps us initially observe whether any trend in the observations exists or whether we should expect that the type of smell and the reported perception of the developers do not represent relevant factors for the reduction of the severity.

Subsequently, similarly to $\mathbf{RQ}_1$, we verify if the statistical significance value given by the Chi-square test remains stable when controlling for additional factors. Therefore, we build a new binary regression model using the same dependent variable (*i.e.*, *'decreased'* or *'not decreased'*) as well as the same explanatory variables shown in Table III, to which we add a categorical variable representing the type of the smell (*e.g.*, *God Class*). Finally, we address $\mathbf{RQ}_2$ by verifying the significance of these variables: should they turn to be significant, it would indicate that the type of smell is related to the code smell severity reduction. We also analyze whether the significant smells (if any) are those reported as more important/perceived by developers when surveyed in previous literature.

### F. $RQ_3$. Methodology

To answer $\mathbf{RQ}_3$, we conduct a qualitative analysis of reviewers' discussions. We consider the code reviews in which we observe a variation in code smell severity (as done with $\mathbf{RQ}_2$, we exclude code smell co-occurrences to avoid biases in our observations). Specifically, we perform an open card sort [57] that involves two of the authors of this paper (a graduate student and a research associate - who have more than seven years of programming experience each). Hereafter, we refer to them as the *inspectors*. An open card sort is a well-established technique used to extract salient themes and it allows the definition of taxonomies from input data [57]. In our case, first we used it to organize reviewers' discussions and classify whether the variation in code smell severity is due to (i) a specific request of the reviewers (*i.e.*, a reviewer explicitly asks the author to modify the code to reduce its smelliness) or (ii) a side-effect of the reviewers' requests (*i.e.*, no smell/design issue was mentioned, thus the smelliness was reduced as a consequence of other types of comments made by the reviewers).

In the cases in which the smelliness is reduced because of an explicit request of the reviewers, we further verify whether this is due to refactoring. In remaining cases, we classify the side-effects leading to a variation of code smell severity, with the aim of understanding if there are particular reviewers' requests connected to the phenomenon of interest. To perform such an analysis, we extract a stratified and balanced sample of 365 code smell instances from our dataset. Such a sample is statistically significant (confidence interval = 5%), reflects the distribution of different code smell types, and is also balanced in the number of smells whose severity appears as reduced and not during the code review activity. Afterwards, the process consists of the two iterative sessions, conducted as described in the following.

**Iteration 1:** Initially, the two inspectors independently analyze an initial set of 50 code review threads each. They consider a code smell as perceived only if the design problems described by the reviewers are clearly traceable onto the definition of the code smell affecting the code component under review. Then, they open a discussion on the classifications made so far and reach a consensus on the names and types of both code smell symptoms and side-effects triggering a variation of the severity. As an output, this step provides a draft categorization.

**Iteration 2:** The first two inspectors re-classify the 50 initial reviewers' discussions according to the decisions taken during the debate; afterwards, they use the draft categorization to classify the remaining set of 334 code review threads. In the cases where the inspectors cannot directly apply the categories defined so far, they report such cases to the other inspectors so that a new discussion is opened. This event did not eventually happen; the inspectors could fit all

the discussions in the previously defined taxonomy, even when considering new systems. This result suggests that the categorization emerging from the first iteration reached a saturation [58], valid at least within the sample of threads considered in this study.

### G. Threats to Validity

**Construct validity.** We employed a heuristic-based code smell detector that computes code metrics and combines them to identify instances of the six considered design flaws. Such heuristics are inspired by those defined by Lanza and Marinescu [44]. In addition, to answer the $\mathbf{RQ}_3$, we manually inspected a statistically significant set of 365 code smell candidates output by the detector. The validation revealed a precision of 92%, which we deemed high enough in the context of our study to assess its suitability for our purpose. Moreover, to allow replication of our study, the detector is publicly available in our online appendix [23].

As a proxy to measure reviewers' perception of code smells ($\mathbf{RQ}_3$), we considered whether a reviewer explicitly referred to a symptom of a smell within a discussion. To reduce the subjectivity of the classification, we involved more inspectors who iteratively conducted the process and constantly discussed the classifications they made of the reviewers' discussions.

Finally, we relied on the reviewers' discussions to understand the reasons behind the variation of code smell severity. To try to mitigate the risk that developers may discuss design decisions through other channels [59], we select software systems whose developers are mostly remote [32]–[34] and that have a large number of code review data (thus indicating that developers actively use the code review platform).

**Conclusion validity.** To ensure that the selected binary logistic regression models ($\mathbf{RQ}_1$ and $\mathbf{RQ}_2$) were appropriate for the data taken into account, we performed a number of preliminary checks such as: (1) we ran a multilevel regression model [60] to account for the multilevel structure determined by the presence of data from different projects; and (2) we built the models by adding the selected independent variables step-by-step and found that the coefficients remained stable, thus further indicating little to no interference among the variables. Furthermore, in our statistical model, we control for product and process metrics, which have been shown by previous research to be correlated to code smells [2], [4].

**External validity.** We conducted our study on the code review data belonging to a set of seven systems having different size and scope. Even though this size compares well with other experiments done in code review research [21], [32]–[34], a study investigating different projects may lead to different results. Furthermore, we only considered projects written in Java (due to the lack of techniques and tools able to accurately identify code smells in different programming languages [61], [62]), thus results in other languages may vary.

## IV. Analysis of the Results

This section reports the results of the study, discussing each research question independently.

### A. $RQ_1$ The Influence Of Code Review On Code Smells

Our analysis highlights that among 128,691 reviewed files (one file could be reviewed multiple times), 89,562 (70%) were affected by at least one code smell before entering a review. The severity of the smells in these affected files is reduced only for a small fraction (4%) with code reviews. This is in line with previous findings in the field [10], [12], [63], which report how the degree of smelliness of classes tends to increase over time.

Previous research showed that code review participation, percentage of discussion, and churn are good proxies to the goodness of a code review [25], [26], [64]. As a preliminary study, we compare these code review metrics and the decrease of severity using (i) the Wilcoxon rank sum test [65] (with confidence level 95%) and (ii) Cohen's d [66] to estimate the magnitude of the observed difference. We choose the Wilcoxon test since it is a non-parametric test (it does not have any assumption on the underlying data distribution), while we interpret the results of Cohen's d relying on widely adopted guidelines, *i.e.*, negligible for $|\delta| < 0.10$, small for $0.10 \leq |\delta| < 0.33$, medium for $0.33 \leq |\delta| < 0.474$, and large for $|\delta| \geq 0.474$ [66].

From Figure 2, we see that code reviews having higher participation, coverage, and activity have a notable difference concerning the distribution of decreased and not decreased smell severity. In other words, the reviews involving a higher number of reviewers, with a higher percentage of comments, and with a higher amount of churn are those in which the severity of smells tends to decrease more often. In all the three cases, the differences observed in the distributions are statistically significant ($\alpha < 0.001$), with a *medium* effect size when considering the number of reviewers as well as $\delta$-churn, and a *small* one in the case of the percentage of comments. Thus, on the basis of the results achieved so far, we see that (i) code smells generally do not decrease in terms of severity even if they are reviewed but, as expected, (ii) the higher the quality of the code review process, the higher the likelihood to observe a reduction of code smell severity.

Table IV: Multilevel logistic regression model.

| | **Has the severity of any smell decreased?** | | |
|---|---|---|---|
| | Estimate | S.E. | Significance |
| Intercept | -2.762e+00 | 2.031e-01 | *** |
| PatchSize | -9.360e-04 | 8.489e-05 | *** |
| ReviewDuration | 7.676e-05 | 6.563e-06 | *** |
| SizeBefore | 1.111e-04 | 2.764e-05 | *** |
| ChurnBefore | 2.490e-04 | 8.812e-05 | ** |
| DeltaChurn | 1.056e-03 | 4.091e-04 | ** |
| Comments | 1.476e-02 | 6.522e-04 | *** |
| PercentageComments | 1.736e+00 | 6.716e-02 | *** |
| SelfApproved | -1.007e+00 | 6.335e-02 | *** |
| marginal $R^2$ = 0.40, conditional $R^2$ = 0.45 | | | |

*Significance codes: '***'p <0.001, '**'p <0.01, '*'p <0.05, '.'p <0.1*

After this first preliminary analysis, we now consider *all* the collected metrics (depicted in Table III) using a regression model. From Table IV we observe that all the explanatory variables, including the ones referring to the code review

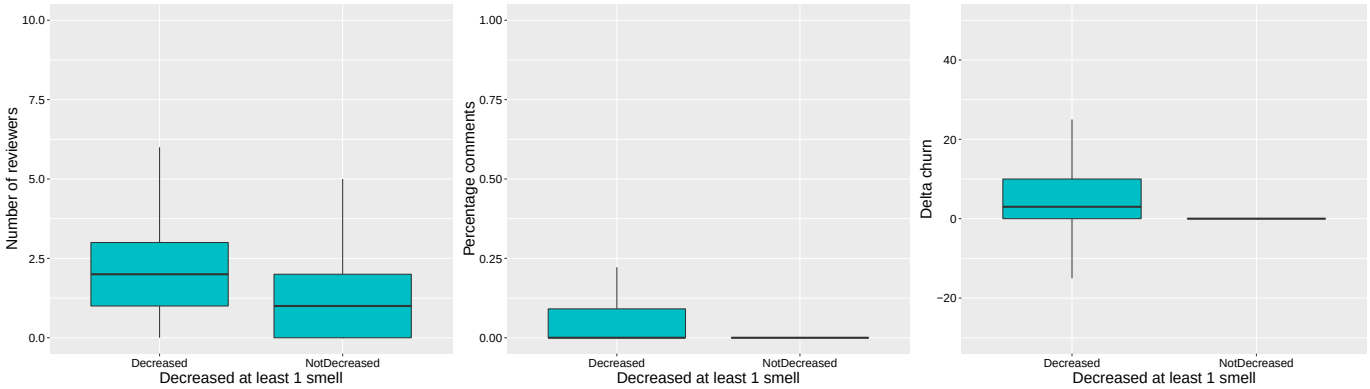



Figure 2: Number of reviewers, percentage of comments, and $\delta$-churn where code smell severity has been decreased and not.

process, appear to be statistically significant.[1] Thus, also controlling for possible confounding factors (such as the number of files reviewed and the size of the file under review), the three considered dimensions of code review represent a relevant factor to explain the reduction of code smell severity. We also evaluated the goodness of fit of our model, *i.e.*, how well the built model actually fits a set of observations. With multilevel logistic regression model, we cannot use the traditional *Adjusted* $R^2$ [67], thus we used the method proposed by Nakagawa and Schielzeth [68]: the *Marginal* $R^2$ (the variance explained by the fixed effects) measured 0.40 and the *Conditional* $R^2$ (the variance explained by the entire model) measured 0.45. These results are in line with the one reported by Morales *et al.* [21] when studying the impact of code review on design quality.

**Finding 1:** In 96% of the cases, the severity of the code smells in the files under review does not decrease. However, higher values for code review quality dimensions are indeed significantly related to a decrease in code smell severity.

### B. $RQ_2$ The Influence Of Review On Smells, By Smell Type

Figure 3 shows the distribution of decreased vs. not decreased severity, by type of smells. The Chi-square statistical test reports that the distributions are differ significantly ($p << 0.01$). The biggest Chi-square contributions come from *Inappropriate Intimacy* and *Lazy Class* classes, while the others contribute almost nothing. Particularly, with an odds ratio [69] of 2.5, *Lazy Class* is 2.5 times more likely to decrease after a code review than any other smell.

Table V reports the results of the logistic regression modeling with the added independent variable *SmellType*. We also consider whether each smell was previously reported as perceived (P) or not perceived (NP) by developers in previous studies [12], [16], [17]. Confirming the results of the aforementioned single-value analysis, *LazyClass* is statistically significant. Moreover, *Complex Class* and *God Class* are also

[1]Some of the variables in Table III are not in the statistical models: These variables were removed after we found that they were collinear with others. In cases of collinearity, we kept the variable(s) that were simpler to compute and explain, in the interest of having a simpler model.

Table V: Multilevel logistic regression model with the additional independent variable *SmellType*. P and NP specify whether the specific smell was reported as being perceived by developers in previous studies [12], [16], [17].

| | **Has the severity of the smell decreased?** | | |
|---|---|---|---|
| | Estimate | S.E. | Significance |
| Intercept | -2.889e+00 | 2.387e-01 | *** |
| PatchSize | -1.501e-03 | 1.466e-04 | *** |
| ReviewDuration | 7.782e-05 | 9.179e-06 | *** |
| SizeBefore | -2.721e-03 | 2.646e-04 | *** |
| ChurnBefore | 1.615e-03 | 2.978e-04 | *** |
| DeltaChurn | 5.765e-03 | 9.813e-04 | *** |
| Comments | 1.340e-02 | 8.019e-04 | *** |
| PercentageComments | 1.898e+00 | 1.155e-01 | *** |
| SelfApproved | -1.326e+00 | 9.992e-02 | *** |
| (P) ComplexClass | 3.724e-01 | 2.259e-01 | . |
| (P) GodClass | 3.025e+00 | 1.224e+00 | * |
| (NP) InappropriateIntimacy | -1.545e-01 | 1.699e-01 | |
| (NP) LazyClass | 6.949e-01 | 1.678e-01 | *** |
| marginal $R^2$ = 0.62, conditional $R^2$ = 0.64 | | | |

*Significance codes: '***'p <0.001, '**'p <0.01, '*'p <0.05, '.'p <0.1*

significant, though with a lower value, which is most likely due to the lower overall incidence of these two types of smells in the overall dataset (as visible in Figure 3). Moreover, we notice that there is no trend on whether the smells that should be "more perceived" by developers are those that are actually decreased more often in code review: In fact, both not perceived smells (*i.e.*, *LazyClass*) and perceived ones (*i.e.*, *Complex Class* and *God Class*) decrease with code review.

**Finding 2:** *Lazy Class*, *Complex Class*, and *God Class* are the smells whose severity is the most likely to be reduced with a code review. We find no connection between previously reported developers' perceptions on code smells, by type, and the decrement of these smell types with reviews.

### C. $RQ_3$ The Causes Of The Code Smell Severity Decrement

The findings of the previous **RQ**s showed that (i) code review participation, coverage, and activity are related to the reduction of code smell severity and (ii) it is the specific type of code smells rather than their actual perception to be relevant when explaining the reduction of severity. In this last

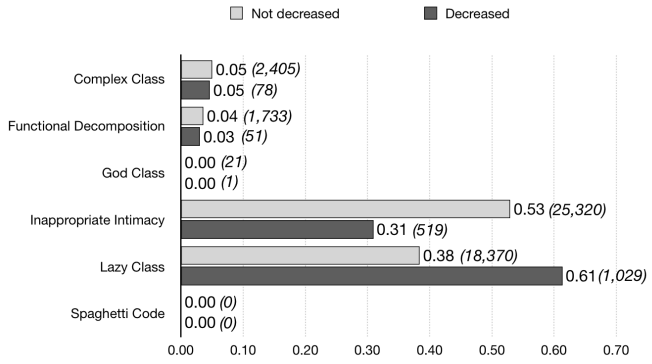



Figure 3: Decrease vs. non-decrease distribution, by smell type.

research question we aim at shedding lights on (i) the actual motivations leading code smell severity to decrease during a code review and (ii) whether instances of the *Lazy Class*, *Complex Class*, and *God Class* smells, which appeared to be the most treated ones during code review in $\mathbf{RQ}_2$, are subject to more refactoring activities than other smells.

Table VI reports the results of our manual analysis conducted on a statistically significant sample of 365 code review discussions. Our taxonomy is in line with the one proposed by Tufano *et al.* [8], who studied how code smells are removed over software evolution. We describe our findings by category.

Table VI: Code smells categories manually inspected.

| Category | Count | Percentage (%) | Ref. to smells |
|---|---|---|---|
| Code Removal | 55 | 15.1% | 0 |
| Code Replacement | 116 | 31.8% | 2 |
| Code Insertion | 152 | 41.6% | 6 |
| Refactoring | 23 | 6.3% | 8 |
| Major Restructuring | 17 | 4.7% | 3 |
| Unclear | 2 | 0.5% | 0 |
| Overall | 365 | 100% | 19 |

**CODE REMOVAL.** The discussion during the code review leads to the suppression[2] of a piece of source code (*e.g.*, a few statements or a method) that directly affects the code smell instance—as a consequence, the code smell severity decreases. Thus, code review discussions that belong to this category often create a drop in the code smell severity as a side effect rather than a conscious decision. Our manual analysis confirms that the severity decreased due to indirect actions in 55 cases that, generically, refer to the suppression of *Blob* and *Complex Class* code smells. This result is in line with the findings of Tufano *et al.* [8]: sometimes the severity of code smells decrease as a side effect of code removal.

**CODE REPLACEMENT.** The code review encourages the replacement of a certain piece of code and, consequently, the code smell (as well as the associated severity) can disappear, increase, or decrease. Typically, code review discussions in this category recommend a rewriting of the code from scratch instead of producing a refactoring of the original one. In our analysis, we observe that, in 32% of the cases, the severity of code smells varies because of code replacements, and it is a side effect rather than a direct action of the code review. Usually, reviewers' comments suggest code changes that correct defects or improve performance. Following these guidelines, the author of the patch produces a new version with the effect of re-organizing the structure of the code and decrease the severity of the code smell.

**CODE INSERTION.** In this category, a discussion during a code review suggests improving the code by adding new statements. Our manual analysis found 42% of the code review discussions falling into this category. We could observe that code insertion requests are mainly concerned with the management of *Lazy Class* instances. While the recommended operation in these cases is represented by the *Inline Class* refactoring (*i.e.*, the incorporation of the lazy class within an existing one [1]), we observed that in some cases reviewers recommend the addition of new functionalities, thus the insertion of new code. For example, this is the case for the class `CouchbaseClient` of the JAVA-CLIENT project: The reviewer explicitly requested the insertion of additional logging mechanisms that would have made the class more connected with the other classes of the system.

**REFACTORING.** In this category, we refer to code changes that are explicitly replaced by applying one or multiple refactoring operations. The few refactoring actions requested within the considered reviewers' discussions (*i.e.*, 6%) confirm the results of previous researchers [8], [9], [11], [18], [70]: Developers tend not to apply refactoring operations on classes affected by code smells. Here we see that such refactorings are not even recommended. However, in eight cases reviewers suggest action to reduce the code smell severity.

**MAJOR RESTRUCTURING.** In this category, a reviewer recommends a restructuring of the system's architecture that consequently changes the structure of the code under review. Given the nature of this category, it involves radical changes in the source code and, as such, it may implicitly include one of the above categories. However, it differs from the others since in this case we are not able to identify the exact code change leading to the smell removal. We only know that it is a consequence of a major system's restructuring. In our manual analysis we found that a major restructuring of the code under review is performed in 5% of the cases and, as a side effect, this causes the severity of smells to change.

**UNCLEAR.** This category clusters the code changes that do not fit into the categories previously defined. Only two cases from our manual analysis belong to this category.

Finally, the last column of Table VI shows the number of times a code smell is referred to during code review. As we can notice, this number is very small: 19 times out of 365 (5%). This result extends previous literature [15], [40][3] at code review time.

[2]Note that this category refers to code that is permanently removed rather than replaced by other source code.

[3]The authors found that developers avoid refactoring of code smells as it is not one of their top priorities [15], [40].

Furthermore, during manual analysis, we searched for *explicit references* to code smells, *i.e.*, "We should refactor this class before it becomes a God class". However, code smells are often symptoms of irregular trends of CK metrics (*e.g.*, code complexity, LOC, fan in or out, number of parameters, *etc.*) values. Hence, during code review, reviewers might discuss these problems (such as a complex class) without mentioning the code smell related to it (*i.e.*, God Class). In these cases, we did not consider these comments as "referencing to a code smell" because we wanted to investigate to what extent developers are aware specifically of code smells.

**Finding 3:** In 95% of the cases, the decrement in the severity of the code smells is a side effect of unrelated changes. The few cases in which reviewers explicitly suggest to reduce the code smell severity happen mostly as *Code Insertion* and *Refactoring* changes.

## V. Discussion and Implications

Our results highlighted some points to be further discussed as well as a set of practical implications for both the research community and tool vendors. In particular:

**Keep the quality of code review high.** We find that code reviews where participation, coverage, and activity are higher tend to lead to a non-targeted, yet significant reduction in code smell severity. As an outcome, we confirm the importance of the research effort devoted to the definition of methodologies and novel approaches helping practitioners keep the quality of the review process high [35], [71]. It is, thus, important when thinking of code review as a set of activities that, in addition to finding defects, include knowledge transfer or team awareness [22]. In these conditions, the presence of smells in the code might produce improper learning or misleading assumptions. Consequently, our study, in line with Pascarella *et al.* [37], revealed the need for further mechanisms to stimulate developers and make them aware of the advantages of participating and actively discussing in code review. These observations impact on educational aspects, which researchers in the code review field should further investigate and promote, but also to the definition of effective involvement strategies (*e.g.*, code review gamification [72]), which are still under-investigated by the research community.

**On the developers' awareness of code smells.** A clear outcome of our research is that the severity of code smells is often left intact or even increased during code review. This result may be a consequence that developers do not perceive code smells, that code smells cannot be properly visualized, or even that developers prefer to ignore these design issues. Moreover, our findings confirmed a problem that the research community already pointed out in the past: developers tend not to refactor or deal with code smells [11], [18]–[20] and, unfortunately, code review does not seem to change the state of things either. We believe that this outcome has implications for a broad set of researchers. First, effective and contextual code smell detectors and static analyzers [73]–[75], able to pinpoint the presence of design issues within the scope of a commit, should be made available to reviewers; as such, researchers in the field of code smells are called to devise new methods and tools allowing a just-in-time detection that can be applied at code review time. Second, we point out a problem of visualization of code smell-related information in code review: indeed, one possible reason behind the results achieved in our research is that existing code review tools are too focused on the changed lines and do not offer an overview of the classes under analysis. In other words, code review tools are yet unable to signal to reviewers the presence of design flaws. Thus, an implication of our study is that augmenting code review tools with new visualizations that support the detection and correction of code smells could be beneficial. Finally, our findings support the research on how to order code changes within code review [76], [77], which may be possibly used as a way to prioritize the analysis and correction of code smells.

**Promoting smart discussions in code review.** As both high participation and discussion within the code review process tend to promote the application of changes that have the side-effect of reducing code smell severity, a possible outcome of our research consists in calling for the definition of novel methodologies that directly address the people involved in the code review rather than the process itself. For example, novel machine learning-based methods that recognize the current context of a discussion and propose smart suggestions to lead reviewers discussing of specific design flaws that impact the maintainability of source code might be a potentially useful way to foster code quality within the code review process. Similarly, the recent advances on code documentation [78]–[80], showing and classifying the comments that help developers understanding source code, might be put in the context of code review to provide developers with further contextual information to locate the sources of technical debt.

## VI. Conclusions

In the study presented in this paper, we investigated the relation of code review to code smells by mining more than 21,000 code reviews belonging to seven Java open-source projects and considering six well-known code smells. Our findings pointed out that active and participated code reviews have a significant influence on the reduction of code smell severity. However, such a reduction is typically a side effect of code changes that are not necessarily related to code smells. The results of the study represent the input for our future research agenda, which primarily includes the definition and evaluation of non-intrusive alert systems that notify the presence of code smells at code review-time. Furthermore, we aim at replicating our study considering projects from different domains as well as closed-source projects, which may have different guidelines and/or best practices for code review activities.

## REFERENCES


[1] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[2] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, and A. De Lucia, "On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation," *Empirical Software Engineering*, pp. 1–34, 2017.

[3] S. Olbrich, D. S. Cruzes, V. Basili, and N. Zazworka, "The evolution and impact of code smells: A case study of two open source systems," in *Empirical Software Engineering and Measurement, 2009. ESEM 2009. 3rd International Symposium on*, pp. 390–400, IEEE, 2009.

[4] F. Khomh, M. Di Penta, Y.-G. Guéhéneuc, and G. Antoniol, "An exploratory study of the impact of antipatterns on class change-and fault-proneness," *Empirical Software Engineering*, vol. 17, no. 3, pp. 243–275, 2012.

[5] D. Spadini, F. Palomba, A. Zaidman, M. Bruntink, and A. Bacchelli, "On the relation of test smells to software code quality," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 1–12, IEEE, 2018.

[6] M. Abbes, F. Khomh, Y.-G. Gueheneuc, and G. Antoniol, "An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension," in *Software maintenance and reengineering (CSMR), 2011 15th European conference on*, pp. 181–190, IEEE, 2011.

[7] D. I. Sjoberg, A. Yamashita, B. C. Anda, A. Mockus, and T. Dyba, "Quantifying the effect of code smells on maintenance effort," *IEEE Transactions on Software Engineering*, no. 8, pp. 1144–1156, 2013.

[8] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk, "When and why your code starts to smell bad (and whether the smells go away)," *IEEE Transactions on Software Engineering*, vol. 43, no. 11, pp. 1063–1088, 2017.

[9] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "An empirical investigation into the nature of test smells," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pp. 4–15, 2016.

[10] R. Peters and A. Zaidman, "Evaluating the lifespan of code smells using software repository mining," in *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*, pp. 411–416, IEEE, 2012.

[11] G. Bavota, A. De Lucia, M. Di Penta, R. Oliveto, and F. Palomba, "An experimental investigation on the innate relationship between quality and refactoring," *Journal of Systems and Software*, vol. 107, pp. 1–14, 2015.

[12] F. Palomba, A. Panichella, A. Zaidman, R. Oliveto, and A. De Lucia, "The scent of a smell: An extensive comparison between textual and structural smells," *Transactions on Software Engineering*, 2017.

[13] F. Palomba, D. A. A. Tamburri, F. A. Fontana, R. Oliveto, A. Zaidman, and A. Serebrenik, "Beyond technical aspects: How do community smells influence the intensity of code smells?," *IEEE transactions on software engineering*, 2018.

[14] A. F. Yamashita and L. Moonen, "Do code smells reflect important maintainability aspects?," in *Proceedings of the International Conference on Software Maintenance (ICSM)*, pp. 306–315, IEEE, 2012.

[15] A. Yamashita and L. Moonen, "Do developers care about code smells? an exploratory survey," in *Reverse Engineering (WCRE), 2013 20th Working Conference on*, pp. 242–251, IEEE, 2013.

[16] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, and A. De Lucia, "Do they really smell bad? a study on developers' perception of bad code smells," in *Software maintenance and evolution (ICSME), 2014 IEEE international conference on*, pp. 101–110, IEEE, 2014.

[17] D. Taibi, A. Janes, and V. Lenarduzzi, "How developers perceive smells in source code: A replicated study," *Information and Software Technology*, vol. 92, pp. 223–235, 2017.

[18] D. Silva, N. Tsantalis, and M. T. Valente, "Why we refactor? confessions of github contributors," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 858–870, ACM, 2016.

[19] F. Palomba, A. Zaidman, R. Oliveto, and A. De Lucia, "An exploratory study on the relationship between changes and refactoring," in *Program Comprehension (ICPC), 2017 IEEE/ACM 25th International Conference on*, pp. 176–185, IEEE, 2017.

[20] M. Kim, T. Zimmermann, and N. Nagappan, "An empirical study of refactoring challenges and benefits at microsoft," *IEEE Transactions on Software Engineering*, vol. 40, no. 7, pp. 633–649, 2014.

[21] R. Morales, S. McIntosh, and F. Khomh, "Do code review practices impact design quality? a case study of the qt, vtk, and itk projects," in *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, pp. 171–180, IEEE, 2015.

[22] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *Proc. of the International Conference on Software Engineering (ICSE)*, pp. 712–721, IEEE, 2013.

[23] Blinded., "- online appendix - https://figshare.com/s/16e49a027434c5235474," 2018.

[24] U. Abelein and B. Paech, "Understanding the Influence of User Participation and Involvement on System Success: a Systematic Mapping Study," *Empirical Software Engineering*, vol. 20, no. 1, pp. 28–81, 2015.

[25] P. Thongtanunam, S. McIntosh, A. E. Hassan, and H. Iida, "Investigating code review practices in defective files: An empirical study of the qt system," in *MSR '15 Proceedings of the 12th Working Conference on Mining Software Repositories*, 2015.

[26] P. Thongtanunam, S. McIntosh, A. E. Hassan, and H. Iida, "Revisiting code ownership and its relationship with software quality in the scope of modern code review," in *Proceedings of the 38th international conference on software engineering*, pp. 1039–1050, ACM, 2016.

[27] P. C. Rigby, D. M. German, L. Cowen, and M.-A. Storey, "Peer review on open source software projects: Parameters, statistical models, and theory," *ACM Transactions on Software Engineering and Methodology*, p. 34, 2014.

[28] C. Sauer, D. R. Jeffery, L. Land, and P. Yetton, "The effectiveness of software development technical reviews: A behaviorally motivated program of research," *Software Engineering, IEEE Transactions on*, vol. 26, no. 1, pp. 1–14, 2000.

[29] A. Porter and L. Votta, "Comparing detection methods for software requirements inspections: A replication using professional subjects," *Empirical software engineering*, vol. 3, no. 4, pp. 355–379, 1998.

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

[31] P. Thongtanunam, S. Mcintosh, A. E. Hassan, and H. Iida, "Review Participation in Modern Code Review," *Empirical Software Engineering (EMSE)*, p. to appear, 2016.

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

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

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

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

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

[37] L. Pascarella, D. Spadini, F. Palomba, M. Bruntink, and A. Bacchelli, "Information needs in contemporary code review," *Proceedings of the ACM on Human-Computer Interaction*, vol. 2, no. CSCW, p. 135, 2018.

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

[39] M. Beller, A. Bacchelli, A. Zaidman, and E. Juergens, "Modern code reviews in open-source projects: Which problems do they fix?," in *Proceedings of the 11th working conference on mining software repositories*, pp. 202–211, ACM, 2014.

[40] R. Arcoverde, A. Garcia, and E. Figueiredo, "Understanding the longevity of code smells: preliminary results of an explanatory survey," in *International Workshop on Refactoring Tools*, pp. 33–36, ACM, 2011.

[41] F. Rahman and P. Devanbu, "How, and why, process metrics are better," in *2013 35th International Conference on Software Engineering (ICSE)*, pp. 432–441, IEEE, 2013.

[42] M. Paixao, J. Krinke, D. Han, and M. Harman, "Crop: Linking code reviews to source code changes," in *International Conference on Mining Software Repositories*, MSR, 2018.

[43] P. Voigt and A. Von dem Bussche, "The eu general data protection regulation (gdpr)," *A Practical Guide, 1st Ed., Cham: Springer International Publishing*, 2017.

[44] M. Lanza and R. Marinescu, *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer, 2006.

[45] W. H. Brown, R. C. Malveau, H. W. McCormick, and T. J. Mowbray, *AntiPatterns: refactoring software, architectures, and projects in crisis*. John Wiley & Sons, Inc., 1998.

[46] T. L. Saaty, "What is relative measurement? the ratio scale phantom," *Mathematical and Computer Modelling*, vol. 17, no. 4-5, pp. 1–12, 1993.

[47] F. Pecorelli, F. Palomba, D. Di Nucci, and A. De Lucia, "Comparing heuristic and machine learning approaches for metric-based code smell detection," in *Proceedings of the 27th International Conference on Program Comprehension*, pp. 93–104, IEEE Press, 2019.

[48] S. Chidamber and C. Kemerer, "A metrics suite for object oriented design," *IEEE Trans. on Software Engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[49] E. Fernandes, J. Oliveira, G. Vale, T. Paiva, and E. Figueiredo, "A review-based comparative study of bad smell detection tools," in *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*, p. 18, ACM, 2016.

[50] J. García-Munoz, M. García-Valls, and J. Escribano-Barreno, "Improved metrics handling in sonarqube for software quality monitoring," in *Distributed Computing and Artificial Intelligence, 13th International Conference*, pp. 463–470, Springer, 2016.

[51] R. Ferenc, L. Langó, I. Siket, T. Gyimóthy, and T. Bakota, "Source meter sonar qube plug-in," in *Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on*, pp. 77–82, IEEE, 2014.

[52] M. Lopez and N. Habra, "Relevance of the cyclomatic complexity threshold for the java programming language," *SMEF 2005*, p. 195, 2005.

[53] F. A. Fontana, V. Ferme, M. Zanoni, and R. Roveda, "Towards a prioritization of code debt: A code smell intensity index," in *2015 IEEE 7th International Workshop on Managing Technical Debt (MTD)*, pp. 16–24, IEEE, 2015.

[54] G. G. Judge, R. C. Hill, W. Griffiths, H. Lutkepohl, and T. C. Lee, "Introduction to the theory and practice of econometrics.," 1982.

[55] W. S. Robinson, "Ecological correlations and the behavior of individuals," *International journal of epidemiology*, vol. 38, no. 2, pp. 337–341, 2009.

[56] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, and A. De Lucia, "A large-scale empirical study on the lifecycle of code smell co-occurrences," *Information and Software Technology*, vol. 99, pp. 1–10, 2018.

[57] H. E. Nelson, "A modified card sorting test sensitive to frontal lobe defects," *Cortex*, vol. 12, no. 4, pp. 313–324, 1976.

[58] D. Finfgeld-Connett, "Use of content analysis to conduct knowledge-building and theory-generating qualitative systematic reviews," *Qualitative Research*, vol. 14, no. 3, pp. 341–352, 2014.

[59] A. Guzzi, A. Bacchelli, M. Lanza, M. Pinzger, and A. v. Deursen, "Communication in open source software development mailing lists," in *Proceedings of the 10th Working Conference on Mining Software Repositories*, pp. 277–286, IEEE Press, 2013.

[60] S. W. Raudenbush and A. S. Bryk, *Hierarchical linear models: Applications and data analysis methods*, vol. 1. Sage, 2002.

[61] E. V. de Paulo Sobrinho, A. De Lucia, and M. de Almeida Maia, "A systematic literature review on bad smells—5 w's: which, when, what, who, where," *IEEE Transactions on Software Engineering*, 2018.

[62] M. I. Azeem, F. Palomba, L. Shi, and Q. Wang, "Machine learning techniques for code smell detection: A systematic literature review and meta-analysis," *Information and Software Technology*, 2019.

[63] A. Chatzigeorgiou and A. Manakos, "Investigating the evolution of bad smells in object-oriented code," in *International Conference on the Quality of Information and Communications Technology*, pp. 106–115, IEEE, 2010.

[64] D. Spadini, M. Aniche, M.-A. Storey, M. Bruntink, and A. Bacchelli, "When testing meets code review: Why and how developers review tests," in *Software Engineering (ICSE), 2018 IEEE/ACM 40th International Conference on*, p. to appear, 2018.

[65] W. J. Conover, *Practical Nonparametric Statistics*. Wiley, 3rd edition ed., 1998.

[66] R. J. Grissom and J. J. Kim, *Effect sizes for research: A broad practical approach.* Lawrence Erlbaum Associates Publishers, 2005.

[67] K. Ohtani, "Bootstrapping r2 and adjusted r2 in regression analysis," *Economic Modelling*, vol. 17, no. 4, pp. 473–483, 2000.

[68] S. Nakagawa and H. Schielzeth, "A general and simple method for obtaining r2 from generalized linear mixed-effects models," *Methods in Ecology and Evolution*, vol. 4, no. 2, pp. 133–142, 2013.

[69] J. M. Bland and D. G. Altman, "The odds ratio," *Bmj*, vol. 320, no. 7247, p. 1468, 2000.

[70] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 5–18, 2012.

[71] P. Thongtanunam, C. Tantithamthavorn, R. G. Kula, N. Yoshida, H. Iida, and K.-i. Matsumoto, "Who should review my code? a file location-based code-reviewer recommendation approach for modern code review," in *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, pp. 141–150, IEEE, 2015.

[72] N. Unkelos-Shpigel and I. Hadar, "Gamifying software engineering tasks based on cognitive principles: The case of code review," in *Cooperative and Human Aspects of Software Engineering (CHASE), 2015 IEEE/ACM 8th International Workshop on*, pp. 119–120, IEEE, 2015.

[73] J. Pantiuchina, G. Bavota, M. Tufano, and D. Poshyvanyk, "Towards just-in-time refactoring recommenders," in *Proceedings of the 26th Conference on Program Comprehension*, pp. 312–315, ACM, 2018.

[74] N. Sae-Lim, S. Hayashi, and M. Saeki, "Context-based code smells prioritization for prefactoring," in *Program Comprehension (ICPC), 2016 IEEE 24th International Conference on*, pp. 1–10, IEEE, 2016.

[75] C. Vassallo, S. Panichella, F. Palomba, S. Proksch, A. Zaidman, and H. C. Gall, "Context is king: The developer perspective on the usage of static analysis tools," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 38–49, IEEE, 2018.

[76] T. Baum, K. Schneider, and A. Bacchelli, "On the optimal order of reading source code changes for review," in *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*, pp. 329–340, IEEE, 2017.

[77] D. Spadini, F. Palomba, T. Baum, S. Hanenberg, M. Bruntink, and A. Bacchelli, "Test-driven code review: An empirical study," in *2019 IEEE/ACM International Conference on Software Engineering (ICSE)*, p. in press, IEEE, 2019.

[78] L. Pascarella and A. Bacchelli, "Classifying code comments in java open-source software systems," in *Proceedings of the 14th International Conference on Mining Software Repositories*, pp. 227–237, IEEE Press, 2017.

[79] L. Pascarella, M. Bruntink, and A. Bacchelli, "Classifying code comments in java software systems," *Empirical Software Engineering*, pp. 1–39, 2019.

[80] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, "Towards automatically generating summary comments for java methods," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pp. 43–52, ACM, 2010.