How Does Modern Code Review Impact Software Design Degradation? An In-depth Empirical Study

Anderson Uchôa*, Caio Barbosa*, Willian Oizumi*, Publio Blenilio†,
Rafael Lima†, Alessandro Garcia*, Carla Bezerra†
*Informatics Department, Pontifical Catholic University of Rio de Janeiro (PUC-Rio), Brazil
Email: {auchoa, csilva, woizumi, afgarcia}@inf.puc-rio.br
†Campus Quixadá, Federal University of Ceará (UFC), Brazil
Email: {publioufc, rgllima}@alu.ufc.br, carlailane@ufc.br

***Abstract*—Software design is an important concern in modern code review through which multiple developers actively discuss and improve each single code change. However, there is little understanding of the impact of such developers' reviews on continuously reducing design degradation over time. It is even less clear to what extent and how design degradation is reversed during the process of each single code change's review. In summary, existing studies have not assessed how the *process of design degradation evolution* is impacted along: (i) *within each single review*, and (ii) *across multiple reviews*. As a consequence, one cannot understand how certain *code review practices* consistently contribute to either reduce or further increase design degradation as the project evolves. We aim at addressing these gaps through a multi-project retrospective study. By investigating 14,971 code reviews from seven software projects, we report the first study that characterizes how the process of design degradation evolves within each review and across multiple reviews. Moreover, we analyze a comprehensive suite of metrics to enable us to observe the influence of certain code review practices on combating or even accelerating design degradation. Our results show that the majority of code reviews had little to no design degradation impact in the analyzed projects. Even worse, this observation also applies, to some extent, to reviews with an explicit concern on design. Surprisingly, the practices of *long discussions* and *high proportion of review disagreement* in code reviews were found to increase design degradation. Finally, we also discuss how the study findings shed light on how to improve the research and practice of modern code review.**

***Index Terms*—code review, software design degradation, code review practices**

I. Introduction

Modern code review is a lightweight, informal, asynchronous, and tool-assisted technique aimed at detecting and removing issues that were introduced during development tasks [1]. Both industrial [2] and open-source [3] projects have been adopting modern code review on a daily basis as a means to promote the quality of their software systems [1]. Along with code reviews, developers inspect and discuss the quality of each other's code changes before accepting them.

Modern code review may play a key role at both improving the design quality of a software as well as its maintainability [4]–[6]. Previous studies [1], [7], [8] suggest that certain code review practices, such as the lack of review participation, may increase design degradation. Other studies have shown that, along with code review, developers often argue about software maintainability and suggest design improvements to the code owners [4], [9], [10].

Code review may or may not be explicitly focused on design quality [4], [11], [12]. Unfortunately, even with explicit design discussions, changes performed by developers along reviews can lead to design degradation. Design degradation is the process where design decisions progressively worsen the structural quality of a system, thereby also hampering external quality attributes such as maintainability. If not properly avoided, identified and combated, design degradation has severe consequences to the software health and also possibly contributing to its (dis)continuation in the future [13]–[18].

Existing studies tend to analyze design degradation considering only single events, such as the introduction of a single design problem [18], [19], or simply analyzing the degradation frequency [8], [20]–[22]. Nevertheless, understanding how the design degradation evolves over time – across reviews and within reviews – is of paramount importance. Otherwise, we are misinforming the research and practice of modern code review. Since the code review also aims to improve design quality, one could expect that, over time, the reviews will gradually reduce multiple degradation symptoms.

To the best of our knowledge, there is no study that performs an in-depth investigation of the impact of modern code review – and its practices – on the design degradation evolution. Hence, it remains unclear whether and to what extent code review helps to combat design degradation. Moreover, there is little knowledge about the impact of developers' design discussions on degradation. Additionally, we do not know which practices may strengthen the combat or the acceleration of design degradation.

This paper addresses the aforementioned limitations through an in-depth empirical study that characterizes the impact of modern code review and its practices on design degradation evolution. To this end, we retrospectively investigate 14,971 code reviews from seven software systems pertaining to two large open source communities. We analyze the characteristics of design degradation across reviews and within reviews. Moreover, we assess how reviews with design discussion tend to impact design degradation. Finally, we analyze a comprehensive suite of metrics to support our observations regarding the relationship between certain code review practices and the

2576-3148/20/$31.00 ©2020 IEEE
DOI 10.1109/ICSME46990.2020.00055

combat (or amplification) of design degradation.

Our contributions include: (i) findings on the characterization of the code review impact on design degradation, (ii) findings on how design degradation evolves along with code reviews, and (iii) statistical analyses concerning the relationship between certain code review practices and design degradation. We summarize our study findings as follows:

1) When developers have an explicit concern with design, the effect on design degradation is usually positive or invariant. However, the sole presence of design discussions is not a decisive factor to avoid degradation;
2) During the revisions of each single review, there is often a wide fluctuation of design degradation. This fluctuation means that developers are both introducing and removing symptoms along a single code review. However, at the end of the review, such fluctuations often result in the amplification of design degradation, even in the context of reviews with an explicit design concern;
3) Certain code review practices increase the risk of design degradation, including long discussions and a high rate of reviewers' disagreement. The finding on long discussions shows that long discussions are introducing more than removing degradation symptoms.

II. Background and Related Work

A. Design Degradation

Design degradation is a phenomenon in which developers progressively introduce design problems in a system [23]. The degradation is caused by design decisions that negatively impact quality attributes such as maintainability and extensibility [13], [18]. An example of degradation is when a class is overloaded with multiple unrelated functionalities, making it difficult to use and increasing the chances of causing ripple effects on other classes. Given the potential harmfulness of design degradation, developers need to identify and refactor source code locations impacted by design degradation.

Empirical studies on design degradation. There are multiple studies about design degradation [17]–[21], [24]–[27]. Oizumi et al. [21], for example, investigated if degradation symptoms appear with higher density and diversity in classes refactored by developers. The authors observed that despite not being removed by refactorings, some types of symptoms might be indeed strong indicators of design problems. Ahmed et al. [26] analyzed how open source projects get worse in terms of design degradation. The authors identified strong evidence that the density of design problems build up over time.

None of the aforementioned studies have analyzed how code changes performed by developers during code reviews impact on design degradation. In this work, we fill this gap in the literature by investigating two categories of symptoms, which are fine-grained (FG) and coarse-grained (CG) smells [28]. *FG smells* are indicators of structural degradation in the scope of methods and code blocks [28]. For instance, the Long Method is a FG smell that occurs in methods that contain too many lines of code. This smell usually indicates modifiability and comprehensibility problems. *CG smells* are symptoms that may indicate structural degradation related to object-oriented principles such as abstraction, encapsulation, modularity, and hierarchy [28], [29]. An example of CG smell is Insufficient Modularization [28]. This symptom occurs in classes that are large and complex due to the accumulation of responsibilities.

B. Modern Code review

Modern code review is typically a lightweight, informal, asynchronous, and tool-assisted practice aimed at detecting and removing issues that were introduced during development tasks [1]. Major companies, such as Facebook [30] and Microsoft [1] use modern code review on a daily basis. Supported by tools such as Gerrit, the modern code review process is initiated by one developer referred to as the *code owner* that *modifies the original codebase* and *submits a new code change* to be reviewed. These code changes are reviewed by other developers, i.e., *code reviewers*, that will inspect it [31]. The code reviewers *inspect the code change* to detect issues such as bugs, design problems, and violations of style [9], [32].

After that, the code reviewers *provide their review feedback*, in the form of code review comments, to the code owner. In turn, the code owner applies fixes and forwards the new version of the source code for inspection, which can be followed by another code review comment. This cycle is iterative and ends up with either the acceptance or rejection of the integration of the change into the codebase [9], [31].

In our study, we use *review* to indicate the entire process of a single code review, from submitting a new code change for review to approving or rejecting the integration of the change into the codebase. In addition, we use *revision* to indicate each iteration of this process during a single review.

Empirical studies on the impact of modern code review. Multiple studies have investigated the impact of code review on software quality [4], [7], [8], [11], [33]. Morales et al. [7] investigated the relation between code review and code smells. The authors observed 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. Pascarella et. al [8] investigated if and how code review influences the severity of six types of code smells in seven Java open-source projects. The authors observed that active and participative code reviews have a significant influence on the reduction of code smell severity. Another study [11] has investigated the impact of code review on the structural high-level design. The authors observed that only 31% of the reviews with design discussions have a noticeable impact on the structural high-level design. In this work, we investigate the same set of systems analyzed by them. Nevertheless, we focused on assessing the impact of modern code reviews on design degradation. In addition, we conducted multiple new analysis, as summarized below.

In a nutshell, our work differs from the existing ones in the following points: (1) we investigated how the occurrence of design discussions during a review may affect the evolution of design degradation; (2) while most studies are focused in

analyzing the degradation as single events, we investigated the manifestation and evolution of the design degradation process (increase and reduction) under different aspects, which are across reviews and along with revisions of each review; and (3) we used a multiple logistic regression model to evaluate the impact of code review practices on design degradation.

III. Motivating Example

We adopt review 53,827 [34] from the jgit system to motivate our study and depict the phenomenon we investigate. The goal of the reviewed task was to "delete non empty directories before checkout a path". The task was developed by one developer and reviewed by two reviewers. As showed in Figure III, through the course of this particular review, different aspects of the code change have been discussed. The main focus of the discussions was related to the functional requirements. Reviewers were concerned, e.g., with possible side effects that the change could introduce. Besides that, there was also great concern about automated testing.

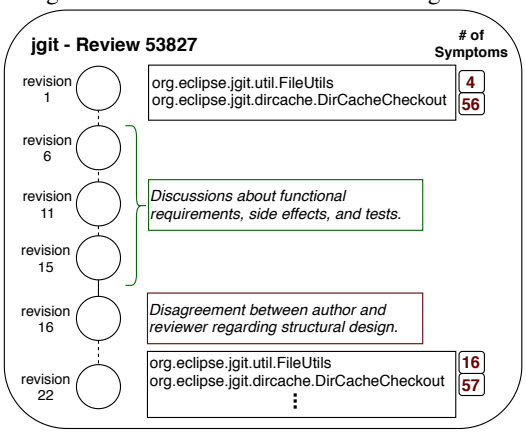

Fig. 1. Example of a code review that introduced degradation symptoms.

Only after 16 revisions, there was a comment about structural design. The reviewer complained about the use of a boolean parameter in the method *checkoutEntry* from the *DirCacheCheckout* class. However, the author disagreed with the reviewer's comment, arguing that there would be no problem with the use of the boolean parameter. After that, the reviewer said that he would not insist, implying that he continues to disagree with the design decision being discussed. After that, no other comments regarding the structural design were made by the reviewers. As a possible consequence of this disagreement between those involved, we observed that many symptoms of potential degradation were ignored.

For example, we observed several occurrences of a fine-grained smell called *Magic Number*. This type of smell occurs when literal numbers are used in the code. The use of a literal number in code structures – such as if statements and assignments – are not advisable because it does not make explicit what the number really means. Instead, the recommended practice is to use constants or enumerations that make the meaning of numbers explicit. Instead of using the recommended practice, the author chose to comment on the code with an explanation of the meaning of the numbers involved. Although the use of comments is a valid approach, it could be combined with the use of constants or enumerations to obtain a higher quality design and to prevent the same number, and its respective explanatory comment, from having to be repeated in several parts of the code. This is a problem that could be identified and removed during code review.

In addition to not seeing possible degradation in the changed code, new degradation symptoms emerged throughout the review. For instance, new occurrences of *Long Statement*, *Complex Method*, *Empty Catch Clause*, and *Magic Number* list were introduced in methods of the *FileUtils* class. Moreover, other classes that were changed also presented more symptoms of possible degradation. Thus, it would be important to effectively assess the design of changed classes during code review as design problems may arise or become more severe.

IV. Study Settings

A. Research Questions

RQ_1: *To what extent do modern code reviews impact design degradation?* **RQ_1** aims at providing evidence on the impact of code reviews on the evolution of design degradation. To achieve this goal, we focus on exploring the evolution of two degradation characteristics: density and diversity of symptoms. We analyze such characteristics in the context of two categories of degradation symptoms, which are the fine-grained and coarse-grained smells. Finally, we analyze the impact on design degradation caused by two code review factors. The first factor is the presence of explicit intent of improving the design. The second one is the presence of explicit design discussions along with the revisions of a review. We provide more details about such factors in Section IV-B.

RQ_2: *How does design degradation evolve along with each code review?* **RQ_2** aims at investigating how degradation characteristics evolve along with the revisions that occur along with each code review. To answer **RQ_2**, we identified and investigated four different evolution patterns for degradation characteristics (i.e., density and diversity). Such investigation provides us with new insights about the evolution of design degradation throughout the reviewing process.

RQ_3: *How do code review practices influence design degradation?* **RQ_3** aims at exploring in depth the relationship of different code review practices with the evolution of degradation characteristics. A correlation between these two variables may evidence that certain code review practices can be used as indicators of increased design degradation. Also, by answering **RQ_3**, we will be able to reveal whether according to previous studies [5], [8], [35], [36] code reviews that are intensely scrutinized, with more team participation, and reviewed for a longer time, usually has a positive effect on design degradation.

B. Study Steps and Procedures

Step 1: Select software systems that adopt modern code review. We selected systems provided by the Code Review Open Platform (CROP) [37], an open-source dataset that links code review data with their respective code changes. CROP

currently provides data for 11 systems, extracted from two large open source communities: Eclipse and Couchbase. All systems in CROP employ Gerrit as their code review tool. Hence, by using CROP, we have access to a rich dataset of source code changes that goes beyond other platforms, such as Github. We selected only Java systems included in the CROP dataset due to the limitations of the DesignateJava tool [38] (see Step 2). We considered only merged reviews, since they represent changes that were integrated into the systems. In addition, we discarded reviews that did not change Java files. Table I provides details about each selected system, where the Eclipse and Couchbase systems are presented in the upper and bottom half of the table, respectively. We also detail the number of merged reviews and revisions in each system, followed by the time-span of our investigation.

TABLE I
SOFTWARE SYSTEMS INVESTIGATED IN THIS STUDY

Systems	# of Reviews	# of Revisions	Time span
jgit	3,736	10,718	10/09 to 11/17
egit	3,607	9,937	9/09 to 11/17
platform.ui	3,072	10,282	20/13 to 11/17
linuxtools	2,947	9,149	6/12 to 11/17
java-client	642	2,064	11/11 to 11/17
jvm-core	629	1,851	4/14 to 11/17
spymemcached	338	1,010	5/10 to 7/17

Step 2: Detect degradation symptoms during code review. We used the DesigniteJava tool [38] to detect a total of 27 degradation symptoms types: 17 coarse-grained (CG) smells, and 10 fine-grained (FG) smells. Hence, for each system under study, we identified these degradation symptoms by considering each review and submitted revisions that have undergone the code review process. For each submitted revision, we used CROP to access the versions of the system before and after the revision took place. Hence, we guaranteed that the introduced degradation symptoms between each version were solely introduced by the code changes in the revisions. Table II lists the 27 symptoms types investigated in our study, where the CG and FG smells are presented in the upper and bottom half of the table, respectively. We provide all descriptions, detection strategies, and thresholds for each type of symptom in our replication package [39].

TABLE II
DEGRADATION SYMPTOMS INVESTIGATED IN THIS STUDY

Coarse-grained Smells
Imperative Abstraction, Multifaceted Abstraction, Unutilized Abstraction, Unnecessary Abstraction, Deficient Encapsulation, Unexploited Encapsulation, Broken Modularization, Insufficient Modularization, Hub Like Modularization, Cyclic Dependent Modularization, Rebellious Hierarchy, Wide Hierarchy, Deep Hierarchy, Multipath Hierarchy, Cyclic Hierarchy, Missing Hierarchy, Broken Hierarchy.
Fine-grained Smells
Abstract Function Call From Constructor, Complex Conditional, Complex Method, Empty Catch Block, Long Identifier, Long Method, Long Parameter List, Long Statement, Magic Number, Missing Default.

Step 3: Compute degradation characteristics for each symptom category during code review. We rely on an existing grounded theory [18] that explains that developers tend to consider multiple degradation characteristics. We take into account two characteristics, namely *density* and *diversity*, as metrics to measure the level of design degradation. For each selected system, we computed these characteristics in the context of each symptom category (CG and FG smells), for all the collected reviews and revisions. For each review and revision, we also used CROP to access the versions of the system before and after each review and revision. *Density* was computed for each version of the system, before and after each revision, as the sum of the number of symptom instances in the set of source code files. Similarly, we computed the *diversity* as a sum of the number of different symptom types in the set of source code files.

The computation of density and diversity before and after revisions, allowed us to generate four different indicators of design degradation for each revision, where each indicator represents the differences in density and diversity of FG and CG smells. In summary, when the degradation characteristic, either density or diversity, after the revision, is larger than the characteristic before the revision there is an *increase in the degradation* as a result of the revision. Similarly, when the degradation characteristic after the revision is smaller than the characteristic before, there is a *reduction of the degradation* as a result of the revision. In total, we have computed the four indicators for 14,971 code reviews and 45,011 revisions.

Step 4: Identify design degradation evolution patterns across reviews. We identified the design degradation evolution across reviews by adapting a recent state-of-the-art classification provided by a previous work [11]. To find evolution patterns, we considered only reviews, identified, and filtered in Step 3, which presented an increase or decrease of degradation. For this purpose, we considered reviews that: (i) have more than one revision, and (2) present symptoms of degradation. We firstly identified the last merged revision of each review, which represents the degradation evolution that was, in fact, incorporated into the system. After that, we compared the degradation characteristics of the last merged revision with all the other previous revisions of each code review. This procedure enabled us to investigate how design degradation evolves across the revisions of each review.

Step 5: Calculate code review activity metrics. Table III shows the 16 metrics that we used to measure the code review activity. The first part of Table III describes the control variables that we computed to avoid some factors that may affect our outcome if not adequately controlled. As control variables, we used *Product* and *Process* metrics, which have been shown by previous research to be correlated with design degradation [40], [41]. The second part of Table III describes the metrics that we considered as independent variables to measure the code review activity. We have grouped each metric in three dimensions. *Review Intensity* measures the scrutiny that was applied during the code review process. *Review Participation* measures how much the development team invests in the code review process. Finally, *Reviewing Time* measures the duration of a code review. We emphasize that these metrics are extensively used by previous works (e.g., [5], [7], [36]) to measure the code review activity. Moreover, all three dimensions investigated in our study suggest practices that may be favorable or not to combat design degradation.

Step 6: Assess the influence of multiple code review practices on software degradation. To assess the influence

TABLE III
INDEPENDENT AND CONTROL VARIABLES USED IN OUR STUDY. THE RATIONALE OF EACH METRIC IS DESCRIBED IN OUR REPLICATION PACKAGE [39]

Type	Metric	Description
	Control variables	
Product	DiffComplexity (DC)	The difference of the sum of the Weighted Method per Class metric computed on the version before and after review of all classes being subject of review.
	DiffSize (DS)	The difference of the sum of the Lines of Code metric computed on version before and after review of all classes being subject of review.
	Patch Size (PS)	Total number of files being subject of review.
Process	Patch Churn (PC)	Sum of the lines added and removed in all the classes being subject of review.
	Independent variables	
Review Intensity	Number of Revisions (NR)	The number of revisions for a patch prior to its integration.
	Discussion Length (DL)	Number of general comments and inline comments written by reviewers.
	Proportion of Revisions without Feedback (PRWF)	The proportion of iterations without discussions started by a reviewer, neither posting a message nor a score.
	Churn during Review (CDCR)	Number of lines that were added and deleted between revisions.
Review Participation	Number of Reviewers (NR)	Number of developers who participate in a code review, i.e., posting a general comment, or inline comment, and assigning a review score.
	Number of Authors (NA)	Number of developers who upload a revision for proposed changes. Changes revised by many authors may introduce more degradation into the system [42], [43].
	Number of Non-Author Voters (NNAV)	Number of developers who assign a review score, excluding the patch author.
	Proportion of Review Disagreement (PRD)	A proportion of reviewers that vote for a disagreement to accept the patch, i.e., assigning a negative review score.
Reviewing Time	Review Length (RL)	Time in days from the first patch submission to the reviewers' acceptance for integration [44], [45].
	Response Delay (RD)	Time in days from the first patch submission to the posting of the first reviewer message [44].
	Average Review Rate (ARR)	Average review rate (KLOC/Hour) for each revision.
	Typical Review Window (TRW)	The length of time between the creation of a review and its final approval for integration, normalized by the size of the change.

of the code review activity metrics in design degradation, we created a statistical model using the *multiple logistic regression* technique. In this model, we used all code review metrics presented in Table III as predictors of the likelihood of a code review to impact design degradation; i.e., whether each code review has either a decreasing or increasing impact in the degradation characteristics. We used *multiple logistic regression* because we are dealing with multiple possible predictors, and we have a binary variable as a response. To avoid the effect of *multicollinearity* on our data, we remove the code review metrics which have a pair-wise correlation coefficient above 0.7 [46]. Moreover, we used odds ratios to report the effect of the metrics over the possibility of a review impacting design degradation. Odds ratios are the increase or decrease in the odds of a review degradation impact occurring per "unit" value of a predictor (metric). An odds ratio < 1 indicates a decrease in these odds, while > 1 indicates an increase. Most of our metrics presented a heavy skew, to reduce it, we applied a $\log_2$ transformation on the right-skewed predictors and a x^3 transformation on the left-skewed. Furthermore, we normalized the continuous predictors in the model to provide normality. As a result, the mean of each predictor is equaled to zero, and the standard deviation to one. Finally, to ensure the statistical significance of the predictors, we employed the customary *p-value* of 0.05 for each predictor in the regression model.

Step 7: Manually analysis and classify reviews. In this step, we used a subset of code reviews that were manually classified in the work of Paixao et al. [11]. We performed a cross-check analysis of such reviews considering the commit messages, discussions between developers, and source code. We also filtered the reviews according to the criteria presented in Step 1, which resulted in a subset of 1,779 manually analyzed reviews. Based on this analysis, we conducted two classifications. In the first one, we classified reviews as *design-related* or *design-unrelated*, according to the developers' intent of improving the structural design of the system. Reviews were tagged as *design-related* when design improving intent was explicit either in the review's *descriptions* or in *discussions*. The second classification consisted in identifying reviews in which explicit design discussions occurred. We considered as *reviews with design discussions* those in which developers have demonstrated awareness of the possible impact of their changes in the system's design.

We performed such classifications according to the following definition of design: *software design is the result of design decisions that affect structural quality attributes, either positively or negatively*. The manual classification process was performed by two authors. Each author was responsible for analyzing the 1,779 code reviews and manually classifying them. We employed a two-phase process: 1) Each author solely and separately inspected and classified the same code reviews; 2) the author discussed all the reviews for which there was a disagreement in the classification until a consensus is reached.

All the data collected in the aforementioned steps as well as the set of manually classified code reviews are available in our replication package [39].

V. RESULTS AND DISCUSSIONS

A. Manifestation of Design Degradation

We address $\mathbf{RQ}_1$ by analyzing the impact of merged reviews on two degradation characteristics: (1) density and (2) diversity of symptoms. Table IV shows the frequency of each type of impact in all target systems. Columns represent the symptom characteristics. We decomposed those characteristics into four distinct groups: Density of Coarse-grained Smells (CG Density), Density of Fine-grained Smells (FG Density), Diversity of Coarse-grained Smells (CG Diversity), and Diversity of Fine-grained Smells (FG Diversity). They represent the amount and heterogeneity of degradation symptoms at different granularity levels. We also categorized the impact of reviews into positive, negative, and invariant. **Positive** are those that end up reducing the degradation characteristic, while **negative** ones are those that contribute to increasing the degradation characteristic. Finally, **invariant** reviews are those that do not affect the degradation characteristic.

Invariant reviews are predominant. Table IV shows that most merged reviews are invariant regarding the evaluated

TABLE IV
TYPE OF IMPACT OF MERGED CODE REVIEWS

Number of Merged Reviews Per Type of Impact				
Impact	**CG Density**	**FG Density**	**CG Diversity**	**FG Diversity**
Positive	1,879 (12%)	2,155 (15%)	101 (<1%)	11 (<1%)
Negative	4,876 (33%)	7,081 (47%)	137 (<1%)	49 (<1%)
Invariant	8,216 (55%)	5,735 (38%)	14,733 (99%)	14,911 (99%)

characteristics. The only case in which the proportion of invariant reviews is below 50% is for the density of fine-grained smells. In this case, most of the reviews (47%) had a negative effect. This result suggests that either (i) modern code review is not enough to avoid design degradation or (ii) code review is enough despite being predominantly invariant. In Section V-B, we explore these two possibilities in detail.

Low impact on the diversity of symptoms. The last two columns of Table IV reveal that most reviews do not impact on the diversity of both categories of symptoms. This happens because the diversity is only impacted when all occurrences of a smell type are removed from the changed code or when a new smell type is introduced. We noticed that the introduction of new types of smell usually occurs in the early stages of development since the codebase is still small and less complex. Complete removals of specific smell types usually occur when significant changes are made to the design structure of the system. An example of this occurred in the review number 11,099 [47] of the spymemcached system.

Impact of reviewing design related tasks. As explained in Section IV, we classified a sub-set of reviews into two groups: design-related and design-unrelated reviews. We used the Chi-Square test to compare the impact of both groups of reviews on the degradation characteristics. Table V shows the results for the density of coarse-grained and fine-grained smells. We will not discuss the results for diversity as they were not statistically significant. Nevertheless, all the results are available in our replication package [39].

TABLE V
CHI-SQUARE RESULTS FOR EVALUATING THE DEPENDENCY BETWEEN THE TYPE OF IMPACT AND THE RELATION WITH DESIGN

Impact	**CG Density**		**FG Density**	
	Design Related	**Design Unrelated**	**Design Related**	**Design Unrelated**
Positive	**190 (156.35) [7.24]**	**125 (158.65) [7.14]**	151 (131.53) [2.88]	114 (133.47) [2.84]
Negative	443 (477.98) [2.56]	520 (485.02) [2.52]	535 (566.33) [1.73]	606 (574.67) [1.71]
Invariant	250 (248.67) [0.01]	251 (252.33) [0.01]	197 (185.14) [0.76]	176 (187.86) [0.75]
Chi Square	X^2 = 19.4775, p-value = .000059		X^2 = 10.672, p-value = .004815	

The last line of Table V shows the Chi-Square factors (X^2) and the *p-values*. The other lines represent the impact type (positive, negative, and invariant) of classified reviews. The 2nd and 3rd columns show the distribution of reviews into the two compared groups regarding their impact on the density of coarse-grained smells, while the last two columns show the same information for the density of fine-grained smells. The numbers in parentheses represent the number of reviews that are statistically expected in each cell, given their classification regarding impact (lines) and design-relation (columns). Outside of the parentheses is the number of reviews that, in fact, were observed in each cell. Finally, in brackets is a value that represents how much each the observed number of reviews contributed to the composition of the Chi-Square factor. The higher the difference between expected and observed number of reviews in the cell, the higher will be the value.

Table V shows that the number of design-related reviews with a positive or invariant impact on the density of smells is higher than expected. Moreover, the number of design-related reviews with negative impact is also lower than expected. Conversely, there were more design-unrelated reviews with a negative impact than would be expected. This result is consistent and statistically significant for both coarse-grained and fine-grained smells. We interpret this result as evidence that design-related reviews tend to have a more positive and neutral impact than other types of review. This means that when there is an explicit intention to improve the design, the degradation can be reduced or at least remain invariant.

Impact of design discussions. To better characterize the reviews, we conducted another comparison. In this case, we compared reviews where there were explicit design discussions (With DD) with reviews without discussions related to design (Without DD). Once again, we applied the Chi-Square test to compare both groups regarding their impact on the density and diversity of symptoms. The results were not statistically significant for the diversity of CG smells and for both symptoms of FG smells. Table VI shows the result of this comparison for the density of coarse-grained smells.

TABLE VI
CHI-SQUARE RESULTS FOR EVALUATING THE DEPENDENCY BETWEEN THE TYPE OF IMPACT AND THE PRESENCE OF DESIGN DISCUSSIONS

Impact	**Coarse-grained Smells**	
	With DD	**Without DD**
Positive	63 (64.10) [0.02]	252 (250.90) [0.00]
Negative	**235 (195.96) [7.78]**	728 (767.04) [1.99]
Invariant	**64 (101.95) [14.12]**	437 (399.05) [3.61]
Chi-Square	X^2 = 27.5229, p-value <.001	

Our results reveal that design discussions tend to be associated with a negative impact on the density of coarse-grained smells. We hypothesize that this result occurred as structurally degraded code usually draws more attention from reviewers, often causing some type of design discussion. Nevertheless, as we observed in our manual analysis, such discussions may not contribute to the reduction of severe design problems.

> **Finding 1**: Reviews with explicit intents of design improvement tend to reduce or avoid design degradation. However, the sole presence of design discussions is not enough for avoiding design degradation.

B. Degradation Evolution along a Single Review

We address $\mathbf{RQ}_2$ by identifying four patterns of design degradation evolution along with a single review. The procedure that we followed to identify these patterns is defined in Section IV-B (Step 4). For each degradation characteristic (density and diversity), we classified reviews into four patterns: invariant, positive, negative, and mixed. Such patterns can be summarized as follows. **Invariant** is composed of reviews in which the characteristic remained the same across all the revisions submitted during the code review. **Positive** is the pattern for reviews in which the last revision reduces the degradation characteristic when compared to the previous ones. **Negative** groups reviews for which the last revision

presents an increase in the degradation characteristic when compared to the previous ones. Finally, **mixed** is composed of reviews with signs of reduction and an increase of the characteristic along the revisions.

Table VII shows the degradation evolution within reviews grouped by symptom category, i.e., *coarse-grained* and *fine-grained* smells, and by characteristic, i.e., *density* and *diversity*. We also group the reviews by type, i.e., *design-related* or *design-unrelated*, and different levels of design discussion. In this table, we show information about a subset composed only of reviews that improved (*Improvement* columns) or degraded (*Degradation* columns) the structural design, according to the degradation characteristics. For each case, we present the ratio of reviews classified into the four aforementioned patterns: invariant *(Inv)*, positive *(Pos)*, negative *(Neg)*, and mixed *(Mix)*.

Degradation characteristics and their variation across revisions. For design-related reviews that presented signs of reduction in degradation characteristics (3rd and 5th main columns of Table VII), we observed that density and diversity of coarse-grained smells remain invariant in 71% and 60% of the cases, respectively. A similar observation applies when we consider fine-grained smells. The density and diversity of such category of smells are invariant in 62% and 100% of the cases, respectively. On the other hand, for design-related reviews with signs of degradation (4th and 6th main columns of Table VII), the ratio of invariant reviews for coarse-grained smells is 54% for both density and diversity. For fine-grained smells, the values of density and diversity are invariant in 52% and 56% of the cases, respectively.

These observations indicate that 64% of design-related reviews, tend to remain invariant, i.e., preserve the same design impact throughout all revisions. Conversely, 52% of design-unrelated reviews tend to remain invariant. We also observed that, for reviews with signs of improvement, the degradation impact tends to change more when the reviewers provide design feedback. In such cases, for design-related reviews, the impact on the density and diversity of coarse-grained smells of the latest merged revision was different than the previous ones in 88% and 50% of the cases, respectively. For fine-grained smells, we observed that in 88% of the cases, the density changed in the latest merged revision. A similar pattern was observed in reviews that presented signs of degradation.

> **Finding 2**: For design-related and design-unrelated reviews, the degradation impact on the latest merged revision in comparison with all previous ones tends to remain invariant in 64% and 52% of the cases, respectively.

Signs of degradation reduction. Table VII presents other observations. For design-related reviews that reduce the density or diversity of coarse-grained smells (3rd and 5th main columns), we did not observe any positive evolution, i.e., changes that improve the structural quality. This happens even when the reviewers provide feedback on the design quality. On the other hand, for the density of fine-grained smells, we observed positive and negative evolution patterns in 4% and 1% of the cases, respectively. Such results are surprising since the ratio of reduction of 4% only for fine-grained smells was below expectations and different from studies that investigate the impact of refactoring, i.e., a technique that is commonly used during code review [11], [12], [48].

As expected, the ratio of design-related reviews with a negative evolution is higher in reviews that present signs of degradation (4th and 6th main columns of Table VII). In such cases, for the density of coarse-grained smells, we observed positive and negative evolution patterns in 1% and 2% of the cases, respectively. Conversely, we observed a negative evolution pattern in 8% of degradation reviews for diversity. Regarding the density of fine-grained smells, we observed positive and negative evolution patterns in 1% and 3% of the cases, respectively. Again, we did not observe any positive or negative evolution related to diversity.

We also observed that for design-related reviews with signs of either improvement or degradation, in which the design is discussed in both the description and comments, there were not successive decreases of the density of coarse-grained smells. Conversely, we observed an increase in the density of coarse-grained smells with a ratio of negative evolution is 14%. This is a surprising finding as we expected that design feedback during code review would lead to improvements, i.e., a reduction of the design degradation. On the other hand, considering the level of design discussion in reviews with fine-grained smells, we observed that when the reviewers provide design feedback during code review, the evolution patterns are predominantly positive (22%) and mixed (67%).

These results indicate that design discussions during code review may influence the review's impact on degradation characteristics. However, such impact tends to be positive only for the density of fine-grained smells, indicating that design discussions provided by developers during code review do not help the developers to decrease coarse-grained smells, i.e., such symptoms are often aggravated rather than minimized. In fact, fine-grained smells are simpler to remove and refactor as they represent smaller readability and understandability problems [10]. Conversely, coarse-grained smells are often hard to remove as they represent more severe problems, requiring more complex refactorings [12].

> **Finding 3**: Code reviews usually do not reduce coarse-grained smells, even when there is design feedback.

Degradation symptoms and their fluctuation during code review. Finally, we observed that the ratio of design-related reviews with a sign of improvement in density and diversity of coarse-grained smells are often classified as mixed evolution in a ratio of 27% and 40% of the cases, respectively. Regarding the impact on the density of fine-grained smells, the ratio of reviews with a mixed evolution is also the highest in 33% of the cases when compared to the reviews with positive and negative evolution. Surprisingly, this pattern of evolution holds even when the developer provides some feedback on the design quality. A similar observation applies when we consider the

TABLE VII
THE RATIO OF REVIEWS GROUPED BY TYPE, LEVEL OF DESIGN DISCUSSION, SYMPTOM CATEGORY, CHARACTERISTIC AND DEGRADATION EVOLUTION

Coarse-grained Smells

Type	Design discussion	Density								Diversity							
		Improvement				Degradation				Improvement				Degradation			
		Inv	Pos	Neg	Mix	Inv	Pos	Neg	Mix	Inv	Pos	Neg	Mix	Inv	Pos	Neg	Mix
Design-related	Never	77%	0%	1%	22%	60%	0%	1%	38%	50%	0%	0%	50%	50%	0%	0%	50%
	Description	77%	0%	0%	23%	65%	3%	0%	32%	100%	0%	0%	0%	100%	0%	0%	0%
	Comments	11%	0%	0%	89%	17%	0%	6%	77%	50%	0%	0%	50%	-	-	-	-
	Both	29%	0%	14%	57%	25%	3%	5%	68%	-	-	-	-	0%	0%	100%	0%
	All	71%	0%	1%	27%	54%	1%	2%	43%	60%	0%	0%	40%	54%	0%	8%	38%
Design-unrelated	Never	69%	4%	5%	21%	61%	1%	3%	35%	71%	0%	7%	21%	71%	0%	0%	29%
	Description	60%	40%	0%	0%	65%	0%	0%	35%	-	-	-	-	0%	0%	0%	100%
	Comments	0%	0%	0%	100%	20%	0%	0%	80%	-	-	-	-	100%	0%	0%	0%
	Both	33%	0%	0%	67%	17%	0%	0%	83%	-	-	-	-	-	-	-	-
	All	66%	6%	5%	24%	57%	1%	3%	40%	71%	0%	7%	21%	71%	0%	0%	29%

Fine-grained Smells

Type	Design discussion	Density								Diversity							
		Improvement				Degradation				Improvement				Degradation			
		Inv	Pos	Neg	Mix	Inv	Pos	Neg	Mix	Inv	Pos	Neg	Mix	Inv	Pos	Neg	Mix
Design-related	Never	73%	1%	1%	25%	59%	1%	2%	39%	100%	0%	0%	0%	67%	0%	0%	33%
	Description	59%	9%	0%	32%	56%	0%	4%	40%	-	-	-	-	100%	0%	0%	0%
	Comments	11%	22%	0%	67%	14%	0%	0%	86%	-	-	-	-	-	-	-	-
	Both	13%	0%	0%	88%	24%	3%	8%	66%	-	-	-	-	0%	0%	0%	100%
	All	62%	4%	1%	33%	52%	1%	3%	44%	100%	0%	0%	0%	56%	0%	0%	44%
Design-unrelated	Never	52%	2%	4%	41%	52%	2%	4%	41%	0%	0%	0%	100%	43%	0%	0%	57%
	Description	60%	40%	0%	0%	34%	3%	3%	59%	0%	0%	0%	100%	0%	0%	0%	100%
	Comments	38%	0%	0%	63%	14%	0%	0%	86%	-	-	-	-	0%	0%	0%	100%
	Both	100%	0%	0%	0%	13%	0%	0%	88%	-	-	-	-	-	-	-	-
	All	68%	1%	5%	26%	48%	2%	4%	46%	0%	0%	0%	100%	30%	0%	0%	70%

design-related reviews that presented signs of degradation, in which nearly 42% of reviews are classified as mixed.

In our manual analysis, we observed that mixed evolution usually occurs as a side effect of two factors: (i) deleting of duplicated or unnecessary code; and (ii) reorganizing the code to make it more reusable. For instance, consider the code review 9,015 from the linuxtools, which caused a significant improvement regarding coarse-grained smells. This review has a total of 11 revisions, in which fluctuation occurred after the following feedback provided by reviewer: *"This class and ProviderOptionsTab are almost identical except for a few small differences [...]. Would it be possible to define some abstract class and have these inherit override just what they need?"*. Such suggestion led to an increase in smells that affect abstraction and encapsulation issues.

These observations suggest that even if the developers identify and remove fine- and cross-grained smells, they still will not be able to see all the ramifications of the impact of their changes along revisions. However, at the end of the code review process, i.e., in the last merged revisions, the developers tend to preserve the positive impact on the system's internal structure. We conjecture that this happens because the existing modern code review tools still lack a mechanism to provide developers a just-in-time recommendation about the impact of their changes on software design degradation [49].

Finding 4: Nearly 34% of design-related reviews present a mixed evolution. This happens even in reviews that present signs of improvement and degradation. This result motivates the proposition of recommenders to better support developers, in the improvement of design quality and prevention of design problems during code review.

C. Code Review Practices and Design Degradation

We address $\mathbf{RQ}_3$ by assessing the influence of code review practices on software degradation. We have applied a *multiple logistic regression* to support this assessment (Step 6 of Section IV-B). Table VIII summarizes the main results. Each row represents the results for each project, separated by symptom category (coarse-grained (CG) and fine-grained (FG) smells) and degradation characteristic (density and diversity). The "all" row represents the results for the data of all projects combined. The gray cells represent the metrics that presented statistical significance relation in a specific combination of symptoms and characteristics. Moreover, we used the $\uparrow$ symbol to indicate a degradation risk-increasing effect, and the $\downarrow$ symbol to indicate a degradation risk-decreasing effect.

The data of three projects were omitted from the Table VIII, but can be fully seen on our replication package [39]. We removed these data because only a few metrics were statistically significant, but they are considered on the "all" row. Additionally, the control variables (Section III) were omitted, but when applied on the model, only the `DiffComplexity` and `DiffSize` variables were statistically significant across projects. To understand if the control variables were collinear with the code review metrics, we executed the model only with the control variables, and the results were similar. This implies that our code review metrics were capable of working as a standalone model to verify the risk-increase or risk-decrease effect on degradation in each system. Next, considering only statistically significant cases, we discuss the code review practices by risk-increasing and risk-decreasing effects as follows.

Risk-increasing effect on software degradation. We observed that the metrics `DL`, `PRWF`, `NNAV`, `PRD`, `RL`, `TRW`, `ARR`, and `RD` presented a risk-increase tendency. By analyzing each metric individually, we observed that four metrics sustained a risk-increasing tendency across projects: Discussion Length (`DL`), Proportion of Revisions without Feedback (`PRWF`), Proportion of Review Disagreement (`PRD`), and Review Length (`RL`). Such behavior was expected for the `PRWF` and `PRD` metrics, since they confirm the rationale presented in Table III. Conversely, the results for the `DL`, `RL` and `TRW` metrics are

TABLE VIII
CODE REVIEW ACTIVITY METRICS OF REVIEWS WITH POSITIVE AND NEGATIVE IMPACT, GROUPED BY SYSTEM, SYMPTOM, CHARACTERISTIC AND DIMENSION. THE ↑ SYMBOL TO INDICATE A RISK-INCREASING EFFECT, AND THE ↓ SYMBOL TO INDICATE A RISK-DECREASING EFFECT

System	Symptom	Characteristic	Review Intensity			Review Participation			Review Time			
			DL	PRWF	CDCR	NA	NNAV	PRD	RL	TRW	ARR	RD
jgit	CG	Density	1.023	1.129	1.713	1.008	0.772 ↓	1.143	1.191	0.856	1.127	1.04
		Diversity	1.046	1.134	1.705	1.007	0.768 ↓	1.132	1.181	0.865	1.118	1.034
	FG	Density	1.334	1.04	1.472	1.014	0.777 ↓	0.93	1.028	1.279 ↑	0.897	1.035
		Diversity	1.334	1.04	1.472	1.014	0.777 ↓	0.93	1.028	1.279 ↑	0.897	1.035
egit	CG	Density	1.497 ↑	0.941	0.342 ↓	0.97	1.116	1.01	1.306 ↑	0.701 ↓	1.094	1.018
		Diversity	1.321	0.994	0.4 ↓	0.961	1.107	1.075	1.279 ↑	0.587 ↓	1.294 ↑	1.009
	FG	Density	1.416 ↑	0.976	0.743	0.956	0.76 ↓	1.012	1.322 ↑	1.069	0.903	1.169 ↑
		Diversity	1.354 ↑	0.974	0.701	0.961	0.767 ↓	1.025	1.366 ↑	1.069	0.903	1.153 ↑
linuxtools	CG	Density	1.112	1.569 ↑		0.864	1.166	1.257 ↑	0.912	1.249	0.713 ↓	1.118
		Diversity	1.116	1.582 ↑		0.854	1.151	1.268 ↑	0.898	1.216	0.732 ↓	1.14
	FG	Density	0.873	1.533 ↑		0.842 ↓	1.231 ↑	1.394 ↑	0.94	1.021	0.826	1.11
		Diversity	0.873	1.533 ↑		0.842 ↓	1.231 ↑	1.394 ↑	0.94	1.021	0.826	1.11
platform.ui	CG	Density	1.073	1	0.723 ↓	0.971	1.175	0.932	1.032	0.827	1.268	0.872 ↓
		Diversity	1.071	1	0.717 ↓	0.946	1.176	0.938	1.031	0.82	1.275 ↑	0.877 ↓
	FG	Density	1.051	0.998	0.737 ↓	0.984	1.053	0.979	1.265 ↑	0.899	1.272 ↑	0.909
		Diversity	1.051	0.998	0.737 ↓	0.984	1.053	0.979	1.265 ↑	0.899	1.272 ↑	0.909
All	CG	Density	1.115 ↑	1.145		0.96 ↓	1.16	1.005	1.045 ↑	0.887	1.076 ↑	0.917
		Diversity	1.106 ↑	1.162		0.948 ↓	1.158	1.016	1.037 ↑	0.845 ↓	1.12 ↑	0.913
	FG	Density	1.155 ↑	1.092		0.962	1.043	1.025 ↑	1.079 ↑	1.101	0.957	0.954
		Diversity	1.144 ↑	1.091		0.963	1.045	1.028 ↑	1.084 ↑	1.102	0.955	0.953

unexpected, since they differ from that were reported by related studies [7], [44], [45], we will discuss latter in this section. By considering the degradation symptoms, all metrics presented themselves as good predictors for all symptoms and characteristics, except for a few metrics: the Number of Non-Authors Votes (NNAV) metric, that on the project linuxtools was only related to fine-grained smells, as a risk-increasing factor; and Average Review Rate (ARR) that was related to coarse-grained smells on 66% of the cases as a risk-increasing factor. Moreover, the Proportion of Revisions without Feedback (PRWF) metric stood out as the metrics with the highest risk-increasing effect, followed by the Discussion Length (DL) metric, reaching values of 1.58, and 1.49, respectively.

Long discussions are not reflected as concerns about structural degradation. By considering the intensity dimension metrics (Table VIII), we observed that two of the three metrics presented a degradation risk-increase tendency in at least one project. The metrics Discussion Length (DL) and Proportion of Revisions without Feedback (PRWF) preserved this behavior in most of the significant cases. This observation indicates that developers tend to introduce more instances and more types of degradation symptoms in reviews that either has longer discussions or do not have any discussions started by human participants. As illustrated in our motivating example (Section III), and also confirmed in our manual analysis, long discussions do not necessarily indicate that developers and reviewers are concerned about the structural quality of the code. Reviewers can often be concerned with functional aspects of the system, paying less attention to possible signs of degradation. Thus, when there are extensive discussions about specific features, the code tends to undergo further modifications that increase design degradation.

Finding 5: Reviews for which the practice of long discussions was applied are often associated with a higher risk of software degradation.

A high proportion of review disagreement leads to a degradation risk-increasing effect. By considering the metrics of the participation dimension, only the Proportion of Review Disagreement (PRD) metric presented a risk-increase tendency across all projects. Nevertheless, the Number of Non-Author Voters (NNAV) metric raised some concerning results on the linuxtools system. As illustrated in our motivating example (Section III), this result indicates that reviews with a high rate of acceptance discrepancy tend to introduce more instances and more types of degradation symptoms.

Finding 6: Reviews following the practice of participation with a higher rate of review disagreement lead to a higher risk of software degradation.

Lack of reviewers' attention and code review speed increase the risk of degradation. Table VIII also shows that the longer (RL) and faster (ARR) review takes to be finished, the higher the risk of degradation. At first, this result seems to be counter-intuitive. However, in our manual analysis, we observed that certain reviews often take a longer time due to the lack of attention of the reviewers during the code review. This observation suggests that reviewers will be able to identify more design problems that are overlooked during the code review, whether they perform a careful code inspection with an appropriate code review rate. By considering our motivating example again, we can observe that this review takes a long time from the first patch submission to the reviewer's acceptance for integration (Aug 16, 2015 to Oct 9, 2015). Moreover, we observed a lack of attention from the reviewer of revision 7 to revision 10. For instance, the code author has addressed the reviewer's lack of attention with the following comment: *"[...] do you have time for a review?"*. After that comment, the code author only got a response from the reviewer after 16 days. Within this time, the code author continued to modify the source code without feedback from the reviewers.

Finding 7: Reviews tend to be longer due to the lack of attention from the reviewer during the code review process, and this increases the risk of software degradation.

Risk-decreasing effect on software degradation. Table VIII shows that the Churn During Code Review (CDCR),

Number of Authors (NA), Number of Non-Authors Votes (NNAV), Time Review Window (TRW), Average Review Rate (ARR), and Response Delay (RD) metrics present a risk-decreasing tendency. Moreover, the metrics NA, and NNAV also presented consistent results, as the NNAV showing a risk-decreasing likelihood for most target systems, except for the linuxtools system. Moreover, this metric was risk-decreasing in 75% of the statistically significant cases. Thus, it can be considered as a reliable predictor of reduction in structural degradation. CDCR showed good results on egit and platform.ui projects, performing very good results on egit (0.34). Looking over the dimensions, there is evidence that the CDCR metric is a reliable predictor of risk-decreasing for the Intensity Dimension, the NNAV represents the Participation Dimension, on Time Dimension no metric really outcome as a reliable predictor. We can see a minor difference between symptoms; the risk-decreasing appeared more (60%) on the coarse-grained symptom.

Active engagements of multiple reviewers decrease the risk of degradation. By considering the participation dimension, the Number of Authors (NA), and the Number of Non-Authors Votes (NNAV) metrics showed to be reliable predictors of risk-decreasing. However, only the NA preserved this behavior across projects. Thus, the higher the number of authors, the higher will be the likelihood of degradation risk-decreasing. This observation is aligned with previous studies [5], [8], [35], [36], by suggesting the greater the number of authors revising the proposed changes during reviews, the more design issues could be identified and removed, especially coarse-grained smells, whose identification and removal often required a better understanding of the codebase.

Finding 8: Reviews with active engagements of multiples reviewers tend to present a degradation risk-decreasing effect, especially for coarse-grained smells.

VI. Threats to Validity

Construct and Internal Validity. Aspects such as the precision and recall of degradation symptoms may have influenced the results of this study. We tried to mitigate this threat by selecting a detection tool that has been successfully used in recent studies involving design degradation [21], [38], [50].

We have selected a set of 12 code review activity metrics that helped us measure different dimensions of code review practices, i.e., intensity, participation, and time. The rationales for using metrics are supported by previous studies (e.g., [7], [36]). We wrote scripts to automate the design degradation evolution pattern computation and code review metrics. These scripts were validated by two of the paper authors. Regarding the code review activity metrics, we measured some metrics based on heuristics. For instance, we have assumed that the review length is the time that elapses between the time a patch has been uploaded and when it has been approved for integration. Thus, although there is a limitation of measuring the code review practices, we rely on state-of-the-art practices based on heuristics to recover this kind of information.

Conclusion and External Validity. We carefully performed our descriptive and statistical analysis. About the descriptive analysis, four paper authors contributed to the analysis of code review impact on design degradation. For the statistical analysis, we rely on the *Multiple Logistic Regression*, as previously stated in Section V-C, we reduced the heavy skew of our metrics applying a $\log_2$ transformation on the right-skewed predictors and a x^3 transformation on the left-skewed. Moreover, we normalized the continuous predictors in the model to provide normality, and, to ensure the statistical significance, we employed the customary *p-value* of 0.05 for each predictor in the regression model. Furthermore, in our statistical model, we controlled some factors that may affect our outcomes via product and process metrics.

Regarding the qualitative analysis of design-related reviews, we employed a two-phase manual classification procedure. In the first, all reviews were classified by two authors. In the second phase, for all reviews in disagreement, both authors discussed to reach a unified classification. Finally, the analysis of code review impact on design degradation is based on two degradation characteristics – density and diversity of symptoms. One might expect different results using other characteristics. We relied on density and diversity because they are widely-adopted for design degradation analysis and have been evaluated in previous studies [18], [21], [51].

VII. Conclusion and Future Work

In this work, we analyzed the impact of modern code review on evolution of design degradation, by mining code review data from two large open source communities. Our findings pointed out that design discussions may not be enough for avoiding design degradation. Conversely, reviews with explicit intent to improve the design tend to have a positive or invariant impact on design degradation. We also observed that, during the revisions of each review, there is often a wide fluctuation of design degradation. Such fluctuations often result in the amplification of design degradation, even in design-related reviews. Finally, we observed that certain code review practices might increase the risk of design degradation, including long discussions, and a high rate of reviewers' disagreement.

As future works, we aim to (i) evaluate the effect of code reviews on other types of degradation symptoms, and different characteristics of design degradation; (ii) expand the code review metrics and dimensions to understand their role on software degradation, and (iii) investigate mechanisms to better support developers, in the continuous improvement of design quality during code review.

Acknowledgment

This work was partially funded by CNPq (grants 434969/2018-4, 312149/2016-6, 140919/2017-1, 141285/2019-2, 131020/2019-6), CAPES/Procad (grant 175956), CAPES/Proex, and FAPERJ (200773/2019, 010002285/2019).

REFERENCES

[1] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *35th ICSE*, 2013, pp. 712–721.

[2] P. C. Rigby and C. Bird, "Convergent contemporary software peer review practices," in *9th FSE*, 2013, pp. 202–212.

[3] P. C. Rigby, "Open source peer review–lessons and recommendations for closed source," *IEEE Software*, 2012.

[4] F. E. Zanaty, T. Hirao, S. McIntosh, A. Ihara, and K. Matsumoto, "An empirical study of design discussions in code review," in *12th ESEM*. ACM, 2018, p. 11.

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

[6] O. Kononenko, O. Baysal, and M. W. Godfrey, "Code review quality: how developers see it," in *38th ICSE*. IEEE, 2016, pp. 1028–1038.

[7] 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 *22nd SANER*. IEEE, 2015, pp. 171–180.

[8] L. Pascarella, D. Spadini, F. Palomba, and A. Bacchelli, "On the effect of code review on code smells," in *27th SANER*, 2020.

[9] M. Beller, A. Bacchelli, A. Zaidman, and E. Juergens, "Modern code reviews in open-source projects: Which problems do they fix?" in *11th MSR*, 2014, pp. 202–211.

[10] M. V. Mäntylä and C. Lassenius, "What types of defects are really discovered in code reviews?" *IEEE Trans. Softw. Eng. (TSE)*, vol. 35, no. 3, pp. 430–448, 2008.

[11] M. Paixao, J. Krinke, D. Han, C. Ragkhitwetsagul, and M. Harman, "The impact of code review on architectural changes," *IEEE Trans. Softw. Eng. (TSE)*, pp. 1–19, 2019.

[12] M. Paixão, A. Uchôa, A. C. Bibiano, D. Oliveira, A. Garcia, J. Krinke, and E. Arvonio, "Behind the intents: An in-depth empirical study on software refactoring in modern code review," in *17th MSR*, 2020.

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

[14] T. Besker, A. Martini, and J. Bosch, "Time to pay up: Technical debt from a software quality perspective." in *CIbSE*, 2017, pp. 235–248.

[15] R. C. Martin, *Agile software development: principles, patterns, and practices*. Prentice Hall, 2002.

[16] J. Brunet, G. C. Murphy, R. Terra, J. Figueiredo, and D. Serey, "Do developers discuss design?" in *11th MSR*. ACM, 2014, pp. 340–343.

[17] W. Oizumi, A. Garcia, L. Sousa, B. Cafeo, and Y. Zhao, "Code anomalies flock together: Exploring code anomaly agglomerations for locating design problems," in *38th ICSE*, 2016.

[18] L. Sousa, A. Oliveira, W. Oizumi, S. Barbosa, A. Garcia, J. Lee, M. Kalinowski, R. de Mello, B. Fonseca, R. Oliveira *et al.*, "Identifying design problems in the source code: A grounded theory," in *40th ICSE*, 2018, pp. 921–931.

[19] A. Yamashita, M. Zanoni, F. A. Fontana, and B. Walter, "Inter-smell relations in industrial and open source systems: A replication and comparative analysis," in *31st ICSME*. IEEE, 2015, pp. 121–130.

[20] A. Eposhi, W. Oizumi, A. Garcia, L. Sousa, R. Oliveira, and A. Oliveira, "Removal of design problems through refactorings: are we looking at the right symptoms?" in *27th ICPC*. IEEE Press, 2019, pp. 148–153.

[21] W. Oizumi, L. Sousa, A. Oliveira, L. Carvalho, A. Garcia, T. Colanzi, and R. Oliveira, "On the density and diversity of degradation symptoms in refactored classes: A multi-case study," in *30th ISSRE*, 2019.

[22] G. A. Oliva, I. Steinmacher, I. Wiese, and M. A. Gerosa, "What can commit metadata tell us about design degradation?" in *13th IWPSE*, 2013, pp. 18–27.

[23] D. L. Parnas, "Software aging," in *16th ICSE*. IEEE, 1994, pp. 279–287.

[24] J. Martins, C. Bezerra, A. Uchôa, and A. Garcia, "Are code smell co-occurrences harmful to internal quality attributes? a mixed-method study," in *34th SBES*, 2020, pp. 1 – 10.

[25] L. Sousa, R. Oliveira, A. Garcia, J. Lee, T. Conte, W. Oizumi, R. de Mello, A. Lopes, N. Valentim, E. Oliveira *et al.*, "How do software developers identify design problems? a qualitative analysis," in *31st SBES*, 2017, pp. 54–63.

[26] I. Ahmed, U. A. Mannan, R. Gopinath, and C. Jensen, "An empirical study of design degradation: How software projects get worse over time," in *10th ESEM*. IEEE, 2015, pp. 1–10.

[27] U. A. Mannan, I. Ahmed, R. A. M. Almurshed, D. Dig, and C. Jensen, "Understanding code smells in android applications," in *3rd MOBILESoft*. IEEE, 2016, pp. 225–236.

[28] T. Sharma and D. Spinellis, "A survey on software smells," *J. Syst. Softw. (JSS)*, vol. 138, pp. 158–173, 2018.

[29] R. C. Martin and M. Martin, *Agile Principles, Patterns, and Practices in C# (Robert C. Martin)*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2006.

[30] D. Feitelson, E. Frachtenberg, and K. Beck, "Development and deployment at facebook," *IEEE Internet Comput.*, vol. 17, no. 4, pp. 8–17, 2013.

[31] T. Hirao, A. Ihara, Y. Ueda, P. Phannachitta, and K.-i. Matsumoto, "The impact of a low level of agreement among reviewers in a code review process," in *12th OSS*, 2016, pp. 97–110.

[32] A. Trifu and R. Marinescu, "Diagnosing design problems in object oriented systems," in *12th WCRE*, 2005, pp. 10–pp.

[33] C. Barbosa, A. Uchôa, F. Falcao, D. Coutinho, H. Brito, G. Amaral, A. Garcia, B. Fonseca, M. Ribeiro, V. Soares, and L. Sousa, "Revealing the social aspects of design decay: A retrospective study of pull requests," in *34th SBES*, 2020, pp. 1 – 10.

[34] Eclipse, https://git.eclipse.org/r/\#/c/53827/, 2020, accessed in: July 2019.

[35] 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 *11th MSR*, 2014, pp. 192–201.

[36] 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 *12th MSR*. IEEE Press, 2015, pp. 168–179.

[37] M. Paixao, J. Krinke, D. Han, and M. Harman, "Crop: Linking code reviews to source code changes," in *15th MSR*, 2018, pp. 46–49.

[38] T. Sharma, P. Mishra, and R. Tiwari, "Designite: a software design quality assessment tool," in *1st BRIDGE*. ACM, 2016, pp. 1–4.

[39] A. Uchôa, C. Barbosa, W. Oizumi, P. Blenílio, R. Lima, A. Garcia, and C. Bezerra. (2020) Replication package for the paper: "how does modern code review impact software design degradation? an in-depth empirical study". Accessed: 2020-08-18. [Online]. Available: https://zenodo.org/record/3989787

[40] 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," *Emp. Softw. Eng. (ESE)*, vol. 23, no. 3, pp. 1188–1221, 2018.

[41] 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," *Emp. Softw. Eng. (ESE)*, vol. 17, no. 3, pp. 243–275, 2012.

[42] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, "Predicting fault incidence using software change history," *IEEE Trans. Softw. Eng. (TSE)*, vol. 26, no. 7, pp. 653–661, 2000.

[43] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu, "Don't touch my code! examining the effects of ownership on software quality," in *19th FSE*, 2011, pp. 4–14.

[44] J. Tsay, L. Dabbish, and J. Herbsleb, "Influence of social and technical factors for evaluating contribution in github," in *36th ICSE*. ACM, 2014, pp. 356–366.

[45] A. Porter, H. Siy, A. Mockus, and L. Votta, "Understanding the sources of variation in software inspections," *ACM Trans. Softw. Eng. Methodol. (TOSEM)*, vol. 7, no. 1, pp. 41–79, 1998.

[46] C. F. Dormann, J. Elith, S. Bacher, C. Buchmann, G. Carl, G. Carré, J. R. G. Marquéz, B. Gruber, B. Lafourcade, P. J. Leitão *et al.*, "Collinearity: a review of methods to deal with it and a simulation study evaluating their performance," *Ecography*, vol. 36, no. 1, pp. 27–46, 2013.

[47] Couchbase, http://review.couchbase.org/\#/c/11099/, 2020, accessed in: July 2019.

[48] D. Cedrim, A. Garcia, M. Mongiovi, R. Gheyi, L. Sousa, R. de Mello, B. Fonseca, M. Ribeiro, and A. Chávez, "Understanding the impact of refactoring on smells: A longitudinal study of 23 software projects," in *11th FSE*, 2017, pp. 465–475.

[49] E. Fernandes, A. Uchôa, A. C. Bibiano, and A. Garcia, "On the alternatives for composing batch refactoring," in *3rd IWoR*, 2019, pp. 9–12.

[50] M. Alenezi and M. Zarour, "An empirical study of bad smells during software evolution using designite tool," *i-Manager's Journal on Software Engineering*, vol. 12, no. 4, p. 12, 2018.

[51] A. Oliveira, L. Sousa, W. Oizumi, and A. Garcia, “On the prioritization of design-relevant smelly elements: A mixed-method, multi-project study,” in *13th SBCARS*, 2019, pp. 83–92.