Four Eyes Are Better Than Two: On the Impact of Code Reviews on Software Quality

Gabriele Bavota*, Barbara Russo*
*Faculty of Computer Science, Free University of Bozen-Bolzano, Bolzano, Italy
gabriele.bavota,barbara.russo@unibz.it

***Abstract*—Code review is advocated as one of the best practices to improve software quality and reduce the likelihood of introducing defects during code change activities. Recent research has shown how code components having a high review coverage (*i.e.,* a high proportion of reviewed changes) tend to be less involved in post-release fixing activities. Yet the relationship between code review and bug introduction or the overall software quality is still largely unexplored.**

This paper presents an empirical, exploratory study on three large open source systems that aims at investigating the influence of code review on (i) the chances of inducing bug fixes and (ii) the quality of the committed code components, as assessed by code coupling, complexity, and readability.

Findings show that unreviewed commits (*i.e.,* commits that did not undergo a review process) have over two times more chances of introducing bugs than reviewed commits (*i.e.,* commits that underwent a review process). In addition, code committed after review has a substantially higher readability with respect to unreviewed code.

***Index Terms*—Code Review, Mining Software Repositories, Empirical Studies**

I. Introduction

Peer Code Review (or simply Code Review) is the process of analysing code written by a teammate to judge whether it is of sufficient quality to be integrated into the main code trunk. With respect to a traditional inspection process formalised by Fagan [14], code review is more informal, tool-based, and used regularly in practice [2]. Previous works have investigated the effects of code review both in open source [28], [29], [5], [6], [23] and in industrial [2], [19] systems. Most of the studies have been performed by mining modern code review repositories (*e.g.,* Gerrit[1], Microsoft's CodeFlow [2], Google's Mondrian[2], Facebook's Phabricator[3]) that allow for automated analysis of code reviews by providing on-line structured information on review comments and participants, and references to commits and their patches.

Recent research has mainly mined such repositories to describe the review process and the communication among reviewers with qualitative and quantitative analysis [10], [9], [25]. In 2014, McIntosh *et al.* [23] have pioneered a study on the effect of code reviews on quality by mining such repositories. The authors have shown that in open source systems the percentage of reviewed changes a code component underwent correlates inversely to its chance of being involved in post-release fixes. The authors have also illustrated that low participation increments the occurrence of such fixes. At the same time, Beller *et al.* [6] have characterised the type of defects fixed in modern code review repositories. All these studies reflected on the code review process and its outcome.

As a new contribution to the field, we examine software quality with and without code review by means of using on-line history repositories. Specifically, we investigate to what extent code review impacts on the chances of inducing bug fixes during commit activities and the internal quality of the committed code components, as assessed by code complexity [22], coupling [12], and readability [11]. To perform our study, we mined the change history of three software systems, `Android frameworks base APIs`, `LibreOffice`, and `Scilab` that use the Gerrit code review repository.

We extracted all commit data of the three systems starting from the day in which they adopted Gerrit as code review platform and collected a total of 87,197 commits. Then, we retrieved all code reviews from the Gerrit repository. By linking commits and reviews, we classified commits as *reviewed* (*i.e.,* commits that underwent a review process before code is committed) or *unreviewed*. We further used the SZZ algorithm [32], [20] to determine commits that induce bug fixes. By definition, these commits contain the last change to a piece of code that has been later changed in fixing a bug. We then compared the proportions of commits that induced a bug and the variation of internal code quality in reviewed and unreviewed commits. Finally, we analysed the participation degree in terms of number of reviewers and comments [23] in all reviewed commits to verify its influence on the chances of inducing bugs. In other words, we study whether lax reviews have higher chances to induce future bug fixes.

Our **findings** show that:

- Unreviewed commits have over two times higher chance of inducing bug fixes with respect to reviewed commits;
- Code committed after review has a significantly higher readability with respect to the code committed without review;
- On two of the three object systems, the lower the number of reviewers involved in a code review, the higher the chance of inducing bug fixes.

The **contribution** of this paper is twofold:

- *An empirical study performed on three large open source systems that aims at investigating the relationship between code review practices and code quality by means of on-line repositories*. To the best of our knowledge the present study is the first that (i) investigates the

[1]https://code.google.com/p/gerrit/
[2]http://tinyurl.com/5v6wyl
[3]http://phabricator.org

Accepted for publication by IEEE. © 2015 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

relation between code review, reviewers participation and the induction of bug fixes, (ii) classifies reviewed and unreviewed commits and compares the chances of inducing bug fixes in the two categories, and (iii) compares code complexity, coupling and readability in committed code of reviewed and unreviewed commits.
- *A comprehensive replication package* [4], including the *R* scripts and working data sets used to run the statistical tests and produce the results reported in this paper.

Structure of the paper. Section II defines our empirical study and the research questions, and provides details about the data extraction process and analysis method. Section III reports the results of the study, answering our research questions, while Section IV discusses the threats that could affect the validity of the results achieved. Section VI concludes the paper and outlines directions for future work, after a discussion of the related literature (Section V).

II. Study Design

The *goal* of the study is to compare reviewed and unreviewed commits and their committed code *in terms* of the chances of inducing bug fixes and the quality of the committed code respectively. The *quality focus* is on bug introduction and code internal quality, which could be influenced by code review activities. The *perspective* is of researchers and practitioners interested in the effects of code review on code quality.

A. Context and Research Questions

The *context* of the study consists of the change and review history of three open source projects, namely `Android frameworks base APIs`, `LibreOffice`, and `Scilab`. `Android frameworks base` is a subsystem of the Android APIs and collects the classes and services that can be used by all devices hosting an Android operating system. `LibreOffice` is an office suite while `Scilab` is a software for numerical computation providing a computing environment for scientific applications. Table I reports the characteristics of the analysed systems: programming language, observation time period, size range (KLOC), number of bugs (#issues), and number of commits.

We choose the object systems according to the following **selection criteria**:

Criterion 1 - *On-line traceability of the reviewed activity with bugs, commits, and patches*. We selected projects that use an on-line versioning system, an issue tracker, and a Gerrit code review repository.

Criterion 2 - *Representativeness of the sample classes*. We selected projects that have a sufficient number of both reviewed and unreviewed commits to perform statistically significant comparison. Each of the selected projects has at least 200 reviewed and unreviewed commits[4] (see Table I).

[4]Note that for the Android APIs we only consider the reviews related to the investigated subsystem (*i.e.,* Android frameworks base).

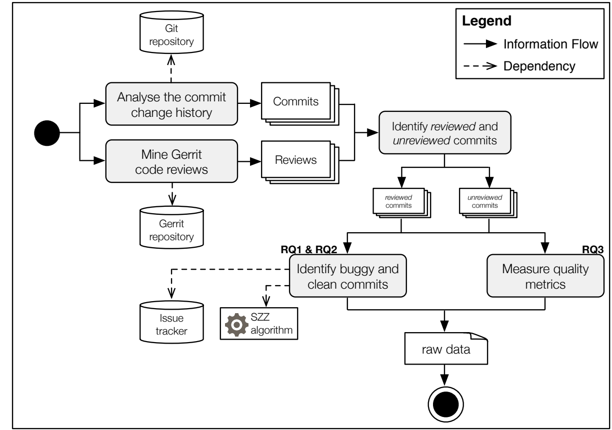

Fig. 1. Data extraction process.

In the context of the study, we formulated the following **research questions**:

RQ_1 - *Does code review affect the chances of inducing bugs during commit activities?* This research question investigates if reviewed commits are less/more prone to induce bug fixes than unreviewed commits. Indeed, one could hypothesise that code written by a developer D and then reviewed by some other developers before being committed is less likely to induce bug-fixes than code written and committed by D without any further check by other developers. The null hypothesis being tested is as following:

$H_{0,1}$ - *There is no difference in the chance of inducing fixes between reviewed and unreviewed commits.*

RQ_2 - *Does the code review participation degree affect the chances of inducing bugs during commit activities?* This research question aims at verifying if there is a relationship between the degree of participation in a code review (*e.g.,* the number of involved reviewers) and the chances of inducing a bug-fix. The null hypothesis being tested is as following:

$H_{0,2}$ - *There is no difference in participation degree between reviewed commits that induce and not induce bug fixes.*

RQ_3 - *Does code review affect the quality of the code components committed by developers?* This research question investigates whether the code components committed in reviewed commits exhibit a higher/lower quality than those committed in unreviewed commits. In particular, we measure three attributes of quality: complexity, coupling, and readability. For any of these quality attributes, we test the following null hypothesis:

$H_{0,3}$ - *There is no difference between the quality attribute of code components in reviewed and unreviewed commits.*

B. Data Extraction Process

Fig. 1 depicts the main steps behind the data extraction process that we followed. The top left steps describe how we collected commits and link them to reviews by mining change

TABLE I
CHARACTERISTICS OF SYSTEMS UNDER ANALYSIS

Project	Link	Language	Period	KLOC	#Issues	#Commits	
						Reviewed	Unreviewed
Android frameworks base APIs	http://tinyurl.com/lwoypdk	Java	Jan 2010-Dec 2014	534-1,043	9,311	1,593	31,761
LibreOffice	http://www.libreoffice.org/	C++	Mar 2011-Oct 2014	64-6,812	3,104	5,337	45,763
Scilab	http://www.scilab.org	Java/C++	Jan 2010-Sep 2014	679-3,172	1,421	2,499	243
Overall	-	-	-	-	**13,836**	**9,429**	**77,677**

logs of the Git version control system and reviews in the Gerrit repository, respectively.

We first mine the change log of code checked-in by developers in Git. To reduce potential noise in our analysis, we excluded (i) the first commit that imports entire initial version of the system into the repository and (ii) commits involving more than 30% of the files in the repository that are likely due to re-organization of files in folders or license statement changes. In addition, we only analyse the change history of the object systems starting from the first day in which they started using Gerrit (*i.e.,* the day in which the first review was created in Gerrit). This is needed since, once we collected the commits, we classified them into *reviewed* and *unreviewed* based on the information stored in the Gerrit repositories.

Firstly, we downloaded from each Gerrit instance of a project the set of reviews related to *merged* changes (*i.e.,* changes that have been reviewed and then merged in the versioning system). Gerrit is a web based code review system for projects using the Git version control system. The typical reviewing process in Gerrit is performed as follows:

1) *Authors* of a code change upload the corresponding patch (*i.e.,* a portion of code aimed at updating a software system) in Gerrit, asking for review.
2) A set of *reviewers* are assigned to a code change with the responsibility of verifying the correctness and soundness of the patches. The selection of such reviewers can be manual (*e.g.,* directly performed by the change's author) or automatic (*i.e.,* Gerrit can automatically allocate reviewers to patches on the basis of their experience). A set of *verifiers* can also be assigned to the patch. While the role of reviewers is to look at the code, to ensure it meets the project guidelines, intent *etc.*, verifiers generally just check that the code actually compiles and unit tests pass. For this reason, the verification activity is often automated by sanity bots.
3) The involved reviewers assign a score ranging between -2 (block the change) and 2 (allow the change) to the patch, while the verifiers just assign a fail (-1) or pass (+1) score to the patch. Based on the project policy, changes that receive positive scores (*e.g.,* the patch has been verified by the bot, and has received at least one +2 and no -2 from reviewers) are merged in the Git repository, and the status of the change becomes *merged*. Instead, in case of negative reviews, the patch might be updated and re-submitted for a new review round or abandoned.

For each review, we store: (i) the author of the change, (ii) the list of reviewers[5], (iii) the comments present in the discussion[6], and (iv) its *Change-Id*. The latter is an alpha-numeric hash code uniquely identifying each review. When reviewed changes are merged (committed) into the versioning control system, the Change-Id is automatically reported in the commit message for traceability purposes. Through the Change-Id, we link reviews to commits and label a commit as *reviewed* if it is linked to a review having at least one reviewer, excluding the author of the change if present among the reviewers. Otherwise the commit is classified as *unreviewed*.

The right-bottom part of Fig. 1 illustrates the mining process of change logs to identify commits inducing bug-fixes. Firstly, we identified bug fixing commits by mining regular expressions containing issue IDs in the change log of the versioning system, *e.g.,"fixed issue #ID"* or *"issue ID"*. Secondly, for each issue ID related to a commit, we downloaded the corresponding issue reports from their issue tracking system and extracted the following information from them (i) *product name*; (ii) *issue's type*, *i.e.,* whether an issue is a bug, enhancement request, *etc.*; (iii) *issue's status*, *i.e.,* whether an issue was closed or not; (iv) *issue's resolution*, *i.e.,* whether an issue was resolved by fixing it, or whether it was a duplicate bug report, or a "works for me" case; (v) *issue's opening date*; (vi) *issue's closing date*, if available.

Then, we checked each issue's report to be correctly downloaded (*e.g.,* the issue's ID identified from the versioning system commit note could be a false positive). After that, we used the issue type field to classify the issue and distinguish bug fixes from other issue types (*e.g.,* enhancements). Finally, we only considered bugs having *Closed* status and *Fixed* resolution. In this way, we restricted our attention to (i) issues that were related to bugs, and (ii) issues that were neither duplicate reports nor false alarms.

To answer RQ_1 and RQ_2, we identify commits that were likely to induce fixes by means of the SZZ algorithm [32], [20], last step of the data collection process for these research questions (Fig. 1). The algorithm relies on the annotation/blame feature of versioning systems. In essence, given a bug-fix BF_k, identified by the bug ID, k, the approach works as follow:

1) For each file f_i, $i = 1 \dots m_k$ involved in BF_k (m_k is the number of files changed in BF_k), and fixed in its revision *rel-fix*$_{i,k}$, we extract the file revision just *before* the bug fixing (*rel-fix*$_{i,k} - 1$).
2) starting from the revision *rel-fix*$_{i,k} - 1$, for each source line in f_i changed to fix the bug k the *blame* feature of Git is used to identify the file revision where the last change

[5]Note that reviewers, on the opposite of verifiers, are not automatic bots.

[6]We removed comments left by automatic bots exploited in Gerrit as verifiers.

to that line occurred. In doing that, blank lines and lines that only contain comments are identified using an island grammar parser [24]. This produces, for each file f_i, a set of $n_{i,k}$ fix-inducing revisions $\textit{rel-bug}_{i,j,k}$, $j = 1 \ldots n_{i,k}$. Thus, more than one commit can be indicated by the SZZ algorithm as responsible for the inducing of a fix.

To collect data for RQ_3, we developed a tool that measures complexity, coupling, and readability in the code involved in each commit we have previously identified (see right-bottom steps in Fig. 1). Worth noticing here that our analysis has been performed at "file level" for sake of simplicity. Thus, we measure code complexity of a file as the sum of the McCabe's cyclomatic complexity [22] of the methods it contains. Coupling for a file is measured as the the sum of the Coupling Between Object [12] of the classes it contains. Note that both the values of complexity and coupling are non-negative and unbounded. Finally, we measured the readability of a file by exploiting the metric proposed by Buse and Weimer [11]. This metric combines a set of low-level code features (*e.g.,* identifiers length, number of loops, *etc.*) and has been shown to be 80% effective in predicting developers' readability judgments. We used the authors' implementation of such a metric[7]. Given a code file, the readability metric takes values between 0 (lowest readability) and 1 (maximum readability).

From a commit C, our tool extracts its id, the date in which it has been performed, and the list of files added, deleted, and modified in such a commit; then it computes the complexity, coupling, and readability of the source code files that have been *modified* in C before and after the changes applied in C; and (ii) the complexity, coupling, and readability of the source code files *added* in C. We compute the complexity of the code involved in a commit C as:

$$comp(C) = \frac{(comp_C(M) - comp_{C_{-1}}(M)) + comp_C(A)}{|M| + |A|}$$

where $comp_C(M) - comp_{C_{-1}}(M)$ is the complexity diff of the files modified in C, $comp_C(A)$ is the complexity of the files added in C, and $|M|$ and $|A|$ are the number of files modified and added in C, respectively. We do not consider files deleted in C since their complexity cannot be charged in any way (positive or negative) on commit C. The complexity of the files deleted in C is charged to the previous commits that either modified/added any of these files. We further discuss such a choice in Section IV. The computation of coupling(C) and readability(C) is specular to the complexity(C) one. Note that complexity, coupling, and readability for a commit C can assume positive as well as negative values. For instance, a commit C not adding any new file but reducing the complexity of the files it modifies will have a negative value of complexity(C).

In total, our tool mined 87,197 commits, providing for each of them the three quality measures described above. This process took four weeks of computation on a workstation having a quad-core 3.2Ghz CPU and 8Gb of RAM.

[7] Available at http://tinyurl.com/kzw43n6

C. Analysis

This subsection describes the analyses and statistical procedures that we used to answer the three research questions formulated in Section II-A.

To address $\mathbf{RQ}_1$, we compute the contingency matrix defined by:

- $\mathbf{NB}_{NR}$, the number of unreviewed commits that did not induce a bug-fix;
- $\mathbf{B}_{NR}$, the number of unreviewed commits that did induce a bug-fix;
- $\mathbf{NB}_{R}$, the number of reviewed commits that did not induce a bug-fix;
- $\mathbf{B}_{R}$, the number of reviewed commits that did induce a bug-fix.

Then, we use the Fisher's exact test [30] to test whether the proportions B_{NR}/NB_{NR} and B_R/NB_R significantly differ. Correspondingly, we use the Odds Ratio (OR) [30] of the two proportions as effect size measure. An OR of 1 indicates that the condition or event under study (*i.e.,* the chances of inducing a bug) is equally likely in both groups. An OR greater than 1 indicates that the condition or event is more likely in the first group (*i.e.,* the *unreviewed commits* in our case). Vice versa, an OR lower than 1 indicates that the condition or event is more likely in the second group (*i.e.,* the *reviewed commits*).

In the context of $\mathbf{RQ}_2$, we further investigate the participation in code review and its effects on the chance of inducing a bug fix during commit activities, similarly to what has been done by McIntosh *et al.* [23]. Participation is measured by using as proxies (i) the number of involved reviewers and (ii) the number of comments left in the review discussion. One could expect that *reviewed commits* for which the linked review has a high participation (*i.e.,* several reviewers involved and/or several comments left in the discussion) are less prone to induce bugs. To verify such a hypothesis, we present a descriptive statistics reporting:

1) The distribution of the number reviewers in reviewed commits that induced a bug (*buggy reviewed commits*) and in reviewed commits that did not induce a bug (*clean reviewed commits*).
2) The distribution of the number of comments in buggy and clean reviewed commits.

To compare the two distributions of buggy and clean reviewed commits, we exploit the Mann-Whitney test [13]. This latter is used to analyse statistical significance of the differences between the number of reviewers and the number of comments in buggy and clean reviewed commits. The results are intended as statistically significant at $\alpha = 0.05$. We also estimated the magnitude of the measured differences by using the Cliff's Delta (or d), a non-parametric effect size measure [15] for ordinal data. We followed well-established guidelines to interpret the effect size values: negligible for $|d| < 0.10$, small for $0.10 \leq |d| < 0.33$, medium for $0.33 \leq |d| < 0.474$, and large for $|d| \geq 0.474$ [15].

Concerning $\mathbf{RQ}_3$, we report and compare the descriptive statistics of the distribution (box plots) of complexity, cou-

TABLE II

NUMBER OF COMMITS REVIEWED (OR NOT) AND INDUCING (OR NOT) A BUG, AND RESULTS OF THE FISHER'S EXACT TEST.

System	NB_{NR}	B_{NR}	NB_R	B_R	p-value	OR
Android APIs	23,439	8,322	1,521	81	**<0.001**	**3.98**
LibreOffice	43,216	2,547	5,190	147	**<0.001**	**2.08**
Scilab	119	15	2,370	129	**0.001**	**2.20**

TABLE III

SIZE COMPARISON, IN TERMS OF IMPACTED LOC, BETWEEN *reviewed* AND *unreviewed* COMMITS.

System	Reviewed commits			Unreviewed commits			p-value	d
	Mean	Median	St. Dev.	Mean	Median	St. Dev.		
Android APIs	2,619	1,155	3,658	3,586	1,731	4,972	**<0.01**	**0.16 (Small)**
LibreOffice	3,841	2,075	5,461	3,175	1,571	5,078	**<0.01**	**-0.11 (Small)**
Scilab	1,524	699	2,198	976	427	1,454	**<0.01**	**-0.18 (Small)**

pling, and readability for reviewed and unreviewed commits. Also in this case, the Mann-Whitney test and the Cliff's Delta are adopted to statistically support our findings.

III. RESULTS DISCUSSION

This section discusses the results achieved in our study according to the three research questions.

A. *RQ_1: Does code review affect the chances of inducing bugs during commit activities?*

Table II reports the number of commits in each of the categories (NB_{NR}, NB_R, B_{NR}, and B_R) and the results of the Fisher's exact test and ORs for each project.

In all three systems, the proportion of bug inducing commits is significantly different for commits involved or not in code review activities. *Unreviewed commits* have always a higher chance of inducing a bug-fix with respect to *reviewed commits*. In particular, this chance is from 2.08 (LIBREOFFICE) up to 3.98 (ANDROID APIS) times higher—see Table II. Indeed, on the ANDROID APIS 5% of *reviewed* commits induces a bug, against 17% of *unreviewed* commits, on LIBREOFFICE 3% of *reviewed* commits induces a bug against 6% of *unreviewed* commits, and on SCILAB the comparison is 5% against 13%.

This result is likely due to the fact that some bugs are caught during the reviewing process, thus reducing the chance of inducing a future fix. However, we also have to take into account the possible role played by the "size" confounding factor. Indeed, as highlighted in previous work [20], [34], larger commits (*i.e.,* commits impacting a larger set of code components) have a higher chance of being classified by the SZZ algorithm as bug-inducing commits. Thus, it is possible that the achieved results are simply due to the fact that *unreviewed commits* are larger than *reviewed commits*, thus having a higher chance of inducing bugs. Table III reports descriptive statistics of the size for *reviewed* and *unreviewed* commits measured as the number of impacted lines of code as assessed by the *Unix diff*. Table III also reports the results of the Mann-Whitney test (p-value) and Cliff's Delta (d) computed and interpreted by adopting the procedure described in Section II-C. The first thing that leaps to the eyes is that, while there is always statistically significant difference in the size of *reviewed* and *unreviewed* commits, the magnitude of such differences is always small. Also, on two out of the three systems (*i.e.,*LIBREOFFICE and SCILAB), *reviewed* commits are larger than *unreviewed* commits, thus strengthen the likely role played by the code review process in avoiding bug introduction. On the opposite, on the ANDROID APIS project *unreviewed* commits are in general larger than *reviewed* commits, highlighting a possible role played by the size factor on the results achieved for this system. Nevertheless, the consistent results reported in Table II for all three systems (*i.e., reviewed* commits are less likely to induce bugs than *unreviewed* commits), make us confident about the role played by the code review process in reducing the chances of inducing bugs. Indeed, also on systems where *reviewed* commits are larger than *unreviewed* commits, we still observe their lower chance of inducing bugs.

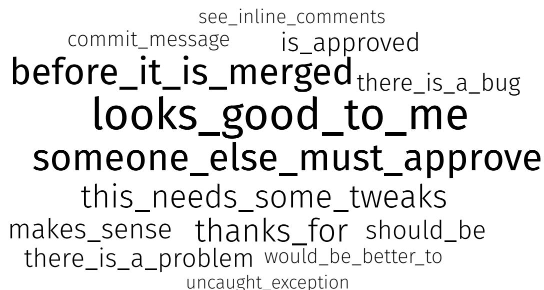

Fig. 2. Word cloud of the 15 most common n-grams in reviewer's comments.

To have a more qualitative confirmation of this finding, we performed a coarse grained automatic analysis of comments left by reviewers in the *reviewed* changes. The goal of this analysis is just to get an idea of the most frequent topics tackled in the reviewers' comments. To this aim, we extracted from comments the n-grams composing them, considering $n \in [2..4]$. Fig. 2 reports the 15 most common n-grams we found. Note that if x-gram was included in an y-gram with $x < y$ (*e.g.,* the two-gram "is merged" is included in the three-gram "it is merged"), we sum the x-gram frequency to the y-gram ones. Also, we discarded all comments automatically generated by Gerrit.

As we can notice there are two main "categories" among the most frequent n-grams. The first concerns n-grams related with comments indicating the approval of the reviewed patch; part of this category are *looks good to me*, *thanks for*, *someone else must approve*[8], *makes sense*, *is approved*. The second category relates instead to problems found by the reviewers in the patch, *e.g., this needs some tweaks*, *before it is merged*, *there is a problem*, *there is a bug*. This second category shows as often reviewers ask for changes to the author of the change, highlighting possible issues in the code under review. In particular, n-grams like "*there is a problem*" and "*there is a bug*" seem to support our quantitative findings, indicating situations in which potential bugs have been fixed before being committed (merged) in the code repository.

Summary for RQ_1. The achieved results allow us to reject the null hypothesis $H_{0,1}$ and state that unreviewed commits

[8]The complete sentence generally is: "looks good to me but someone else must approve".

TABLE IV
Reviewed commits: IMPACT OF CODE REVIEW PARTICIPATION ON THE CHANCES OF INDUCING A BUG.

System	Clean reviewed commits					Buggy reviewed commits					Statistical test	
Number of reviewers												
	mean	median	min	max	st. dev.	mean	median	min	max	st. dev.	p-value	d
Android APIs	4.15	4.00	1.00	15.00	2.19	4.28	4.00	1.00	12.00	2.39	0.54	0.09 (Negligible)
LibreOffice	1.28	1.00	1.00	6.00	0.60	1.08	1.00	1.00	3.00	0.31	**<0.001**	**0.13 (Small)**
Scilab	1.22	1.00	1.00	5.00	0.50	1.06	1.00	1.00	4.00	0.33	**<0.001**	**0.14 (Small)**
Number of comments												
	mean	median	min	max	st. dev.	mean	median	min	max	st. dev.	p-value	d
Android APIs	7.56	6.00	2.00	113.00	7.10	7.49	6.00	2.00	20.00	4.46	0.83	-0.06 (Negligible)
LibreOffice	4.48	3.00	1.00	165.00	5.32	4.43	3.00	1.00	92.00	3.68	0.47	0.01 (Negligible)
Scilab	5.49	4.00	1.00	95.00	5.08	4.21	4.00	2.00	11.00	1.57	0.05	0.09 (Negligible)

have a much higher chance (over two times) of inducing bugs with respect to reviewed commits.

B. *RQ_2: Does the code review participation degree affect the chances of inducing bugs during commit activities?*

Table IV reports the descriptive statistics for number of reviewers (top part of Table IV) and number of comments (bottom part of Table IV) for both clean and buggy reviewed commits, the p-value of the Mann-Whitney test, and of the Cliff's d effect size when comparing clean and buggy reviewed commits in terms of number of reviewers and number of comments. The Mann-Whitney test indicates that the number of reviewers plays a role in the chances of inducing a bug for two of the systems. The Cliff's d is positive, but small in both projects. On the one side, this implies that the number of reviewers for clean reviewed commits tends to be slightly higher than for buggy reviewed commits. On the other side, given its narrow range (see values for mean, median, and standard deviation), this also indicates that the difference "clean vs. buggy commits" in number of reviewers is limited (one-two people). No significant difference can be observed instead in the number of reviewers for the ANDROID APIS.

The Mann-Whitney test does not report significant differences when comparing the number of comments in *reviewed* and *unreviewed* commits across the three projects (see Table IV). Thus, we are not able to support or disprove the results on participation in McIntosh *et al.* [23] given that we cannot tell whether comments have any role in inducing fixes.

Summary for RQ_2. In the case of number of reviewers, we can reject the null hypothesis $H_{0,2}$ for two of the three projects and state that a lower number of reviewers is correlated to a higher chance of inducing new bug fixes.

C. *RQ_3: Does code review affect the quality of the code components committed by developers?*

Figures 3, 4, and 5 depict the distribution of complexity, coupling, and readability, respectively, between reviewed and unreviewed commits on the three object systems[9].

In all three systems, the median of code complexity is lower in *reviewed commits*, Fig. 3. Also, a substantial part of the distribution assumes negative values for *reviewed commits*, indicating a drop in complexity of the code committed after

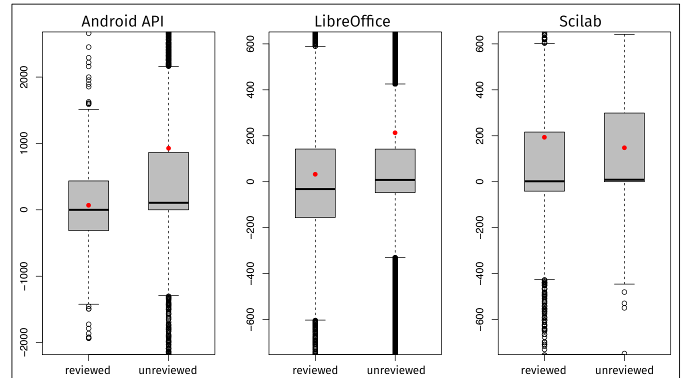

Fig. 3. Code complexity. The red spot is the mean.

a review. The Mann-Whitney test (p-value) confirms the difference between the distributions of complexity in reviewed and unreviewed commits on two out of the three systems (see Table V). The positive effect size (Cliff's d) indicates that this difference is due to higher complexity of code committed without a review. This result is further confirmed in many of the review comments we manually inspected. For instance, in `LibreOffice`, in the review to change 5327[10], one of the reviewers commented:

> "*This needs some tweaks before it is merged.
> Some inline comments, check the rest of the change if there are more similar instances where there is unnecessary complex dance between OUStringBuffers and OUStrings.*"

Code review does not seem to have any effect on coupling, Fig. 4. The box plots illustrate that there is higher coupling in unreviewed commits on average, but this difference is not so strong over the whole distribution. Namely, the Mann-Whitney test is statistically significant for LIBREOFFICE and the ANDROID APIS with positive effect size, but negligible for the former, and small for the latter.

While we were not able to state a priori any specific reason for this result, the manual inspection of commits and committed code suggested that reviewers are mainly focused on low level changes that often pertain few lines of code or a single method. Conversely, coupling changes act on relations between classes and, as such, require a larger view of the system. This observation reminds what Bacchelli and Bird re-

[9] Y-axis in Figures 3 and 4 has been delimited for sake of legibility.

[10] https://gerrit.libreoffice.org/#/c/5327/

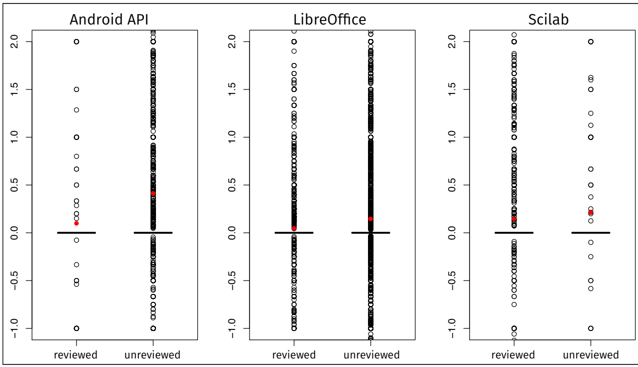

Fig. 4. Code coupling. The red spot is the mean.

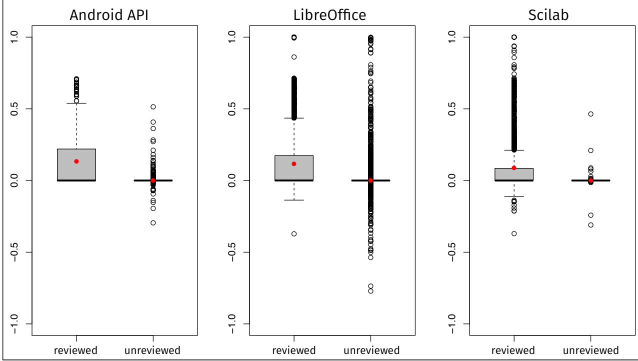

Fig. 5. Code readability. The red spot is the mean.

ported about Microsoft's developers complaining that reviews often focus on minor logic errors rather than discussing deeper design issues [2].

Readability of the committed code is definitely better after review. The distribution of readability is skewed toward positive values for reviewed commits, Fig. 5. The result of the Mann-Whitney test is significant and the Choen's d is negative (*i.e.,* higher readability in *reviewed commits*) and medium/large in all projects—see Table V. This result is likely due to two different and complementary factors:

1) *Review process*. We can reasonably think that improvements suggested by the reviewers during the review process help in achieving a higher level of code readability. Also in this case we manually analysed some of the code reviews object of our study, founding dozens of examples showing comments left by the reviewers aimed at improving code readability. For instance, in a `LibreOffice` review[11], one of the reviewers commented: "*Patch is awesome thanks so much for this! [. . .] I'll push a cleanup patch [. . .]. In general, I think we can have [...suggested changes to the code...] to make it simpler and more readable*".
2) *Developer's pride*. Developers knowing that their code will be subject to a review process performed by their fellow colleagues, are likely to pay more attention in writing clean and readable source code.

Summary for RQ$_3$. We reject the null hypothesis $H_{0,3}$ for the internal quality attributes complexity (on two systems)

[11] https://gerrit.libreoffice.org/#/c/10349/

TABLE V
CODE INTERNAL QUALITY ATTRIBUTES IN *reviewed* AND *unreviewed* COMMITS: MANN-WHITNEY TEST (p-VALUE) AND CLIFF'S DELTA (d).

complexity		
System	p-value	d
Android APIs	**<0.001**	**0.21 (Small)**
LibreOffice	**<0.001**	**0.17 (Small)**
Scilab	0.055	0.07 (Negligible)

coupling		
System	p-value	d
Android APIs	**<0.001**	**0.12 (Small)**
LibreOffice	**<0.001**	**0.03 (Negligible)**
Scilab	0.221	0.03 (Negligible)

readability		
System	p-value	d
Android APIs	**<0.001**	**-0.47 (Large)**
LibreOffice	**<0.001**	**-0.50 (Large)**
Scilab	**<0.001**	**-0.37 (Medium)**

and readability. On the other side, we cannot reject the null hypothesis $H_{0,3}$ on coupling.

IV. THREATS TO VALIDITY

This section describes the threats that can affect the validity our study.

Threats to *construct validity* concern the relation between the theory and the observation, and in this work are mainly due to the measurements we performed. This is the most important kind of threat for our study, and is related to:

- *Misclassification of reviewed and unreviewed commits*. We used the reviews' *Change-Id* to link commits and reviews. The *Change-Id* is automatically reported in the commit message making us confident in the correct identification of such links. However, we cannot exclude that some commits have been reviewed outside the Gerrit system (*e.g.,* in mailing lists), leading us to misclassify some commits. At least, we only analysed the change history of the object system in the time period in which they adopted Gerrit as code review platform. Thus, all reviewed commits *should* be identifiable via the Gerrit repository.
- *Missing or wrong links between bug tracking systems and versioning systems [7]*. Although not much can be done for missing links, we verified that links between commit notes and issues are correct.
- *Imprecision in issue classification made by issue-tracking systems [1]*. While we cannot exclude misclassification of issues (*e.g.,* an enhancement classified as a bug), at least the three systems we consider use an explicit classification of bugs on issue tracking systems, distinguishing them from other issues.
- *Approximations due to identifying bug-inducing changes using the SZZ algorithm [20]*. At least, we used heuristics to limit the number of false positives, for example excluding blank and comment lines from the set of bug-inducing changes. Also, we analysed the size of *reviewed* and *unreviewed* commits to verify the role played by the size confounding factor in the identification of bug-inducing commits [34].

- *Imprecision due to tangled code changes [17]*. We cannot exclude that some *reviewed* commits grouped together tangled code changes, of which just a subset has been object of review process.
- *Decision of not considering deleted files in the computation of complexity, coupling, and readability*. For the reasons explained in Section II-B, we did not consider files deleted in each commit when measuring its values for complexity, coupling, and readability. While this design choice might have had an impact on the achieved results, the percentage of commits deleting at least one file in the three object systems is very low: 2.9% (ANDROID APIS), 2.2% (LIBREOFFICE), and 5.8% (SCILAB). Thus, we do not expect this choice to have played a major role in our main findings.

Threats to *internal validity* concern external factors we did not consider that could affect the variables and the relations being investigated. Such threats can intrude the relation between code review and bug induction or quality. To reinforce the internal validity and interpret better the statistical results, when possible, we integrated the quantitative analysis with a qualitative one, showing examples we found by manually inspecting the Gerrit repositories.

Threats to *conclusion validity* concern the relation between the treatment and the outcome. Although this is mainly an observational study, wherever possible we used an appropriate support of statistical procedures, integrated with effect size measures that, besides the significance of the differences found, highlight the magnitude of such differences.

Threats to *external validity* concern the generalisation of results. We involved in our study three large open source systems. Again we used significance tests and effect size to ensure statistical generalisation and measure the magnitude of our observations. Yet, other systems should be analysed to support our conclusions. This is especially needed in the case in which the statistic output is not significant as in code review participation and code coupling.

V. RELATED WORK

Our study is mainly related to studies (i) dealing with code reviews, and (ii) mining software change history to identify factors that can play a role in the bug introduction/induction.

A. Code Reviews

Weißgerber *et al.* [33] mine mailing lists looking for messages containing patches. They analyse the time required by a patch to get accepted, also studying the impact of the patch size on the likelihood of being accepted. Their findings show that smaller patches have a higher chance to be accepted.

Rigby *et al.* [28] qualitatively examine the review practices adopted in the Apache projects. Their study showed that Apache reviews are performed continuously on small and independent pieces of code by a small group of people. Also, the authors highlighted several benefits provided by the review process, such as the education of new developers and the discussions aimed at fixing defects [28]. Our work studies the code review process from a more quantitative point of view, trying to actually measure the benefits it provides.

Kemerer and Paulk [19] show that design and code review reduces the amount of defects in student projects. With the available data they were also able to study the impact of review rate on the inspection performance. They found high review rates (*i.e.,* a high number of reviewed LOC/hour) to be associated with a decrease in inspections effectiveness. Differently from [19], our study targets three large open source systems, and quantitatively analyses the impact of code review on different aspects of software quality (*i.e.,* defect-proneness and internal code quality).

Rigby and Storey [29] analyse how the different stakeholders involved in the review process interact in open source systems. Two of the several findings they report are that (i) developers tend to postpone reviews, thus leading patches that fail to not generate interest among core developers until they become interesting, and (ii) core developers tend to avoid reviews that can lead to long, unproductive discussions [29].

Bacchelli and Bird [2] present a study performed in Microsoft and analysing the motivations, challenges, and outcomes of tool-based code reviews. Their findings show that while finding defects remain the main motivation for modern code review, this latter brings several other benefits, like knowledge transfer and creation of alternative solutions to problems. Similar findings have been found in the study by Beller *et al.* [6], that however focus on open source systems. Both these studies are qualitative in nature, and with different goals with respect to our work.

Baysal *et al.* [5] investigate the impact of non-technical factors (*e.g.,* reviewer load and activity, patch writer experience, *etc.*) on code review response time and outcome. The study, performed on the `WebKit` system, highlights as the analysed factors can strongly impact the code review outcomes.

Bosu [10] characterise the vulnerable code changes identified during peer code review. He found that less experienced authors have up to 24 times higher chance of committing vulnerable changes. Bosu and Carver [9] study the impact of developer reputation on code review outcomes in open source projects. Their findings highlight as changes implemented by core developers as compared to those implemented by peripheral developers are (i) reviewed faster and (ii) more likely to be accepted in the project codebase.

Morales *et al.* [25] show the positive effect of code review on software quality, and in particular of the likelihood of introducing anti-patterns (*i.e.,* reviewed code components are less likely to be affected by anti-patterns).

The recent study by McIntosh *et al.* [23] is the closest to our work. The authors mine the code review history of three systems (*i.e.,*`Qt`, `VTK`, and `ITK`) to analyse the relationship between code review coverage and participation and post-release defects. Their results show that the higher the percentage of reviewed changes a code component underwent, the lower is its chance of being involved in post-release defects fixes. This result is inline with what we observed in our RQ_1.

They also found that developer participation in code review[12] is associated with the incidence of post-release defects. Our RQ_2's results only partially confirm this finding, since we found only the number of reviewers involved to have an impact on the chance of inducing a bug. Note that our work has several differences from [23]:

1) *The level of granularity.* While in [23] the impact of code review on code quality is measured by looking at the relationship between post-release defects in code components and their percentage of reviewed changes, we directly compare the chance of inducing a bug in reviewed and unreviewed commits, by using the SZZ algorithm to identify fix-inducing commits.
2) *The target systems.* It is not by chance that there is no overlap between the object systems exploited in our paper and those used in [23]. Indeed, in [23] the authors focused their attention on "systems where a large number of the integrated patches have been reviewed". On our side, we needed systems with a good number of both reviewed and unreviewed commits.
3) *The additional quality aspects investigated in our work.* Our analysis of the impact of code review on the internal quality of the committed code is a premiere.

B. Investigating Factors Impacting the Likelihood of Inducing Fixes

Hassan [16] shows as the entropy of the code change process is a good proxy for identifying defect-prone code files. Eyolfson *et al.* [18] analyse the impact on the commits' bugginess of three characteristics pertinent to commits: (i) time of the day, (ii) day of the week, and (iii) developer's experience. Their results show that: (i) late-night commits are buggier than others, (ii) no day is buggier than another, and (iii) more experienced developers introduce bugs less frequently. Their second finding is in contrast with what observed by Sliwerski *et al.* [31], that identified Friday as the day when developers are likely to introduce bugs.

Rahman and Devanbu [27] investigate the impact of the ownership and the developer experience on the likelihood of introducing bugs. The authors mined software repositories looking for pieces of code modified to fix a bug; this code was tagged as *implicated code*. The analysis of this implicated code highlights that: (i) implicated code has higher ownership levels than non-implicated code, (ii) implicated code owner has lower contribution at a file level, and (iii) the experience of an author has no clear association with implicated code.

Bird *et al.* [8] mine change history of `Windows Vista` and `Windows 7` to verify the existence of a relationship between code ownership and software quality. They found that high levels of ownership are associated with fewer bugs. Kim *et al.* [20] use a machine learning-based classifier to determine if a performed change is more likely to be a fix-inducing change or a clean one.

[12]As in our paper, developer participation has been measured in [23] by using as proxies the number of reviewers involved and the number of comments they left.

Bavota *et al.* [3] analyse the chances of refactoring operations of inducing bug fixes. They found that while the percentage of faults likely induced by refactorings is relatively low (*i.e.,* 15%), there are some specific kinds of refactorings that are very likely to induce bug fixes, such as Pull Up Method and Extract Subclass, where the percentage of fixes likely induced by such refactorings is around 40%.

Posnett *et al.* [26] analyze the impact of developer's focus on bug introduction by analogising the developer-artifact contribution network to a predator-prey food web. Their results show that project leaders and top committers tend to be less focused on specific aspects of the system and showed that developers having specific focus introduce fewer bugs.

Our work enriches the current literature about factors impacting the likelihood of inducing bugs by investigating the impact of the code review process on the chance of inducing a bug fix.

VI. Conclusion and Future Work

In this paper, we reported an empirical analysis conducted on three open source systems and aimed at investigating the impact of code review practices on (i) the chances of inducing a bug fix during commit activities, and (ii) the quality of the committed code components as assessed by coupling, complexity, and readability metrics.

We analysed the change and review history of three open source systems, by linking the commits to the corresponding reviews extracted from Gerrit. Commits linked to a review were classified as *reviewed commits*, while those not linked to any review were labeled as *unreviewed commits*. Then, we used the SZZ algorithm [20] to determine whether each of the considered commits induced bug fixes. Also, we computed code metrics assessing the quality of the components committed in each commit.

Our results highlight the benefits of the code review process, and in particular that:

1) *unreviewed* commits have a much higher chance (over two times) of inducing bug fixes with respect to *reviewed* commits. The most reasonable explanation for this result, also partially confirmed by our qualitative analysis, is that bugs are caught during the review process thus reducing the chances of inducing future fixes.
2) code committed through reviewed commits have a higher readability with respect to the code committed in unreviewed commits. Also in this case we performed a qualitative analysis to at least in part confirm our quantitative findings.

Our future work agenda includes (i) replicate our study on other systems, in order to corroborate or contradict our findings; (ii) enlarge the set of quality metrics considered, including additional metrics from the CK suite [12] and semantic metrics [21]; and (iii) investigate the impact of code review on abandoned commits to verify which commit's characteristics pushes the reviewers to recommend an abandonment.

REFERENCES

[1] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y.-G. Guéhéneuc, "Is it a bug or an enhancement?: a text-based approach to classify change requests," in *Proceedings of the 2008 conference of the Centre for Advanced Studies on Collaborative Research, October 27-30, 2008, Richmond Hill, Ontario, Canada*. IBM, 2008, p. 23.

[2] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13, 2013, pp. 712–721.

[3] G. Bavota, B. De Carluccio, A. De Lucia, M. Di Penta, R. Oliveto, and O. Strollo, "When does a refactoring induce bugs? an empirical study," in *Source Code Analysis and Manipulation (SCAM), 2012 IEEE 12th International Working Conference on*, 2012, pp. 104–113.

[4] G. Bavota and B. Russo, "Four eyes are better than two: on the impact of code reviews on software quality – replication package," 2015. [Online]. Available: http://www.inf.unibz.it/~gbavota/reports/code-review/

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

[6] 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*, ser. MSR 2014. New York, NY, USA: ACM, 2014, pp. 202–211.

[7] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. T. Devanbu, "Fair and balanced?: bias in bug-fix datasets," in *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2009, Amsterdam, The Netherlands, August 24-28, 2009*. ACM, 2009, pp. 121–130.

[8] 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 *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. ACM, 2011, pp. 4–14.

[9] A. Bosu and J. C. Carver, "Characteristics of the vulnerable code changes identified through peer code review," in *36th International Conference on Software Engineering, ICSE '14, Companion Proceedings, Hyderabad, India, May 31 - June 07, 2014*, 2014, pp. 736–738.

[10] ——, "Impact of developer reputation on code review outcomes in OSS projects: an empirical investigation," in *2014 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM '14, Torino, Italy, September 18-19, 2014*, 2014, p. 33.

[11] R. P. Buse and W. R. Weimer, "Learning a metric for code readability," *IEEE Transactions on Software Engineering*, vol. 36, no. 4, pp. 546–558, 2010.

[12] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering (TSE)*, vol. 20, no. 6, pp. 476–493, June 1994.

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

[14] M. E. Fagan, "Design and code inspections to reduce errors in program development," *IBM Syst. J.*, vol. 38, no. 2-3, pp. 258–287, Jun. 1999. [Online]. Available: http://dx.doi.org/10.1147/sj.382.0258

[15] R. J. Grissom and J. J. Kim, *Effect sizes for research: A broad practical approach*, 2nd ed. Lawrence Earlbaum Associates, 2005.

[16] A. E. Hassan, "Predicting faults using the complexity of code changes," in *ICSE*. Vancouver, Canada: IEEE Press, 2009, pp. 78–88.

[17] K. Herzig and A. Zeller, "The impact of tangled code changes," in *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, San Francisco, CA, USA, May 18-19, 2013*, 2013, pp. 121–130.

[18] L. T. Jon Eyolfso and P. Lam, "Do time of day and developer experience affect commit bugginess?" in *Proceedings of the 8th Working Conference on Mining Software Repositories*, ser. MSR '11, 2011, pp. 153–162.

[19] C. Kemerer and M. Paulk, "The impact of design and code reviews on software quality: An empirical study based on psp data," *Software Engineering, IEEE Transactions on*, vol. 35, no. 4, pp. 534–550, 2009.

[20] S. Kim, E. J. W. Jr., and Y. Zhang, "Classifying software changes: Clean or buggy?" *IEEE Transactions on Software Engineering*, vol. 34, no. 2, pp. 181–196, 2008.

[21] A. Marcus, D. Poshyvanyk, and R. Ferenc, "Using the conceptual cohesion of classes for fault prediction in object-oriented systems," *IEEE Transactions on Software Engineering*, vol. 34, no. 2, pp. 287–300, 2008.

[22] T. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, 1976.

[23] 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*, ser. MSR 2014, 2014, pp. 192–201.

[24] L. Moonen, "Generating robust parsers using island grammars," in *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on*, 2001, pp. 13–22.

[25] 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 *Proc. of the 22nd Int'l Conf. on Software Analysis, Evolution, and Reengineering (SANER)*, 2015, pp. 171–180.

[26] D. Posnett, R. D'Souza, P. Devanbu, and V. Filkov, "Dual ecological measures of focus in software development," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. IEEE Press, 2013, pp. 452–461.

[27] F. Rahman and P. Devanbu, "Ownership, experience and defects: a fine-grained study of authorship," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11, 2011, pp. 491–500.

[28] P. C. Rigby, D. M. German, and M.-A. Storey, "Open source software peer review practices: A case study of the apache server," in *Proceedings of the 30th International Conference on Software Engineering*, ser. ICSE '08, 2008, pp. 541–550.

[29] P. C. Rigby and M.-A. Storey, "Understanding broadcast based peer review on open source software projects," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11, 2011, pp. 541–550.

[30] D. J. Sheskin, *Handbook of Parametric and Nonparametric Statistical Procedures*, 4th ed. Chapman & Hall/CRC, 2007.

[31] J. Sliwerski, T. Zimmermann, and A. Zeller, "Don't program on fridays! how to locate fix-inducing changes," in *Proceedings of the 7th Workshop Software Reengineering*, May 2005.

[32] ——, "When do changes induce fixes?" in *Proceedings of the 2005 International Workshop on Mining Software Repositories, MSR 2005*. ACM, 2005.

[33] P. Weißgerber, D. Neu, and S. Diehl, "Small patches get in!" in *Proceedings of the 2008 International Working Conference on Mining Software Repositories*, ser. MSR '08, 2008, pp. 67–76.

[34] C. Williams and J. Spacco, "Szz revisited: Verifying when changes induce fixes," in *Proceedings of the 2008 Workshop on Defects in Large Software Systems*, ser. DEFECTS '08, 2008, pp. 32–36.