Investigating Code Review Quality: Do People and Participation Matter?

Oleksii Kononenko*, Olga Baysal†, Latifa Guerrouj‡, Yaxin Cao†, and Michael W. Godfrey*
*David R. Cheriton School of Computer Science, University of Waterloo, Canada
Email: okononen@uwaterloo.ca, migod@uwaterloo.ca
† Department of Computer Science and Operations Research, Université de Montréal, Canada
Email: yaxin.cao@umontreal.ca, olga.baysal@umontreal.ca
‡ Department of Computer Science and Software Engineering, Concordia University, Canada
Email: latifa.guerrouj@polymtl.ca

***Abstract*—Code review is an essential element of any mature software development project; it aims at evaluating code contributions submitted by developers. In principle, code review should improve the quality of code changes (patches) before they are committed to the project's master repository. In practice, bugs are sometimes unwittingly introduced during this process.**

In this paper, we report on an empirical study investigating code review quality for Mozilla, a large open-source project. We explore the relationships between the reviewers' code inspections and a set of factors, both personal and social in nature, that might affect the quality of such inspections. We applied the SZZ algorithm to detect bug-inducing changes that were then linked to the code review information extracted from the issue tracking system. We found that 54% of the reviewed changes introduced bugs in the code. Our findings also showed that both personal metrics, such as reviewer workload and experience, and participation metrics, such as the number of involved developers, are associated with the quality of the code review process.

***Index Terms*—Code review, code review quality, bug-inducing changes, mining software repositories, Mozilla, empirical study.**

I. Introduction

Code review is an essential element of any mature software development project; it aims at evaluating code contributions submitted by developers. Code review is considered to be one of the most effective QA practices for software projects; it is relatively expensive in terms of time and effort, but provides good value in identifying defects in code changes before they are committed into the project's code base [1]. Reviewers are the gatekeepers of a project's master repository; they must carefully validate the design and implementation of patches to ensure they meet the expected quality standards.

In principle, the code review process should improve the quality of code changes (patches) before they are committed to the project's master repository. However, in practice, the execution of this process can still allow bugs to enter into the codebase unnoticed. This work aims to investigate the quality of code review.

In this paper, we have studied code review of a large open source system: the Mozilla project. For Mozilla, code review is an important and vital part of their development process, since contributions may come not only from core Mozilla developers but also from the greater user community. The Mozilla community embraces code review to help maintain consistent design and implementation practices among the many casual contributors and across the various modules that comprises the Mozilla codebase [2]. They have found that code review increases code quality, promotes best practices, and reduces regressions [3].

The Mozilla code review process requires that every submitted patch be evaluated by at least one reviewer [4]. Reviewers are advised to grant approval to a patch if 1) they believe that the patch does no harm, and 2) the patch has test coverage appropriate to the change. If a reviewer feels unable to provide a timely, expert review on a certain patch, they can re-direct the patch to other reviewers who may be better able to perform the task. However, even given the careful scrutiny that patches undergo, software defects are still found after the changes have been reviewed and committed to the version control repository. These post-release defects raise red flags about the quality of code reviews. Poor-quality reviews that permit bugs to sneak in unnoticed can introduce stability, reliability, and security problems, affecting the user's experience and ultimately their satisfaction with the product.

Previous research on code review has examined topics such as the relation between code coverage/participation and software/design quality [5], [6]; however, the topic of code review quality remains largely unexplored. In this paper, we perform an empirical case study of a large open source Mozilla project including its top three largest modules: *Core*, *Firefox*, and *Firefox for Android*. We apply the SZZ algorithm [7] to detect bug-inducing changes that are then linked to the code review data extracted from the issue tracking system. We address the following overarching research questions:

RQ1: *Do code reviewers miss many bugs?*
The goal of code review is to identify problems (e.g., the code-level problems) in the proposed code changes. Yet, software systems remain bug-prone.

RQ2: *Do personal factors affect the quality of code reviews?*
Previous studies found that code ownership has a strong relationship with both pre- and post-release defect-proneness [8]–[10].

RQ3: *Does participation in code review influence its quality?*
A recent study demonstrated that low review participation has a negative impact on software quality [5].

Paper organization. The rest of the paper is organized as follows. We first provide some background about the Mozilla code review process in Section II. Section III describes the methodology followed in this study. In Section IV, we present and discuss the results of our three research questions. In Section V, we address the threats to validity. In Section VI, we discuss related research. Finally, Section VII summarizes our results, and highlights future work.

II. The Mozilla Code Review Process

Mozilla employs a two-tiered code review process for assessing submitted patches: *review* and *super review* [2]. A *review* is performed by the owner of the module (or peers of the module) in question; reviewers thus have domain expertise in the specific problem area of concern. *Super reviews* are required if the patch involves integration or modifies core Mozilla infrastructure (e.g., major architectural refactoring, changes to API, or changes that affect how code modules interact). Currently, there are 30 super-reviewers for all Mozilla modules [3], 162 reviewers for the *Core* module, 25 reviewers for *Firefox*, and 11 reviewers for *Firefox for Android* (also called "Fennec") [11]. However, any person who is not on the list of designated reviewers but has level three commit access — i.e., core product access to the Mercurial version control system — can review a patch.

The Mozilla team reviews every patch using the Bugzilla issue-tracking system, which records and stores all information related to code review tasks. The process works this way. First, a developer submits a patch to Bugzilla that contains their proposed code changes; they then request a review from a designated reviewer of the module where the code will be checked in. Patches that have significant impact on the design of Mozilla may trigger a super review. After the code has been reviewed, the reviewer will indicate a positive or negative outcome, and may provide feedback comments. Once the reviewers approve a patch, the code changes are committed to Mozilla's source code repository.

As we can see, Mozilla employs a strict review policy. Investigating what makes developers miss bugs in code during review process is the topic of our work.

III. Methodology

To address our research questions we followed a data mining process shown in Figure 1 that consists of the following stages. First, we extracted commits from the Mozilla's version control repository (step 1). We then linked these commits to the corresponding bug reports in the Bugzilla issue tracking system (step 2). After that, we extracted information about linked bug reports and review-related information for patches attached to them (steps 3 and 4). Finally, we established the links between commits and reviewed patches (step 5) and identified bug-inducing commits (step 6).

A. Studied Systems

Mozilla uses Mercurial as their version control system and maintains several repositories, with each repository built around a specific purpose and/or set of products. We considered `mozilla-central`[1] as the main repository; it contains the master source code for Firefox and Gecko, Mozilla's layout engine.

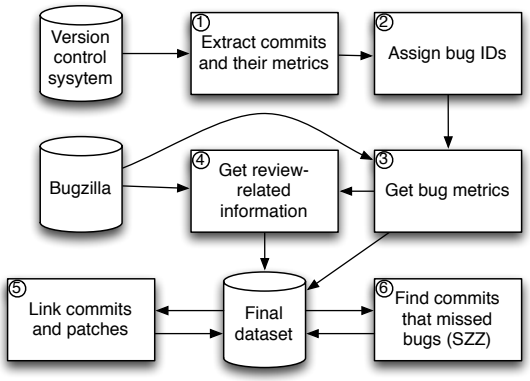

Fig. 1: Process overview.

TABLE I: Overview of the studied systems.

System	Commits	Reviews	Writers	Reviewers
Mozilla-all	27,270	28,127	784	469
Core	18,203	18,759	544	362
Firefox	2,601	2,668	214	110
Firefox for Android	2,106	2,160	108	72

For our study, we took all code changes that were committed to `mozilla-central` between January 1, 2013 and January 1, 2014. In this paper, we use terms "code change" and "commit" interchangeably. We studied four systems: *Mozilla-all* (full set of commits), as well as the three largest modules: *Core*, *Firefox*, and *Firefox for Android*. Table I describes the main characteristics of these systems; the numbers represent "clean" datasets that we obtained after performing the steps described in Sections III-B, III-C, and III-D. We report the number of commits, reviews, writers, and reviewers for our Mozilla-all, Core, Firefox, and Firefox for Android datasets.

B. Data extraction

We extracted a total of 44,595 commits from `mozilla-central`. During the extraction phase, we collected a variety of information about each commit including its unique identifier, the name and email address of the person who made this commit, the date it was added to the repository, the textual description of a change, and the size statistics of the commit. To calculate the size statistics, we analyzed the `diff` statements of each commit. We looked only at textual `diffs`, and we excluded those that describe a change in binary files such as images. We recorded the number of files changed, the total number of lines that were added and removed, and the total number of code chunks found in the investigated `diffs`.

Linking revisions to bugs. Prior to identifying bug-inducing changes, we had to detect changes that aim to fix bugs. For that, we linked commits in the version control repository to bugs in the issue tracking system using the

[1] http://hg.mozilla.org/mozilla-central

commit descriptions. Manual inspection of commit summaries confirmed that developers consistently include a bug ID in the commit summary, and also tend to use the same formatting. Based on this finding, we wrote a set of case-insensitive regular expressions to extract bug ID values. If a regular expression found a match, we checked whether a commit description contains any review flags to eliminate matches from unreviewed commits. If such flags were found and commits contained bug ID numbers, we linked bug ID numbers to them.

As a result of this, we were able to assign bug ID values to 35,668 (80%) commits. As suggested by Kim et al. [12], we manually checked summaries of both matched and non-matched commits and found no incorrectly assigned bug IDs. The analysis of non-matched commits (8,927 in total) showed that 2,825 commits (6.3%) were *backed out* commits, 5,520 (12.3%) commits were *merges*, 413 (1%) of them were *"no bug"* commits, and 169 of them were *other* commits.

Getting additional data from Bugzilla. We scraped Bugzilla for each linked bug ID to get detailed information, including the date when the bug was submitted, the name and email address of the person who reported the bug, the bug severity and priority, the module affected by the bug, and the list of proposed patches. For each patch, we recorded the author of the patch, the submission date, and the review-related flags. For each review-related flag, we extracted the date and time it was added, as well as the email address of the person who performed the flag update.

Out of 22,015 unique bug IDs assigned to the commits, we were unable to extract the data for 188 bugs that required special permissions to access them. For 490 bugs, we did find no patches with review-related flags. Such a situation might arise in only two cases: if we incorrectly assigned bug ID in the first place, or if a patch landed into the code base without undergoing the formal code review process. To investigate this, we performed a manual check of several randomly-selected bug IDs. We found no examples of incorrect assignment: all of the checked bug IDs were bugs with no reviewed patches in Bugzilla. Since commits having no information about reviews can not contribute to our study, we disregarded them, reducing the number of unique bug IDs by 678 and the number of commits in the dataset to 34,654.

C. Linking Patches and Commits

Since each commit in the version control system is typically associated with a single patch, we linked each commit to its corresponding patch and its review-related information. However, establishing these links requires effort. The best matching of a patch to a commit can be achieved by comparing the content of the commit to the contents of each patch, and then verifying if the two are the same. However, this approach does not work in the environment where developers constantly make commits to the repository independently from one another. For example, a patch p_1 was added to Bugzilla at time t_1 and was committed to the repository at time t_2. If there were no changes to the files affected by the patch between t_1 and t_2, the commit and the patch would be the same. If another patch p_2 changing some of those files was committed to the repository during that time frame, the content of the commit of p_1 might not match the content of the patch p_1 itself. The most precise way of matching patches and commits would be to employ some code cloning techniques to detect matches on the string level; however, applying such techniques was beyond the scope of our paper.

In our approach, we decided to opt for a less precise but conservative way of mapping commits to patches. For each commit with a bug ID attached, we took all reviewed patches ordering them by their submission date. We then searched for the newest patch such that (1) the last review flag on that patch was `review+` or `super-review+`, and (2) this review was granted before the changes were committed to the version control system. Previous research showed that patches can be rejected after they receive a positive review [13], [14]. The first heuristic makes sense as patches with last review flags being `review-` are unlikely to land into the code. On the contrary, patches that were first rejected and later accepted (e.g., another more experienced reviewer reverted a previous negative review decision) are likely to be incorporated into the code base. The second heuristic ensures that changes can not be committed without being reviewed first; it facilitates proper mapping when several commits in the version control system are linked to the same bug, and there are multiple patches on that bug. For example, a bug can be fixed, reopened, and fixed again. In this case, we would have two different patches linked to two commits; without the second heuristic, the same patch would be linked to both commits.

By applying these heuristics, we were able to successfully link 28,888 out of a total of 34,654 (i.e., 83%) commits to appropriate patches. The manual inspection of the remaining 17% of the commits revealed that the main reason why we did not find corresponding patches in Bugzilla was incorrect date and time values of the commits when they were added to the version control system. For example, a commit with ID `147321:81cee5ae7973` was "added" to the repository on 2013-01-28; the bug ID value assigned to this commit is 904617. Checking this bug history in Bugzilla revealed that the bug was reported on 2013-08-13, almost *7 months after* it was fixed.

D. Data Pre-Processing

Prior to data analysis, we tried to minimize noise in our data. To eliminate outliers, we performed data cleanup by applying three filters:

1) We removed the largest 5% of commits to account for changes that are not related to bug fixes but rather to global code refactoring or code imports (e.g., libraries). Some of the commits are obvious outliers in terms of size. For example, the largest commit ("Bug 724531 - Import ICU library into Mozilla tree") is about 1.1 million lines of code, while the median value for change size is only 34 lines of code. This procedure removed all commits that were larger than 650 lines (1,403 commits in total).

2) Some changes to binary files underwent code review. However, since the SZZ algorithm can not be applied to such changes, we removed the commits containing only binary `diffs` (52 commits in total).
3) We found that for some changes the submission date of their associated patches was before the start of our studied period. We believe that these patches fell on the floor but later were found and reviewed. To eliminate these outliers, we removed all commits representing patches that were submitted before 2012-09-01. This filter excluded 163 commits.

Our final dataset[2] contains 27,270 unique commits, which corresponds to 28,127 reviews (some linked patches received multiple positive reviews, thus, commits can have more than one review).

E. Identifying Bug-Inducing Changes

To answer our research questions, we had to identify reviews that missed bugs, i.e., the reviews of the patches that were linked to bug-inducing commits. We applied the SZZ algorithm proposed by Śliwerski et al. [7] to identify the list of bug-inducing changes. For each commit that is a bug fix, the algorithm executes `diff` between the revision of the commit and the previous revision. In Mercurial, all revisions are identified using both a sequential numeric ID and an alphanumeric hash code. Since Mercurial is a distributed version control system, `RevisionId - 1` is not always a previous revision and thus cannot be used in the algorithm. To overcome this problem, we extracted the parent revision identifier for each revision and used it as a previous revision value for executing `diff`. The output of `diff` produces the list of lines that were added and/or removed between the two revisions. The SZZ algorithm ignores added lines and considers removed lines as locations of bug-introducing changes.

Next, the Mercurial `annotate` command (similar to `blame` in Git) is executed for the previous revision. For each line of code, `annotate` adds the identifier of the most recent revision that modified the line in question. SZZ extracts revision identifiers for each bug-introducing line found at the previous step, and builds the list of revisions that are candidates for bug-inducing changes.

Kim et al. addressed some limitations of the SZZ algorithm as it may return imprecise results if `diff` contains changes in comments, empty lines, or formatting [17]. The problem with false positives (precision) occurs because SZZ treats those changes as bug-introducing changes even though such changes have no effect on the execution of the program. Since we implemented SZZ according to the original paper, i.e., without any additional checks, we wanted to find out how many false positives are returned by SZZ. To assess the accuracy of the SZZ algorithm, we performed a manual inspection of the returned results (that is, potential candidates returned by SZZ) for 100 randomly selected commits. We found 9% (39 out of 429 candidates) of false positives with 19 of those being changes in formatting and the rest 20 candidates being changes in comments and/or empty lines. While we think the percentage of false positives is relatively small, the limitations of SZZ remain a threat to validity.

Finally, the algorithm eliminates those candidates that were added to the repository after the bug associated with a commit was reported to the issue tracking system. The remaining revisions are marked as bug-inducing code changes.

We ran the SZZ algorithm on every commit with a bug ID, and obtained the list of changes that led to bug fixes. Some of the changes might have been "fixed" outside of our studied period and thus would not be marked as bug-inducing. To account for such cases, we also analyzed the changes that were committed within a six-month time frame after our studied period: we assigned bug ID values, scraped Bugzilla for bug report date, and executed the SZZ algorithm; the results were added to the list of bug-inducing commits. The commits from the dataset were marked as bug-inducing if they were present in this list; otherwise, they were marked as bug-free commits.

F. Determining Explanatory Factors

Previous work demonstrated that various types of metrics have relationship with code review time and outcome [14]. Similarly, we grouped our metrics into three types: *technical*, *personal*, and *participation*.

Table II describes the metrics used in our study and provides rationale for their selection. The metrics for technical factors were calculated on our dataset. However, the personal and participation metrics could not be extracted from our data due to its fixed time frame. For example, one developer started to participate in code review in 2013, while another one has been performing review tasks since 2010; if we compute their expertise on the data from our dataset (i.e., a 12-month period of 2013), the experience of the second developer will be incorrect, i.e., his experience for previous three years (2010–2012) will be not taken into account. To overcome this problem, we queried an Elastic Search cluster containing the complete copy of the data from Bugzilla [18]. The nature of how Elastic Search stores the data allowed us to get the "snapshots" of Bugzilla for any point in time and to accurately compute the personal and participation metrics. While computing the review queue values, we found that many developers have a noticeable number of "abandoned" review requests, i.e., the requests that were added to their loads but never completed. Such requests have no value for the `review queue` metric; therefore, any pending review request on the moment of 2014-01-01 was ignored when calculating developer review queues.

The metrics of three types presented in this section served as explanatory variables for building our models that we describe next.

G. Model Construction and Analysis

To study the relationship between personal and participation factors and the review quality of the studied systems, we built

[2]https://cs.uwaterloo.ca/~okononen/bugzilla_public_db.zip

TABLE II: A taxonomy of considered technical, personal, and participation metrics used.

Type	Metric	Description	Rationale
Technical	Size (LOC)	The total number of added and removed lines of code.	Large commits are more likely to be bug-prone [15]; thus the intuition is it is easier for reviewers to miss problems in large code changes.
	Chunks	The total number of isolated places (as defined by `diff`) inside the file(s) where the changes were made.	We hypothesize that reviewers are more likely to miss bugs if the change is divided into multiple isolated places in a file.
	Number of files	The number of modified files.	Similar, reviews are more likely to be prone to bugs if the change spread across multiple files.
	Module	The name of the Mozilla module (e.g., Firefox).	Reviews of changes within certain modules are more likely to be prone to bugs.
	Priority	Assigned urgency of resolving a bug.	Our intuition is that patches with higher priority are more likely to be rushed in and thus be more bug-prone than patches with lower priority levels.
	Severity	Assigned extend to which a bug may affect the system.	We think that changes with higher levels of severity introduce less bugs because they are often reviewed by more experienced developers or by multiple reviewers.
	Super review	Indicator of whether the change required super review or not	Super review is required when changes affect core infrastructure of the code and, thus, more likely to be bug-prone.
	Number of previous patches	The number of patches submitted before the current one on a bug.	Developers can collaborate on resolving bugs by submitting improved versions of previously rejected patches.
	Number of writer's previous patches	The number of previous patches submitted by the current patch owner on a bug.	A developer can continue working on a bug resolution and submit several versions of the patch, or so called resubmits of the same patch, to address reviewers concerns.
Personal	Review queue	The number of pending review requests.	While our previous research [14] demonstrated that review loads are weakly correlated with review time and outcome; we were interested to find out whether reviewer work loads affect code review quality.
	Reviewer experience	The overall number of completed reviews.	We expect that reviewers with high overall expertise are less likely to miss a bug.
	Reviewer experience for module	The number of completed reviews by a developer for a module.	Reviewers with high reviewing experience in a certain module are less likely to miss defects; and on the contrary, reviewers with no past experience in performing code reviews for some modules are more likely to fail to catch bugs.
	Writer experience	The overall number of submitted patches.	Developers who contribute a lot to the project — have high expertise — are less likely to submit buggy changes.
	Writer experience for module	The number of submitted patches for a module.	Developers who make few changes to a module are more likely to submit buggy patches.
Participation	Number of developers on CC	The number of developers on the CC list at the moment of review decision.	Linus's law states that "given enough eyeballs, all bugs are shallow" [16].
	Number of comments	The number of comments on a bug.	The more discussion happens on a bug, the better the quality of the code changes [5].
	Number of commenting developers	The number of developers participating in the discussion around code changes.	The more people are involved in discussing bugs, the higher software quality [5].
	Average number of comments per developer	The ratio of the comment count over the developer count.	Does the number of comments per developer has an impact on review quality?
	Number of reviewer comments	The number of comments made by a reviewer.	Does reviewer participation in the bug discussion influence the quality of reviews?
	Number of writer comments	The number of comments made by a patch writer.	Does patch writer involvement in the bug discussion affect review quality?

Multiple Linear Regression (MLR) models. Multiple linear regression attempts to model the relationship between two or more explanatory variables and a response variable by fitting a linear equation to observed data [19]. The model is presented in the form of $y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + ... + \beta_n x_n$, where y is the response variable and x_1, x_2,... x_n are explanatory variables. In our MLR models, the response variable is the code review quality (buggy or not) and the explanatory variables are the metrics described in Table II. The value of the response variable ranges between 0 and 1 — we used the value of 1 for bug-prone reviews and the value of 0 for bug-free inspections. Our goal was to explain the relationship (if any) between the explanatory variables (personal and participation metrics) and the response variable (code review quality). In our models we control for several technical dependent factors (size, number of files, etc.) that are likely to influence the review quality. We build our models similar to the ones described in [5], [14], [20], [21].

Transformation. To eliminate the impact of outliers on our models, we applied a log transformation $log(x + 1)$ to the metrics whose values are natural numbers (e.g., size, chunks, number of files, experience, review queues, etc.). Since categorical variables can not be entered directly into a regression model and be meaningfully interpreted, we transform such variables (e.g., priority, severity, etc.) using a "dummy coding" method, which is a process of creating dichotomous variables from a categorical variable. For example, if we have a categorical variable such as `priority` that has 5 levels (P1–P5), then four dichotomous variables are constructed that contain the same information as the single categorical variable. By using these dichotomous variables we were able to enter

the data presented by categorical metrics directly into the regression model.

Identifying Collinearity. Collinearity, or excessive correlation among explanatory variables, can complicate or prevent the identification of an optimal set of explanatory variables for a statistical model. We identified collinearity among explanatory variables using the variance inflation factor (VIF). A VIF score for each explanatory variable is obtained using the R-squared value of the regression of that variable against all other explanatory variables. After calculating VIF scores, we removed those with high values. The VIF score threshold was set to 5 [22], thus if the model contained a variable with VIF score greater than 5, this variable was removed from the model and VIF scores for the variables were recalculated. We repeated this step until all variables in our model had VIF scores below the threshold.

Model Evaluation. We evaluated our models by reporting the Adjusted R^2 values. We also performed a stepwise selection [23], a method of adding or removing variables based solely on the *t*-statistics of their estimates. Since we had many explanatory variables, it was useful to fine tune our model by selecting different variables. Our goal was to identify the best subset of the explanatory variables from our full model. For the stepwise variable selection, we applied both the "forward" and "backward" methods.

IV. Results

In this section, we present and discuss the results of our empirical study performed on various Mozilla systems.

RQ1: Do code reviewers miss many bugs?

In theory, code review should have a preventive impact on the defect-proneness of changes committed to the project's source code. Yet, code review might induce bugs since identifying the code-level problems and design flaws is not a trivial task [24]. We determine the proportion of buggy code reviews for the different projects by computing the number of bug-inducing code reviews for each Mozilla module.

As indicated in Table IV, we find that overall 54% of Mozilla code reviews missed bugs in the approved commits. This value proved to be remarkably consistent across the different modules we looked at: the Core module contained 54.3% buggy reviews, Firefox contained 54.2%, and Firefox for Android contained 56%. While the studied systems are of widely varying sizes and have different numbers of commits and reviewers (as reported in Table I), the proportion of "buggy" code reviews in these modules is almost identical.

While we were surprised to see such minute variation across the different modules, the proportion of buggy changes (54–56%) we observed is within the limits of the previously reported findings. Kim et al. [12] reported that the percentage of buggy changes can range from 10% to 74% depending on the project; with Mozilla project having 30% of buggy changes for the 2003–2004 commit history when Mozilla code base was still growing. Śliwerski, Zimmerman, and Zeller [7] found 42% of bug-inducing changes for Mozilla and 11% for Eclipse projects (the dataset contained changes and bugs before 2005).

TABLE IV: Number of code reviews that missed bugs.

System	# Reviews	# Buggy Reviews	% Buggy Reviews
Mozilla-all	28,127	15,188	54.0 %
Core	18,759	10,184	54.3 %
Firefox	2,668	1,447	54.2 %
Firefox4Android	2,160	1,210	56.0 %

RQ2: Do personal factors affect the quality of code reviews?

Intuitively, one would expect that an experienced reviewer would be less likely to miss design or implementation problems in code; also, one would expect smaller work loads would allow reviewers to spend more time on code inspections and, thus, promote better code review quality. To investigate if these are accurate assumptions, we added technical and personal metrics from Table II to our MLR model.

Table IIIa shows that review queue length has a statistically significant impact on whether developers catch or miss bugs during code review for all the four studied systems. The regression coefficients of the review queue factor are positive, demonstrating that reviewers with longer review queues are more likely to submit poor-quality code evaluations. These results support our intuition that heavier review loads can jeopardize the quality of code review. A possible improvement would be to "spread the load on key reviewers" [25] by providing a better transparency on developer review queues to bug triagers.

For all studied systems, reviewer experience seems to be a good predictor of whether the changesets will be effectively reviewed or not. Negative regression coefficients for this metric demonstrate that less experienced developers — those who have conducted relatively fewer code review tasks — are more likely to neglect problems in changes under review. These results follow our intuition about reviewer experience being a key factor to ensure the quality of code reviews. It was surprising to us that writer experience (overall or module-based) does not appear to be an important attribute in most of the models (with the exception of Core). We expected to see that less active developers having little experience in writing patches would be more likely to submit defect-prone contributions [9], [26] and thus, increase the chances of reviewers in failing to detect all defects in poorly written patches.

Factors such as the number of previous patches on a bug and the number of patches re-submitted by a developer seem to have a positive effect on review quality for one of the four systems: Firefox for Android (and also on the overall Mozilla-all). A possible explanation is that Firefox for Android is a relatively new module, and the novelty of the Android platform may attract a variety of developers to be more involved in contributing to the Android-based browser support building on each other's work (i.e., improving previously submitted patches). However, we have not attempted to test this hypothesis.

Among the technical factors, size of the patch has a statistically significant effect on the response variable in all four

TABLE III: Model statistics for fitting data. Values represent regression coefficients associated with each variable.

(a) Technical and personal factors.

	Mozilla	**Core**	**Firefox**	**FF4A**
Adjusted R^2	0.128	0.123	0.173	0.138
Size (LOC)	0.102***	0.098 ***	0.108***	0.115***
Chunks	†	†	†	†
Number of files	0.058***	0.059 ***	0.109***	0.062*
Module	⋆	n/a	n/a	n/a
Priority	⋆	⋆	‡	·
Severity	‡	‡	·	‡
Super review	-0.139**	-0.177***	·	n/a
Review queue	0.017***	0.0204***	0.038**	0.045**
Reviewer exp.	-0.013***	-0.012***	-0.029***	-0.041***
Reviewer exp. (mod.)	†	†	‡	0.018*
Writer exp.	·	-0.004*	‡	‡
Writer exp. (module)	†	†	‡	·
# prev patches	†	†	†	-0.045***
# writer patches	-0.012***	·	·	†

†Disregarded during VIF analysis (VIF coefficient > 5).
⋆ "It's complicated": for categorical variables see explanation of the results in Section IV.
FF4A = Firefox for Android.

(b) Technical and participation metrics.

	Mozilla	**Core**	**Firefox**	**FF4A**
Adjusted R^2	0.134	0.128	0.173	0.147
Size (LOC)	0.105***	0.103***	0.105***	0.117***
Chunks	†	†	†	†
Number of files	0.060***	0.059***	0.090***	0.067***
Module	⋆	n/a	n/a	n/a
Priority	‡	⋆	‡	⋆
Severity	⋆	‡	⋆	‡
Super review	-0.124***	-0.160***	‡	n/a
# of devs on CC	0.053***	0.056***	‡	0.049*
# comments	†	†	†	†
# commenting devs	-0.124***	-0.102***	-0.075***	-0.176***
# comments/ # dev	-0.039***	-0.029**	‡	‡
# reviewer comments	0.010**	‡	0.026*	‡
# writer comments	·	·	·	-0.047**

‡Disregarded during stepwise selection.
Statistical significance: '***' $p < 0.001$; '**' $p < 0.01$; '*' $p < 0.05$; '·' $p \geq 0.05$.

models. Its regression coefficients are positive, indicating that larger patches lead to a higher likelihood of reviewers missing some bugs. Similarly, number of files has a good explanatory power in all four systems. The need for a super review policy is well explained, as super reviews have a positive impact on the review quality. This shows that such reviews are taken seriously by Mozilla-all and Core projects (our dataset contains no super reviews for Firefox for Android). It is not surprising as the role of super reviewer is given to highly experienced developers who demonstrated their expertise of being a reviewer in the past and who has a greater overall knowledge of the project's code base.

When examining the impact of `module` factor on code review effectiveness, we noticed that for some Mozilla modules such as Core, Fennec Graveyard, Firefox, Firefox for Metro, Firefox Health Report, Mozilla Services, productmozilla.org, Seamonkey, Testing, and Toolkit, the model contains negative regression coefficients that are statistically significant; this indicates that these modules maintain a better practice of ensuring high quality of their code review process.

We found that while the bug priority level is associated with a decrease of poorly conducted reviews (P5 patches for Mozilla-all with regression coefficient being -0.13, $p < 0.05$ and P3 patches for Core module with the regression coefficient = -0.10, $p < 0.05$), it does not have a significant impact on other two modules.

RQ3: Does participation in code review influence its quality?

Previous research found that the lack of participation in code review has a negative impact on quality of software systems [5]. To investigate whether code review quality is affected by the involvement of the community, we added metrics that relate to developer participation in review process, described in Table II to our models.

Table IIIb shows that the number of developers on the CC list has a statistically significant impact on review bugginess for three of the four systems; and its regression coefficients are positive, indicating that the larger number of developer names is associated with the decrease in review quality. This may sound counterintuitive at first. However, from the developer perspective, their names can be added to CC for different reasons: developer submitted the bug, wrote a patch for the bug, wants to be aware of the bug, commented on the bug, or voted on the bug. Thus, we think that CC is negatively associated with review quality due to its ambiguous purpose: "CC is so overloaded it doesn't tell you why you are there" [25].

The number of commenting developers has a statistically significant impact on the models of all four of the studied systems. The regression coefficients are negative, indicating that the more developers that are involved in the discussion of bugs and their resolution (that is, patches), the less likely the reviewers are to miss potential problems in the patches. A similar correlation exists between review quality and the metric representing average number of comments per developer and having statistically significant negative coefficients for two of the four systems (Mozilla-all and Core). This shows that reviews that are accompanied with a good interactions among developers discussing bug fixes and patches are less prone to bugs themselves. The number of comments made by patch owners is also demonstrated to have a statistically significant negative impact on review bug-proneness in the model for Firefox for Android only. These results reveal that the higher rate of developer participation in patch discussions is associated with higher review quality.

While any developer can collaborate in bug resolution or participate in critical analysis of submitted patches, reviewers typically play a leading role in providing feedback on the patches. Thus, we expected to see that the number of comments made by reviewers has a positive correlation with review quality. However, in Table IIIb we can see that while having a statistically significant impact in the models for two of the four systems, the regression coefficients are positive, indicating

that more reviewers participate in discussing patches, the more likely they would miss bugs in the patches they review. A possible explanation of these surprising results is that if a reviewer posts many comments on patches, it is possible that he is very concerned with the current bug fix (its implementation, coding style, etc.). Or, as our previous qualitative study revealed, the review process can be sensitive due to its nature of dealing with people's egos [25]. As one developer mentions "there is no accountability, reviewer says things to be addressed, there is no guarantee that the person fixed the changes or saw the recommendations." Code review is a complex process involving personal and social aspects [27].

Table IIIb demonstrated that while developer participation has an effect on review quality, technical attributes such as patch size and super review are also good predictors. All models suggest that the larger the code changes, the easier it is for reviewers to miss bugs. However, if changes require a super review, they are expected to undergo a more rigorous code inspections. For two of the three studied systems, super review has negative regression coefficients; but it does not have a significant impact for Firefox (Firefox for Android patches have no super reviews).

Code reviews in `modules` such as Core, Fennec Graveyard, Firefox, Firefox for Metro, Firefox Health Report, Mozilla Services, productmozilla.org, Seamonkey, Testing, and Toolkit are statistically less likely to be bug-prone; the regression coefficients for these modules have negative values and are -0.209 ($p < 0.05$), -0.339 ($p < 0.01$), -0.191 ($p < 0.05$), -0.205 ($p < 0.05$), -0.197 ($p < 0.05$), -0.263 ($p < 0.05$), -0.679 ($p < 0.001$), -0.553 ($p < 0.01$), -0.204 ($p < 0.05$) and -0.297 ($p < 0.001$) respectively. Similar to the findings for RQ2, code inspections performed in these modules appear to be more watchful than in other components.

Priority as a predictor has a statistically significant impact on review outcome for Core and Firefox for Android only. For the Core module, priority P3 has a negative effect (regression coefficient = -0.09, $p < 0.05$), i.e., the patches with P3 level are expected to undergo more careful code inspections. For Firefox for Android, patches with priority P1 are more likely to be associated with poor reviews (regression coefficient = 0.17, $p < 0.001$). Among severity categories, we found that patches of trivial severity have statistically significant negative impact on review bug-proneness in the models for Mozilla-all and Firefox (regression coefficients =-0.125 and =-0.385 $p < 0.05$, respectively). Developers find that "priority and severity are too vague to be useful" [25] as these fields are not well defined in the project. But since these metrics are associated with the review quality, developer should be given some estimation of the risks involved to decide how much time to spend on patch reviews.

While the predictive power of our models remain low (even after rigorous tune-up efforts), the best models appear to be for Firefox (Adjusted R^2 = 0.173 for fitting technical and personal factors, as well as technical and participation metrics). The goal in this study is not to use MLR models for predicting defect-prone code reviews but to understand the impact our personal and participation metrics have on code review quality, while controlling for a variety of metrics that we believe are good explainers of review quality. Thus, we believe that the Adjusted R^2 scores should not become the main factor in validating the usefulness of this study.

V. Threats to Validity

External validity. Our findings cannot be generalized across all open source projects. While we only study one (large) open source community, we considered various Mozilla modules including Core, Firefox, and Firefox for Android. Our goal was to study a representative open source system in detail. Nevertheless, further research studies are needed to be able to provide greater insight into code review quality.

Internal validity concerns with the rigour of the study design. In this study, the heuristics used, as well as the data filtering techniques adopted may be a threat. We mitigate such a threat by providing details on the data extraction and filtering and by using a well-known outliers filtering procedure. The choice of metrics may be seen as a threat. We selected widely used metrics characterizing code review activities, bugs, code changes (patches/commits), and developer attributes.

We assume that a code review is documented and communicated via Bugzilla issue tracking system. While this assumption holds in most cases, some code review tasks can be carried out via other channels such as email, face-to-face meetings, etc. When investigating the relation between code change and reviewer, we assumed that patches are independent; this might have introduced some bias since several different patches can often be associated with the same bug ID and "mentally" form one large patch. In our study we considered that the most recent patch is the one that gets incorporated into the code.

In Bugzilla, bug reports per se actually serve several purposed: they can be bug-fix requests, or requests for adding new functionality, or documentation-related changes, etc. Since Bugzilla does not provide mechanisms of distinguishing between "true" bugs and new feature requests, we treat all changes as bug fixes.

When calculating review queue length of developers, we assume that at any given point the number of review requests for a developer defines his or her current review load. This heuristic is a "best effort" approximation; accurate review loads are hard to determine by scraping the data from the existing code review system.

Conclusion validity is the degree to which conclusions we reach about relationships in our data are reasonable. Proper regression models were built for the sake of showing the impact of studied factors on the code reviews bugginess. In particular, we built our MLRs for two types of metrics and evaluated them based on the appropriate measures such as the deviance explained and Adjusted R^2.

VI. Related Work

In this section, we present the main contributions relevant to code review, software quality, as well as code review and software quality.

A. Code Review

Rigby and German have shown the existence of a number of review patterns and quantitatively analyzed the review process of the Apache project [28]. Later, Rigby and Storey discovered that the identification of defects is not the sole motivation for modern code review. Other motivations exist including non-technical issues such as feature, scope, or process issues [29].

This finding is inline with those by Baysal et al. who have shown that review positivity, i.e., the proportion of accepted patches, is also influenced by non-technical factors [14]. Organizational (the company) and personal dimensions (reviewer load and activity, patch writer experience) influence review timeliness, as well as the likelihood of a patch being accepted [14]. This corroborates the results by Nagappan et al. who demonstrated that organizational metrics are better predictors of defect-proneness than traditional measures [30].

Jiang et al. have empirically shown that developer experience, patch maturity, and priori subsystem churn affect the patch acceptance while reviewing time is impacted by submission time, the number of affected subsystems, the number of suggested reviewers and developer experience [31].

A recent qualitative study at Microsoft revealed that while finding defects remains the main motivation for review, other motivations exist such as knowledge sharing among team members [24].

B. Software Quality

Researchers have studied the impact of design and code review on software quality. For example, Kemerer et al. have empirically shown that allowing sufficient preparation time for reviews and inspections can produce better performance [32]. Other recent works have studied software risky changes. For example, Śliwerski et al. suggested a technique called, SZZ, to automatically locate fix-inducing changes by linking a version archive to a bug database [7]. In this study we applied SZZ to detect bug-inducing changes which we later link to code review data. SZZ was successfully applied to understand whether refactorings inducing bug-fixes [33], as well as to build prediction models that focus on identifying defect-prone software changes [34]. Eyolfson et al. analyzed the relation between a change bugginess and the time of the day the change was committed and the experience of the developer who made the change [26]. Several other technical metrics have been proposed as well [35]–[39].

Rahman and Devanbu suggested that code review is an essential part of the software quality assurance [9] while Mende and Koschke [40] have proposed effort-aware bug prediction models to help allocate software quality assurance efforts including code review.

C. Code Review and Software Quality

Recently, McIntosh et al. empirically shown that code review coverage and participation significantly impact on software quality. These results confirm that poor code review negatively affect software quality [5]. Beller et al. found that the types of changes due to the modern code review process in OSS are similar to those in the industry and academic systems from literature, featuring a similar ratio of maintainability-related to functional problems [41]. Mäntylä and Lassenius suggest that code reviews may be most valuable for long-lived software products [42] while Hatton [43] found relevant differences in defect finding capabilities among code reviewers.

Previous research agree that personal and organizational factors have a significant impact on software quality. Understanding these factors, and properly allocating people resources can help managers enhance quality outcomes. We hypothesize that a modern code review process can neglect buggy changes and that this may be due to several factors technical, personal, and organizational.

VII. Conclusion

Code review is an essential and vital part of modern software development. Code review explicitly addresses the quality of contributions before they are integrated into project's codebase. Due to volume of submitted contributions and the need to handle them in a timely manner, many code review processes have become more lightweight and less formal in nature. This evolution of review process increases the risks of letting bugs slip into the version control repository, as reviewers are unable to detect all of the bugs.

In this paper, we have explored the topic of code review quality by investigating what factors might affect it. We tried to understand what aspects contribute to poor code review quality to help software development projects better understand their processes and practice. We built and analyzed MLR models to explain the relationships between personal characteristics of developers, team participation and involvement in code review, and technical properties of contributions on the effectiveness of code review.

Our findings suggest that developer participation in discussions surrounding bug fixes and developer-related characteristics such as their review experience and review loads are promising predictors of code review quality for all studied systems. Among technical properties of a change, its size, the number of files it affects, its impact on the rest of the project's code (or the need to perform a super review) have also a significant link with the review bug-proneness. We believe that these findings provide practitioners with strong empirical evidence for revising current code review policies and promoting better transparency of the developers' review queues and their expertise on the modules.

While in this work we have provided only a quantitative investigation of what factors may influence code review quality (with some triangulation with our previous qualitative study), as a part of our future work we plan to conduct an extensive qualitative study with Mozilla developers. We are working on designing a survey and want to conduct interviews and observations with developers to study the reasons of why they miss bugs when performing code review tasks.

REFERENCES

[1] M. E. Fagan, "Design and code inspections to reduce errors in program development," *IBM Syst. J.*, vol. 15, no. 3, pp. 182–211, Sep. 1976.

[2] Mozilla, "Code Review FAQ," https://developer.mozilla.org/en/Code_Review_FAQ, February 2015.

[3] ——, "Code-Review Policy," http://www.mozilla.org/hacking/reviewers.html, February 2015.

[4] MozillaWiki, "Code Review."

[5] 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. New York, NY, USA: ACM, 2014, pp. 192–201.

[6] 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, p. To appear.

[7] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *Proceedings of the 2005 International Workshop on Mining Software Repositories*, ser. MSR '05. New York, NY, USA: ACM, 2005, pp. 1–5.

[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. New York, NY, USA: ACM, 2011, pp. 4–14.

[9] 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. New York, NY, USA: ACM, 2011, pp. 491–500. [Online]. Available: http://doi.acm.org/10.1145/1985793.1985860

[10] S. Matsumoto, Y. Kamei, A. Monden, K.-i. Matsumoto, and M. Nakamura, "An analysis of developer metrics for fault prediction," in *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, ser. PROMISE '10. New York, NY, USA: ACM, 2010, pp. 18:1–18:9.

[11] MozillaWiki, "Modules," https://wiki.mozilla.org/Modules, February 2015.

[12] S. Kim, E. J. Whitehead, Jr., and Y. Zhang, "Classifying software changes: Clean or buggy?" *IEEE Trans. Softw. Eng.*, vol. 34, no. 2, pp. 181–196, Mar. 2008.

[13] O. Baysal, O. Kononenko, R. Holmes, and M. Godfrey, "The secret life of patches: A firefox case study," in *Proc. of the 19th Working Conference on Reverse Engineering*, 2012, pp. 447–455.

[14] O. Baysal, O. Kononenko, R. Holmes, and M. W. Godfrey, "The influence of non-technical factors on code review." in *WCRE*. IEEE, 2013, pp. 122–131.

[15] P. Weissgerber, D. Neu, and S. Diehl, "Small patches get in!" in *Proc. of the 2008 Int. Working Conf. on Mining Soft. Repos.*, 2008, pp. 67–76.

[16] R. E., *The cathedral and the bazaar. Musings on Linux and Open Source by an accidental revolutionary.* Cambridge: O'Reilly & Associates, 1999.

[17] S. Kim, T. Zimmermann, K. Pan, and E. J. J. Whitehead, "Automatic identification of bug-introducing changes," in *ASE '06: Proceedings of the 21st IEEE International Conference on Automated Software Engineering (ASE'06)*. IEEE Computer Society, 2006, pp. 81–90.

[18] Mozilla, "BMO/ElasticSearch," https://wiki.mozilla.org/BMO/ElasticSearch.

[19] J. Cohen, *Applied Multiple Regression - Correlation Analysis for the Behavioral Sciences*, 2003.

[20] M. Cataldo, A. Mockus, J. Roberts, and J. Herbsleb, "Software dependencies, work dependencies, and their impact on failures," *Software Engineering, IEEE Transactions on*, vol. 35, no. 6, pp. 864–878, Nov 2009.

[21] A. Mockus, "Organizational volatility and its effects on software defects," in *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*. New York, NY, USA: ACM, 2010, pp. 117–126.

[22] J. Fox, *Applied Regression Analysis and Generalized Linear Models*. SAGE Publications, 2008.

[23] ——, *Applied Regression Analysis, Linear Models, and Related Methods*. SAGE Publications, 1997.

[24] 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. Piscataway, NJ, USA: IEEE Press, 2013, pp. 712–721.

[25] O. Baysal and R. Holmes, "A Qualitative Study of Mozilla's Process Management Practices," David R. Cheriton School of Computer Science, University of Waterloo, Waterloo, Canada, Tech. Rep. CS-2012-10, June 2012. [Online]. Available: http://www.cs.uwaterloo.ca/research/tr/2012/CS-2012-10.pdf

[26] J. Eyolfson, L. Tan, 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. New York, NY, USA: ACM, 2011, pp. 153–162.

[27] J. Cohen, *Best Kept Secrets of Peer Code Review*. Smart Bear Inc., Austin, TX, USA, 2006.

[28] P. C. Rigby and D. M. German, "A preliminary examination of code review processes in open source projects," University of Victoria, Tech. Rep. DCS-305-IR, January 2006.

[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. New York, NY, USA: ACM, 2011, pp. 541–550.

[30] N. Nagappan, B. Murphy, and V. Basili, "The influence of organizational structure on software quality: An empirical case study," in *Proceedings of the 30th International Conference on Software Engineering*, ser. ICSE '08. New York, NY, USA: ACM, 2008, pp. 521–530.

[31] Y. Jiang, B. Adams, and D. M. German, "Will my patch make it? and how fast?: Case study on the linux kernel," in *Proceedings of the 10th Working Conference on Mining Software Repositories*, ser. MSR '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 101–110.

[32] C. F. Kemerer and M. C. Paulk, "The impact of design and code reviews on software quality: An empirical study based on psp data," *IEEE Trans. Softw. Eng.*, vol. 35, no. 4, pp. 534–550, Jul. 2009.

[33] 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 *Proceedings of the 2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*, ser. SCAM '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 104–113.

[34] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, "A large-scale empirical study of just-in-time quality assurance." *IEEE Trans. Software Eng.*, vol. 39, no. 6, pp. 757–773, 2013.

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

[36] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *Proceedings of the 27th International Conference on Software Engineering*, ser. ICSE '05. New York, NY, USA: ACM, 2005, pp. 284–292.

[37] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *Proceedings of the 30th International Conference on Software Engineering*, ser. ICSE '08. New York, NY, USA: ACM, 2008, pp. 181–190.

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

[39] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy, "Cross-project defect prediction: A large scale experiment on data vs. domain vs. process," in *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC/FSE '09. New York, NY, USA: ACM, 2009.

[40] T. Mende and R. Koschke, "Effort-aware defect prediction models," in *Proceedings of the 2010 14th European Conference on Software Maintenance and Reengineering*, ser. CSMR '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 107–116.

[41] 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.

[42] M. V. Mantyla and C. Lassenius, "What types of defects are really discovered in code reviews?" *IEEE Trans. Softw. Eng.*, vol. 35, no. 3, pp. 430–448, May 2009.

[43] L. Hatton, "Testing the value of checklists in code inspections." *IEEE Software*, vol. 25, no. 4, pp. 82–88, 2008.