Code Review Quality: How Developers See It

Oleksii Kononenko
School of Computer Science
University of Waterloo
Waterloo, ON, Canada
okononen@uwaterloo.ca

Olga Baysal
School of Computer Science
Carleton University
Ottawa, ON, Canada
olga.baysal@carleton.ca

Michael W. Godfrey
School of Computer Science
University of Waterloo
Waterloo, ON, Canada
migod@uwaterloo.ca

ABSTRACT

In a large, long-lived project, an effective code review process is key to ensuring the long-term quality of the code base. In this work, we study code review practices of a large, open source project, and we investigate how the developers themselves perceive code review quality. We present a qualitative study that summarizes the results from a survey of 88 Mozilla core developers. The results provide developer insights into how they define review quality, what factors contribute to how they evaluate submitted code, and what challenges they face when performing review tasks. We found that the review quality is primarily associated with the thoroughness of the feedback, the reviewer's familiarity with the code, and the perceived quality of the code itself. Also, we found that while different factors are perceived to contribute to the review quality, reviewers often find it difficult to keep their technical skills up-to-date, manage personal priorities, and mitigate context switching.

CCS Concepts

•**Software and its engineering** → **Maintaining software;** *Collaboration in software development;*

Keywords

Code review, review quality, survey, developer perception

1. INTRODUCTION

In a large, long-lived project, an effective code review process is key to ensuring the long-term quality of the code base. Code review is considered to be one of the most effective QA practices in software development. While it is relatively expensive in terms of time and effort, it delivers benefits of identifying defects in code modifications before they are committed into the project's code base [9]. Reviewers play a vital role in the code review process not only by shaping and evaluating individual contributions but also by ensuring the high quality of the project's master code repository.

Code review explicitly addresses the quality of contributions before they are integrated into project's code base. 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 [4, 25]. 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 our recent study [18], we explored the topic of code review quality by conducting a quantitative investigation of what factors may influence the quality of evaluating code contributions. The study was of quantitative nature as it employed data mining and analysis of project's repositories. While we found that both technical and personal attributes are associated with the review quality, many other factors such as organization, its culture and structure, development cycles, time pressures, etc., can potentially influence how reviewers assess code changes. Since these "hidden" factors are difficult to take into account in a quantitative analysis because such data is not available, easily accessible, or extractable from the available artifacts, we decided to employ qualitative research methods to fill the gap in the knowledge we had about the developer perception and attitude towards the code review quality.

Our qualitative study is organized around an exploratory survey that we design based on the state-of-the-art qualitative research [8, 12, 28] and our own observations of the Mozilla code review process and interactions with Mozilla developers during our previous research project [5]. We conducted an exploratory survey with 88 Mozilla core developers. Our qualitative analysis of the survey data aims at addressing the following research questions:

- RQ1: *How do Mozilla developers conduct code review?*

  Existing literature offers several case studies of how code review processes are employed by various software development projects and organizations [4, 6, 11, 21, 26, 27].

- RQ2: *What factors do developers consider to be influential to review time and decision?*

  Code review is a complex process that involves people, their skills and social dynamics, as well as development artifacts and environments; thus, it can be affected by both technical [15, 21, 25, 27] and non-technical factors [7, 12, 19, 29].

- RQ3: *What factors do developers use to assess code review quality?*

  While the quality assessment of code contributions is an active research area, the topic of code review quality remains largely unexplored. To better understand

ICSE '16, May 14-22, 2016, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/2884781.2884840

developer perception and attitudes towards the quality of the process that evaluates code changes, we further refine this research question into the following subquestions:

  - RQ3.1: *How do reviewers assess the quality of a patch?*
  - RQ3.2: *How do developers define a well-done code review?*
  - RQ3.3: *What factors are perceived to contribute to the review quality?*

- RQ4: *What challenges do developers face when performing review tasks?*

  We believe that it is important to understand what ongoing problems developers deal with to provide them with better tools to support their daily tasks and activities.

Our main findings reveal that the review quality is primarily associated with the thoroughness of the feedback, the reviewer's familiarity with the code, and the perceived quality of the code itself. As expected, we found that different factors including technical, personal and social signals, are perceived to contribute to the review quality. Also, we found that reviewers often find it difficult to keep their technical skills up-to-date, manage personal priorities, and mitigate context switching.

The paper makes the following contributions:

- A qualitative study with the professional OSS developers who participated in our survey on the topic of code review quality.
- A thorough survey analysis that offers insights into the developer perception of the code review quality and factors affecting it, as well as identifies of the main challenges developers face when conducting review tasks.
- A publicly available dataset of 88 anonymized survey responses.

The rest of the paper organized as follows. Section 2 summarizes prior related work. Section 3 describes our methodology including the survey design, participants, and data analysis. Section 4 presents the results of the qualitative study. Section 5 discusses implications of our work and suggests possible future research directions. Section 6 addresses limitations of our work. Finally, Section 7 concludes the paper.

2. RELATED WORK

A large body of work has attempted to assess modern code review as practised in the development of large software systems. Mockus et al. [21] were among the pioneer researchers who studied open source development. By analyzing the Mozilla and Apache projects, they identified the main characteristics of open source communities such as the dependency on the contributions from outside developers, and developers being free to choose tasks to work on. Rigby and German [26] presented an investigation of code review processes as practised within four open source projects: GCC, Linux, Mozilla, and Apache. They identified the existence of several review patterns as well as quantitatively studied the execution of the review process inside the Apache project. Later, Rigby and Storey [27] investigated the mechanisms employed by developers of five open source projects to identify code changes they are competent to review. They explored the way stakeholders interact with one another during the code review process. Their findings provided insights to developers about how to effectively manage large quantities of reviews. Additionally, their investigation reveals that the identification of defects is not the sole motivation for modern code review. Resolving non-technical issues, such as scope or process issues, was among other motivations of code review.

Baysal et al. [7] empirically studied the WebKit project and showed that review positivity, i.e., the proportion of accepted patches, is also affected by non-technical factors. The authors found that factors from both personal and organizational dimensions affect code review response time and the outcome of a review. Jiang et al. [15] studied, through an analysis of the Linux Kernel, the relation between patch characteristics and the probability of patch acceptance as well as the time taken for patches to be integrated into the code base. The results of their study indicated that developer experience and patch maturity impact the patch acceptance, while the number of suggested reviewers, affected subsystems, developer experience, and the time of submission affect the duration of a review. Bacchelli and Bird [4] examined the motivations, challenges, and outcomes of tool-based code reviews. They found that while the identification of defects is the main purpose of code review, other motivations such as team awareness and knowledge sharing exist as well.

Kemerer et al. [16] investigated the effect of review rate (i.e., lines of code reviewed per hour) on the effectiveness of defect removal and the quality of software products. Their study showed that allowing sufficient preparation time for reviews and inspections can improve their quality. Hatton [13] found that defect detection capabilities differ among code reviewers — the "worst" reviewer was ten times less effective than the best reviewer. Moreover, he found almost 50% improvement in defects detection between settings where the source code is inspected by two developers together (76% of faults found) and where the source code is inspected by two developers separately (53% of faults found).

Pull-based development is a new model for distributed software development; in this model proposed changes submitted as pull requests. In a quantitative study, Gousios et al. [11] explored what factors affect the decision to accept a pull request, as well as the time needed to make such a decision. They found that decision is dominated by the number of recent changes to the files being modified by the pull request, the number of such files, and the size of a project; while the time is mainly affected by the developer's track record and the test coverage of a project. Tsay et al. [29] analyzed the influence of social and technical factors on acceptance of pull requests. Their study showed that while many different factors affect pull request acceptance, the social distance and collaborator status are the most influential ones.

Although modern code review has received a significant attention recently, there is little empirical evidence on what factors contribute to code review *quality*. A recent qualitative study by Gousios et al. [12] explored the practices of pull-based development model. They conducted a two-round survey with 21 and 749 GitHub integrators on what factors they consider in their decision making process to accept or reject a pull request. They found that integrators are most concerned with quality and prioritization.

In our recent empirical study [18], we quantitatively investigated the relationships between the reviewers' code inspections and a set of factors, both personal and social in nature, that might affect the review quality. We found that 54% of the reviewed changes introduced bugs in the code; we also found that 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. While we identified which of these factors are more likely to contribute to the review quality, our approach was mainly quantitative; we were limited by the number of metrics whose values we could extract from the code review data stored in the issue tracking system. To improve the understanding of how developers perceive the quality of their reviews and the factors they consider to be important in preserving the quality of the code assessment, we extend our previous work here by presenting a qualitative study of developer perception and attitudes.

3. METHODOLOGY

We conducted an exploratory qualitative study that involved data collection through a survey with professional developers. This section describes the survey design, the participants, and the analysis of the responses in detail.

3.1 Survey Design

The survey consisted of three main parts: nine questions about the developer's demographic background and work practices, three Likert-scale questions related to different aspects of code review, and seven follow-on open-ended questions to allow developers to elaborate on issues raised by the multiple choice questions. Participants were asked to spend 5–10 minutes to complete the survey.

The main goal of conducting the survey was to solicit developer feedback on the perceived quality of code reviews and factors affecting review time, decision, and quality. We also wished to identify key problem areas within the existing review process.

3.2 Participants

We decided to continue our work within the Mozilla project developer community for several reasons: much of our previous work has studied this project and we have good intuition about the system and its development practices, we have made good contacts within the project who are supportive of our research goals, and because Mozilla is a well known, very large, and long-lived open source project.

To identify potential participants for our study, we looked at the 12 month history (from May 10, 2014 to May 10, 2015) of all contributions to the Mozilla project as they are recorded in Bugzilla issue tracking system. Because of the Bugzilla's limitations on the search results, we directly queried Mozilla's Elastic Search cluster that contains the up-to-date copy of Bugzilla data [23]. By processing the queried data, we extracted 3,142 unique email addresses (Bugzilla uses an email address as a unique user identifier). After that, we queried the cluster for each email address to get the information about developer's activity: number of contributions submitted for review and the number of patches that were reviewed by the developer during the studied period. Finally, we used Bugzilla's REST API to extract developers' real names.

We decided to limit our survey to experienced developers who were not new to the project. We computed an experience value as the sum of submitted and reviewed patches. We set a threshold for the experience value at 15 — meaning that anyone with a combined experience of at least 15 patches will pass the filter — which reduced the list of potential participants to 843 (27%) people. To filter out developers who were new to the Mozilla project — regardless of their experience level — we defined familiarity as having contributions (submitted and/or reviewed patches) at least 6 months prior to the beginning of the studied period. This filter further reduced the list of experienced developers to 403 (13%) people.

Once we selected developers whom we wanted to survey, we sent out 403 personalized emails. Each email contained the number of contributions submitted or reviewed during the 12 months period and an invitation to participate in the survey. The survey was open for 3 weeks (from May 29 to June 19, 2015) and received 88 responses (22% response rate).

The beginning of the survey consisted of background-related questions. By analyzing the responses, we found that we had successfully targeted highly experienced developers: about 48% of respondents said that they have more than 10 years of software development experience, while another 26% of them have between 7 and 10 years of experience. Most of the respondents have been performing code review for more than 3 years (67%).

3.3 Survey Data Analysis

We applied a grounded theory methodology to analyze the survey data; as we had no predefined groups or categories, we used an open coding approach. As we analyzed the quotes, themes and categories emerged and evolved during the open coding process [20].

Author Kononenko created all of the "cards", splitting 88 survey responses into 938 individual quotes; these generally corresponded to individual cohesive statements. In further analysis, authors Kononenko and Baysal acted as coders to group cards into themes, merging themes into categories. For each open-ended question, we proceeded with this analysis in three steps:

1. The two coders independently performed card sorts on the 20% of the cards extracted from the survey responses to identify initial card groups. The coders then met to compare and discuss their identified groups.
2. The two coders performed another independent round, sorting another 20% of the quotes into the groups that were agreed-upon in the previous step. We then calculated and report the coder reliability to ensure the integrity of the card sort. We selected two of the most popular reliability coefficients for nominal data: percent agreement and Cohen's Kappa. Coder reliability is a measure of agreement among multiple coders for how they apply codes to text data. To calculate agreement, we counted the number of cards for each emerged group for both coders and used ReCal2 [10] for calculations. The coders achieved a substantial degree of agreement; on average two coders agreed on the coding of the content in 96% of the time (the average percent agreement varies across the questions and is within the range of 94.2–97.2%; while the average Cohen's Kappa score is 0.68).
3. The rest of the card sort (for each open-ended question) — 60% of the quotes — was performed by both coders together.

4. RESULTS

During the open coding process, 30 main (including "irrelevant") categories emerged. Table 1 presents these categories in detail reporting the number of quotes, the number of respondents, the question numbers, the totals, and the average percent agreement for each question.

4.1 RQ1: How do Mozilla developers conduct code review?

First, we wanted to understand the current practices of performing code review tasks at Mozilla. We asked developers several multiple choice questions with an option of providing their own detailed response. The first pair of questions focused on the workload that developers face: the average number of patches they write and the average number of reviews they perform each week. While the answers to these two questions are skewed towards smaller workloads (fewer than 5 patches per week submitted or reviewed, 69% and 57% respectively), we received many more responses for the heavier review workload than for the patch workload. About 10% of the respondents reported that they review 11 to 20 patches each week, while another 4% said that they review more than 21 patches each week. The analysis of a contingency table for these two variables shows that developers with high workloads (i.e., over 10 patches/reviews per week) tend to concentrate their efforts on a single task type, i.e., either writing patches or reviewing them. The need for "dedicated" reviewers is pursued to bring their unique knowledge and expertise, e.g., overall architecture or domain knowledge, to the project to ensure the correctness and fit of code contributions. This finding mirrors Mozilla's notion of super reviewers — a small set of developers enlisted by Mozilla who provide an additional review for certain kinds of changes [24].

The remaining two questions focused on where developers perform code review (i.e., within what environment) and where they discuss patches under review. While all review-related information is stored in Bugzilla, there is no requirement in the Mozilla's code review policies on where a review should be performed. Surprisingly, although Mozilla provides their developers with a code review platform called MozReview [32], only 5% of the respondents said that they are using it. The majority of the respondents (80%) conduct their code review tasks inside Bugzilla itself, while another 8% copy a patch locally into their IDE. As for the locations of patch discussions, developers were allowed to select multiple of the proposed answers and/or their own answer. The two overwhelmingly popular answers were Bugzilla and IRC channel (99% and 78% respectively), while VoIP, email, and face-to-face discussions received a similar number of responses (around 22% each). While this wide adoption of IRC might be influenced by Mozilla itself, it also might be explained by the fact that IRC allows them to have real-time, less formal discussions with ability to bring in more people into a conversation as needed.

RQ1: While most of developers write patches as well as review them, a dedicated group of developers is responsible for reviewing code changes. The majority of reviewers conduct code review in Bugzilla despite having access to a custom built code review tool, and use various communication channels for discussing code modifications.

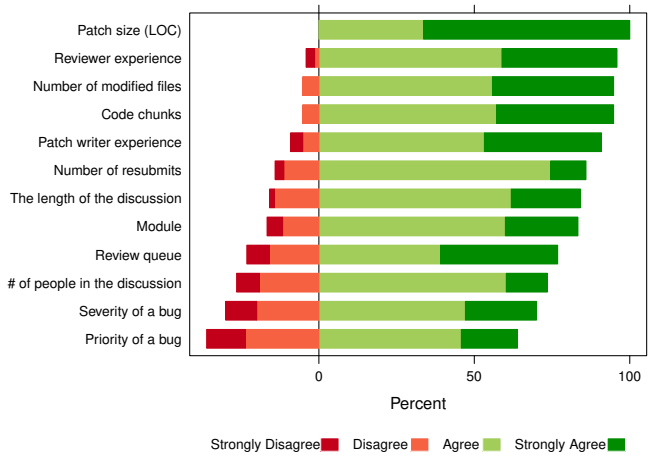

Figure 1: Factors influencing code review time.

4.2 RQ2: What factors do developers consider to be influential to review time and decision?

We asked developers about the factors that they believe are most likely to affect the length of time needed to review a patch, as well as the decision of the review (i.e., accept or reject). For each aspect (review time and decision), we solicit developers' opinions via a 5-point Likert-scale question and probe more in-depth information via an optional follow-on open-ended question. The proposed answers to Likert-scale questions were compiled from the factors that were previously reported in the literature [7, 15, 31] to have an impact on time and outcome. The open-ended questions provided developers an opportunity to specify any other factors not covered by the Likert-scale question.

1) Time. The analysis of the Likert-scale question (Figure 1) shows that size-related factors (patch size, the number of modified files, and the number of code chunks) are the ones the developers feel are most important (100%, 95%, and 95% of positive responses respectively). This finding is consistent with several previous quantitative studies that demonstrate the correlation between the size of the code change and the review time (i.e., smaller patches are more likely to receive faster responses) [7,15,31]. The second most positive group is experience — reviewer experience (96%) and patch writer experience (91%). Again, this also mirrors previous research that found that the increase in experience leads to faster reviews. While all other proposed factors received more than 50% of positive responses, the two factors with the biggest numbers of negative responses stand out: bug priority and severity received 36% and 30% of negative responses respectively. Such high values speak against the very idea of bug triage. It may be because Mozilla developers use the priority and severity fields inconsistently [14], or because these fields are not used as intended (for example, in our previous study, we found that over 96% of all patches in WebKit project are assigned the same priority value [7]).

The manual coding analysis of the open-ended question revealed several categories that developers believe have an impact on code review time. The biggest theme identified in the responses is *code quality*, which includes *code quality* and *change complexity* categories. As explained by R67, *"The amount of in-code comments describing what the patch*

Table 1: The list of categories that emerged during open coding.

Category	Q11		Q13		Q14		Q15		Q17		Q18	
	#Q	#R	#Q	#R	#Q	#R	#Q	#R	#Q	#R	#Q	#R
Code quality	49%	57%	24%	30%	31%	65%	9%	22%	8%	15%	1%	2%
Testing	13%	28%	6%	9%	12%	36%	7%	15%	8%	15%	–	–
Time constraints	1%	4%	–	–	–	–	8%	20%	14%	19%	17%	25%
Change scope/rationale	11%	22%	9%	12%	26%	58%	–	–	4%	8%	10%	15%
Understanding code change/base	–	–	6%	9%	–	–	21%	30%	20%	31%	31%	38%
Human factors	–	–	–	–	–	–	17%	28%	14%	23%	11%	16%
Tools	–	–	4%	5%	–	–	–	–	6%	12%	9%	9%
Communication	–	–	2%	2%	–	–	–	–	8%	12%	1%	2%
Change complexity	–	–	18%	23%	10%	31%	–	–	8%	15%	10%	15%
Relationship/trust	5%	9%	3%	5%	–	–	–	–	–	–	–	–
Usefulness	–	–	–	–	1%	3%	1%	3%	–	–	–	–
Workload	–	–	–	–	–	–	–	–	4%	8%	4%	5%
Submitter related	2%	6%	3%	5%	–	–	–	–	–	–	–	–
Architecture/design	–	–	–	–	5%	15%	6%	10%	–	–	–	–
Reviewer related	9%	13%	–	–	–	–	–	–	–	–	–	–
Discussion	1%	4%	–	–	–	–	–	–	–	–	–	–
Conformance to project goals	6%	7%	–	–	–	–	–	–	–	–	–	–
Bug type	–	–	9%	12%	–	–	–	–	–	–	–	–
Selecting correct reviewer	–	–	7%	12%	–	–	–	–	–	–	–	–
Performance	–	–	–	–	2%	8%	–	–	–	–	–	–
Integration into code base	–	–	–	–	4%	15%	–	–	–	–	–	–
Security	–	–	–	–	1%	3%	–	–	–	–	–	–
Memory management	–	–	–	–	1%	2%	–	–	–	–	–	–
Familiarity with the author	–	–	–	–	1%	5%	–	–	–	–	–	–
Thorough feedback	–	–	–	–	–	–	23%	38%	–	–	–	–
Catching bugs	–	–	–	–	–	–	4%	8%	–	–	–	–
Organizational factors	–	–	–	–	–	–	–	–	4%	4%	–	–
Documentation	–	–	–	–	–	–	–	–	2%	4%	–	–
Context switch	–	–	–	–	–	–	–	–	–	–	6%	10%
Irrelevant	3%	7%	9%	12%	5%	15%	4%	10%	–	–	–	–
Total:	141	54	67	43	290	86	219	86	50	26	81	55
Average percent agreement	**97.2%**		**96.6%**		**95.5%**		**94.2%**		**94.2%**		**95.0%**	

Notes: #Q: the number of quotes, #R: the number of respondents, Q11: factors affecting decision, Q13: factors affecting time, Q14: patch quality, Q15: characteristics of code review quality, Q17: other factors affecting review quality, Q18: challenges.

does. Readability/variable-naming affecting how hard it is to understand any particular hunk of the patch on its own." When reviewing patches, the developers stated that *"patches dependency"* (R39) and *"changes to the API surface between modules"* (R32) affect the review time.

Perhaps surprisingly, developers identified that the *bug type* category also plays a role during the review of a patch and affects its time. According to respondent R76, *"When the cause of the bug is obscure, it takes time to review"*, while R74 said *"nature of the bug — some bugs require time-consuming manual testing"*.

Another category that emerged from the responses is *patch scope and rationale*. Here, the scope also includes granularity: *"Whether the patch is broken up into self-contained pieces or whether it's one big patch touching lots of different areas — 5 individual patches are much faster to review in total than one big merged patch of those pieces"* (R19). Developers believe that the clarity of explanation of what is being changed and why affects the review time: *"clearly identified goal for the patch"* (R11) and *"what is patch trying to do, and should we even be doing that?"* (R55). *Understanding the code base* category goes along with the scope of a patch. Several developers stated that amount of knowledge that a reviewer has about the code being changed affects the review time.

Several of the emerged categories can be combined into a *social* theme. One of the categories here is *selecting the correct reviewer*. There are different characteristics that identify the suitability of a reviewer. For R87 it is *"the personality of a reviewer"*, while for R52 it is presence of *"personal backlog of work, and personal priorities"*. Moreover, sometimes the reviewers themselves question their suitability for reviewing a patch: *"am I the best person to be reviewing this patch?"* (R55). Developers also identified the importance of previous *relationship* with an author of a patch: *"if someone has a good track record I won't think about the code in quite as much detail compared to someone with a track record of breaking things often"* (R13). The other categories in this theme are about *submitter type* (e.g., newcomer or not) and the ease of *communication* between a patch writer and a reviewer.

2) Decision. Contrary to the answers to the Likert-scale question about the review time, we found no agreement between developers (i.e., strong prevalence of either positive or negative answers) for the majority of factors in the case of the review decision (Figure 2). Similarly to the previous question, both patch writer experience and reviewer experience are the factors with the most positive answers (86% and 84%). At the same time, the size-related factors (patch size, the number of modified files, and the number of code chunks) no longer have the overwhelming number of positive answers; instead, the respondents are more likely to disagree with the statement that these factors affect review decisions. Surprisingly, bug severity and priority are now the

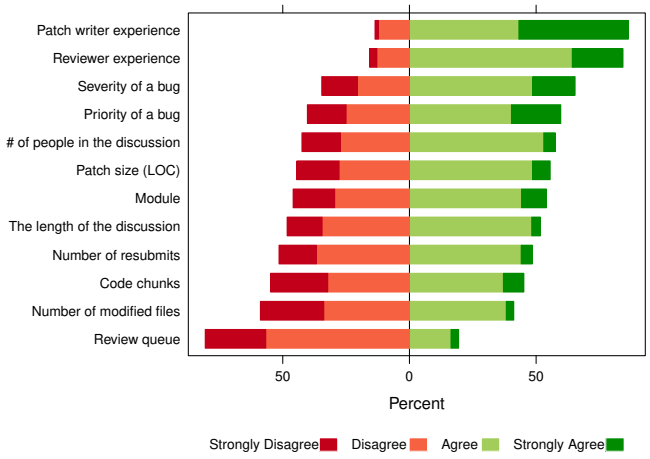

Figure 2: Factors influencing code review decision.

third and the fourth the most agreed factors. Another interesting finding is related to reviewer workload: about 81% of respondents disagree that workload affects the decision in any way. This demonstrates that developers think of reviewers as highly capable of carefully analyzing every patch regardless of the time pressure they might face. While such attitude describes the project's culture, this result contradicts our previous finding that suggest that reviewers with shorter review queues are more likely to reject a patch [7].

Several categories emerged during the analysis of the open-ended question related to review decision. The highest impact on the review decision is perceived to be *code quality* of a submitted patch. While developers associate different meanings with the term "code quality"; they can be grouped into several sub-categories. The first one is adherence to the code style (R57 – *"the quality of the code, and whether it adheres to accepted style and practices"*), as well as spelling (R38 – attention to *"details such as spelling, grammar and code formatting"*). Other two sub-categories are readability and simplicity of a patch (R34 – *"ease of understanding of code/changes, i.e., simplicity of code"*), and presence and quality of design or architectural changes. Finally, developers associate the code correctness and its maintainability with code quality.

The second biggest category identified from the answers is *testing*. When developers submit a patch they can include the results of running existing tests, as well as include the tests they wrote specifically for that patch. The two sub-categories that we identified reflect the option patch writers have. The first sub-category is focused on the presence of automated tests in a patch: *"... changes that are accompanied by tests are much more likely to be accepted"* (R20). Moreover, developers identified that the actual completeness of tests is also important: *"thoroughness of tests included in patch"* (R37). The other sub-category represents the presence of test results for a patch: *"including test results as a message on the bug tracker can either give the reviewer more confidence to accept the patch (if the tests pass) or likewise lead them to reject the patch (if the tests fail)"* (R38).

Change scope and rationale is believed to be an of influential factor for reviewers making their decisions. Reviewers first look for the actual appropriateness of change to be incorporated into the code base: *"Does the feature fit in with the product (for patches submitted out of the blue)"* (R29). As R10 explains: *"... not all fixes or improvements are a good idea to actually land, even if they're correct"*. Also, reviewers expect a clear explanation of the reasoning behind the proposed change, how it solves a problem, and why an author chose a particular way of doing it. According to R38, including such information *"can have a significant impact on some reviewers' confidence to accept the patch"*.

Similarly to the previous question, we have a theme of social categories. The reviews are done by humans, so the process is likely to be influenced by their personalities. Indeed, we identified *reviewer related* factors. Several developers report that with some reviewers it is more difficult to get a patch accepted than with others: *"a new reviewer might feel inclined to find a fault to prove that they done due duty in reviewing the patch"* (R38), and *"the perfectionist syndrome (Can you try ... ?)"* (R49). In addition to that, *"individual quirks/preferences of the reviewer"* (R20) play a role as well. *Relationship/trust* between reviewer and patch writer is found to play a critical role in decision making. Several respondents stated that interpersonal relationship is important for the review outcome. As explained by R36: *"If it's someone you trust you don't have to check things as rigorously"*. And finally, *contributor type* (i.e., whether he is new, mentored, or experienced contributor) can influence reviewers' decisions: *"if the patch writer is a new or first-time contributor, the reviewer may be inclined to encourage them by accepting their patch more readily (after identifying any obvious problems that need fixing)"* (R38).

> *RQ2: Developers believe that factors such as the experience of developers, the choice of a reviewer, size of a patch, its quality and rationale affect the time needed for review; while bug severity, code quality and its rationale, presence and quality of tests, and developer personality impact review decisions.*

4.3 RQ3: What factors do developers use to assess code review quality?

Quality is one of the key attributes of ensuring high standards of both code and project development. With this research question, we explore how developers perceive the quality of a patch and what characteristics they believe to be essential in contributing to a well-done code review. To answer this question, we analyzed two mandatory and one optional open-ended questions, as well as one multiple-choice question of the survey.

1) Perception of a patch quality.

One of the top attributes for developers when evaluating patch quality is *code quality*. Code quality has many interpretations. For some developers it is associated with coding style such as *"the names of things need to be descriptive"*, readability, compactness, maintainability (*"lack of redundant or duplicated code"*, *"strong and unverified coupling"*), *"consistent indentation and style"*, and *"elegance and lack of hacks"*. While for others code quality is about the presence of meaningful comments (*"comments should tell why not what"*), documentation and *"clear and helpful"* commit messages, *"I'm looking for a thoughtful summary that instructs me, reviewer, what is going on and what to expect"* (R28). Some developers find that *"adherence [of the code] to project module standards"* was equally important to ensure the changes are consistent and conformant to the Mozilla Coding Standards.

Change rationale is the second top property that reviewers look for. Patches are assessed for their correctness, *"does [it] actually implements what it intends to do?"* (R19), associated risk and possible alternative solutions, *"are there easier, less risky ways to achieve the same thing?"* (R35), functionality and errors (e.g., *"correct handling of exceptional cases"* (R33), *"are all cases handled?"* (R56)). Reviewers examine whether the patch author understands the source of the problem and the problem domain, without *"introducing any other bugs"* (R62) or ambiguity. Reviewers often think of their own solution to the fix before reviewing it and then compare it with the submitted patch. They also try to understand how much time the author spent on the patch and *"how well the solution has been thought through: does it needlessly reinvent the wheel, does it rewrite everything from scratch even though a spot fix would have been better, ... does it use "clever" tricks that others will struggle to understand"* (R64). In a nutshell, a high quality patch *"usually provides a robust solution for the problem"* (R42).

Change complexity is also perceived as an important property of the patch quality. Developers often look for simple solutions: *"simpler is better"* (R20), *"simplicity of code makes a big difference. Code that is complicated often is the result of not being able to distill the problem down to its core. Also, reducing the cognitive load required to understand the code means it's easier to maintain, and less likely to have bugs in it"* (R34). If a patch is trying to resolve more than one issue, it is expected that submitter split it into multiple patches: *"if the patch is addressing 3 or 4 different things it is lower quality than 3 or 4 separate patches for the individual issues"* (R13). Many developers agree that size of the change is correlated to the bug-proneness: *"small, focused changes are easier to assess than large ones. If bug rate is proportional to lines of code, quality is inversely proportional to patch size. So, small patches preferred"* (R28).

Testing is also a key indicator of quality for developers when they evaluate patches. Reviewers expect code changes to come with a corresponding test change. The lack of such tests is a good sign that *"test coverage is lacking and we're taking a risk accepting the patch"* (R28). The presence of tests in the patch also boosts developers confidence that the patch actually fixes the problem. Many developers run and test patches locally, or when testing is not practical, they perform manual testing as well. As a part of manual testing, developers often perform an operational proof such as code walks through: *"I walk through the changes, executing it as I imagine the machine would, with as much variety of inputs and states as I can imagine. I look for edge cases. I try to consider what is not in the patch (things that are being affected by the patch but are not directly changed by the patch)"* (R21).

Reviewers pay careful attention on how the patch fits into the existing code base. Integration into the code base can be examined by checking how the patch *"melds with the existing code or how it replaces the existing code"* (R23), *"how much change there is and how far spread the change is"* (R12), or whether *"the patch breaks web compatibility"* (R4). Submitters are often expected to be able to anticipate the upcoming surrounding changes and have an overall understanding of the impact of the change on other areas of the code. To support code maintainability, submitters are expected to conduct refactoring tasks if they see the need for it. Reviewers can request to perform necessary refactoring if they find that the patch is *"contributing to code rot"* (R38).

When reviewing patches, developers often examine whether *software architecture and design* meet expectations. For example, whether a code change *"meets other design considerations (e.g., PEP8 for Python code) (R67)"*. It is expected that submitted changes keep the architecture of the code base intact to facilitate code comprehension and maintenance: *"does it continue the architecture of the existing code or diverge in a way that makes future maintenance difficult?"* (R81), *"I look for architectural impact to see if it is making the code cleaner or messier"* (R87). If the code changes rely on APIs, reviewers check whether they are used appropriately: *"could the new APIs be misused?"* (R65).

Among other characteristics that developers consider when assessing changes are *memory management* such as *"no leaks, no unsafe memory usage"* (R4), *"no accesses to dead objects"* (R9), *security* such as security related checks and return types, *performance* that relates to *"the order of algorithms used"* (R38), *"the right trade-offs between simplicity of code and performance"* (R24), efficiency and speed.

Social factors such as *familiarity with the author* play an important role in evaluating patches. Previous relationships with the submitters, their experience and reputation within the project can determine the fate of their patches: *"I set a baseline based on any previous relationship with the submitter, and the area of code concerned. If I know the submitter I have both some idea of what to check and a better idea if they'll be around later to fix subsequent issues"* (R56), *"past experience of patch author is a big factor"* (R43).

2) Characteristics of a well-done code review.

This research question investigates developer perception of the key characteristics contributing to a well-done code review. Through an open-ended question, we asked developers' opinion on what a high quality review means to them.

The majority of the developers (38%) responded that *clear and thorough feedback* is the key attribute of a well-done review. Reviewers are expected to provide feedback that 1) is clear to understand; 2) is not only *"about code formatting and style"* (R6); 3) provides constructive advice, e.g., *"points out major correctness issues first, and points our minor issues that can be clearly fixed without another round of review"* (R24), *"highlighting potential problems ... and how to fix them"* (R42), *"saying 'this is the worst code I've ever seen' is not constructive"* (R81); 4) is done by the correct reviewer who *"has the domain knowledge to properly evaluate the change"* (R55); 5) is delivered via proper communication: *"good code reviews are dialogues between the reviewer and patch author"* (R50); and 6) provides mentoring and training for patch authors: *"providing detailed mentoring to help them improve faster"* (R56), *"to help the author of the patch become a better programmer in the long term"* (R35).

Developers expect reviewers to have *understanding of the code*, in particular to know *"the code that's being changed and what other pieces of code interact with it and what their assumptions are ("what else could break?") ..."* (R19), *"knowledge of the code is paramount because otherwise reviews are superficial"* (R30). Submitters want reviewers to know the outcome, the impact and *"the side effects of the modified code"* (R49), as well as to ensure that the logic of the patch makes sense. Reviewers are also expected to have an overall understanding of the project's code base: *"enough domain knowledge is always the first criteria for a well-done code*

review" (R61) and *"familiarity with utilities in other parts of the repository that could be re-used"* (R38).

We found that *human factors* play a crucial role for developers when receiving feedback. Developers associate good reviews with the reviewers who possess (1) personal attributes such as being *"supportive, yet strict"* (R9), *"patient and stable"* (R61), *"punctual and tactful"* (R28), *"helpful and encouraging, especially when rejecting a patch"* (R55), *"expressing appreciation for contributions"* (R38) especially if contributions come from the newcomers to the OSS community, and (2) inter-personal qualities such as being able to *"establish clear and open-minded communication"* (R73), *"trust the programmer to be competent enough to fix the problems"* (R64), provide positive and constructive feedback *"with the comments written in such a way that the patch author does not take them personally"* (R9) delivered in a *"constructive tone that respects/acknowledges the efforts of the patch writer"* (R21). From the developer perspective, code review relies on the participation of everyone on the project, and an ideal review process is described as the one that *"allows the author and the reviewer to work together to produce better code than either could on their own, maintain quality standards, and build familiarity with the code base"* (R56).

Code quality, once again, is found to be a vital part of the review process. The review quality is associated with patch writers taking into consideration coding style and formatting, preserving code maintainability, embracing *"current best practice within the project"* (R38). While reviewers are responsible for *"not allowing messy code in just because of time"* (R23), ensuring *"the patch achieves what it was intended to achieve"* (R31) and *"the code adheres to community standards"* (R82).

Quick *turnaround time* is also important for the responses as they report that both parties, reviewers and submitters, are expected be done in a timely manner, *"the value of the review feedback is in the proportion to the cost in terms of delays and time spend"* (R87). However, reviewers are to avoid shipping their feedback *"under stress or when there's a deadline"* (R7) as this introduces risks of missing problems. Some developers noted that Mozilla code review suffers from non-responsive reviewers due to overload or too few reviewers available. As a result, the speed of reviews might overweight the risks: *"depending on what module, a faster yet less thorough review is probably going to be OK, and worth the risk"* (R34).

Testing is seen as a feature that helps to accomplish the review process. During the review, developers are expected to apply the patch locally and test it to make sure it causes no regression. Thorough and careful testing of the patch ensures *"it is doing what is is supposed to and not introducing regressions"* (R42). Among other factors contributing to a well-done review are design and code pattern considerations, providing architectural recommendations (e.g., interaction wit other subsystems, use of correct APIs), and catching the bugs left in the patch.

3) Factors affecting code review quality.

Through a mandatory multiple choice question and an optional open-ended question, we asked participants to express their opinion on the factors they find to influence code review quality. The results of the relevant Likert-scale survey question are summarized in Figure 3. The vast majority of the developers agrees that factors such as reviewer experience and technical properties of the patch (patch size, code chunks, number of modified files) are strong indicators of code review quality. Most developers (76-85%) also consider that personal factors such as patch writer experience, reviewer workloads, developer participation in the discussion of code changes, module and number of resubmitted patches are more likely to affect the quality of reviews. While developers have mixed feelings about whether severity and priority of a bug, review response time, and the length of the discussion have an affect on code review and its quality.

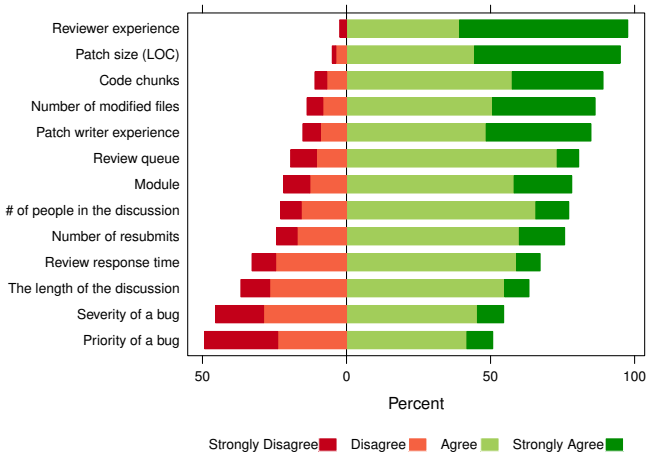

Figure 3: Factors influencing code review quality.

From the open-ended question, we found a number of additional factors that respondents think are influencing the review quality (but were not present in our multiple-choice question). As we have seen from previous findings, developers consider *understanding of the code base* as an important property that characterizes the review quality: *"domain expertise of both the author and reviewer"* (R35), and *"experience of the reviewer and familiarity with the code or domain are pretty important"* (R85).

Human factors such as reviewer mood, personality, experience, communication skills and style, style of making reviews, and productivity (stress level) are seen as ones of the highest determining factors in the quality of the review. *Time-related* factors such as the time of the day the reviewer gets to do a review, time pressure and deadlines, release schedules, release management, and priorities of other tasks are also among the factors that influence reviewer ability to deliver efficient reviews. Code quality and patch complexity, presence of tests, tool support, organizational overhead are mentioned as other potential factors affecting the review quality.

Developer perception of the factors affecting code review quality matches the insights we obtained in our qualitative analysis of the data from the project's repositories [18]. Reviewer experience and their work loads, number of previous patches, discussion around issues, as well as technical characteristics of a code change such as its size and the number of files it spreads across are found to be strong indicators of the code review quality.

RQ3: Developer perception of code review quality is shaped by their experience and defined as a function of clear and thorough feedback provided in a timely manner by a peer with a supreme knowledge of the code base, strong personal and inter-personal qualities.

4.4 RQ4: What challenges do developers face when performing review tasks?

This research question identifies key challenges developers face when conducting code review tasks. We identified two categories of challenges: *technical challenges* affect reviewers' ability to execute effective reviews, while *personal challenges* relate to their self-management and context switching. We also report the responses to an optional open-ended question about the desired tool support that could help developers with their code review activities.

1) Technical challenges

The biggest challenge for developers is *gaining familiarity with the code*. Since reviewers are often asked to review the code they do not own, understanding the unfamiliar code that the patch is modifying can be challenging, *"our module boundaries are broad, so patches often touch areas that I'm not up-to-date on"* (R37). Developers also find that decision-making of whether the change is good can be difficult, *"it's really important that I understand what the patch does"* (R62). Related to this, reviewers often have to assess whether they are capable of reviewing a particular patch or whether they should delegate it to a different reviewer, *"deciding whether I am the most appropriate reviewer or if my knowledge of the area of code is good enough to be an effective reviewer"* (R57). Reviewers are also expected to *fully understand the problem*, which can be a time-consuming process in particular if they review code in diverse areas of the code base. Reviewers have to not only understand the change but also understand its interactions with the existing code and being able to determine what code has to be co-changed, as well as *"spot now-redundant code"* (R74).

Another category of the technical challenges is related to *code complexity*. Reviewers are often required to evaluate large patches. The size of the patch is correlated with the quality of the reviews. Large patches are difficult to review because it can be difficult for developers to see the big picture: *"long patches are hard to review - attention wanes, quality of the review goes [down]"* (R21). R12 mentions that *"if large patches are broken up it can still be difficult to understand the bigger picture"*. The complexity of the pre-existing code can add up to this problem. Nevertheless, being able to see the big picture can be troublesome yet very critical for reviewers.

Finally, many reviewers complained about the current *tool support* available to perform review tasks. Some of them mentioned that reviewing in Bugzilla is difficult, while others refer to Bugzilla as *"a pretty good tool"* (R62). Since running automated tests is a part of the review, developers find applying the patch locally and testing it time-consuming. Reviewers mention that existing tools are good at visualizing line-by-line change ("diff" tools) but fall short in providing a summary of what a patch is changing (not on the file level).

2) Personal challenges

Reviewers often find themselves struggling with *time management skills* such as setting personal priorities and beating procrastination: *"convincing yourself that reviews should have higher priority than whatever other work you're doing"* (R19), *"how to get reviewing on a first priority and still getting your own things done"* (R40). All reviewers have other non-review tasks to conduct such as writing patches, participating in discussions, attending meetings (in person or remotely), engaging and recruiting other members to the community, or educating and training the new generation of hackers. Thus, balancing time to perform reviews, as well as all other daily activities can be a struggle. As R38 says *"I try to respond within 24 hours but sometimes a review for 15 patches can just show up out of the blue requiring a full day to review. That throws all other plans out of schedule"*.

On a personal level, reviewers often feel the pressure of *keeping up the personal technical skill level*: *"I need to constantly improve, so I can help others too"* (R52). While reviewers understand the importance of providing guidance and support to new contributors, they admit that this can be very time-consuming and carry risks of landing bug-prone patches to the code repository. As R54 explains: *"reviewing patches by new contributors where hand-holding is needed... it becomes tempting to simply land the patch to end the hassle for both the contributor and the reviewer. This frequently results in buggy code landing in the tree"*.

Several reviewers find it difficult to work on multiple tasks simultaneously. Working on multiple tasks such as performing reviews and fixing a bug is common for developers. *Context switching* from one problem space to another appears to be challenging. More importantly, when the patch undergoes several revisions, reviewers have to keep the context between revisions of the patch to make sure all their concerns with the proposed changes are addressed by the submitter. From the submitter's point of view, keeping track of the comments from the reviewer or other peers can be difficult. This becomes a challenge when a patch writer is working on a large bug or feature that involves a substantial discussion on the best way to resolve or implement it.

> *RQ4: The key challenges are twofold. Technical challenges are associated with gaining familiarity with the code, coping with the code complexity, and having suitable tool support. While personal challenges are related to time management, technical skills, and context switching.*

3) Tools

The majority of respondents perform code review tasks inside Bugzilla that provides a very basic and limited set of code review related features — it allows side-by-side viewing of the patch and code, as well as adding comments to the patch `diff`. The most commonly requested feature is a built-in `lint`-like tool that should provide automatic static analysis, automatic format and style checking, and automatic spell-checking. Moreover, many developers stated that such a tool should not only automatically check the code, but also automatically fix it (where it is possible). Such a feature would allow them to focus on a bigger picture rather spending time on small problems: *"I should be paying more attention to the architecture and the problem solving mechanics of the patch, rather than whether or not the braces are in the right position"* (R21).

Developers also expressed interests in having *better development environment* that offers the ability to easily get the patch from the issue tracking system into the local editor for analysis. Another feature is autolanding (i.e., incorporation into the code base) of patches once they are reviewed. Finally, the developers expressed a desire for direct access to the indexed source code from inside the issue tracker to better understand how the code that being changed is used, as well as for the ability to get the change history of that code.

Almost every code review involves "before-after" comparison of the code. Therefore, it is not surprising that developers want improved support for *diff tools*. The most desired features here are the ability to see the diff *"in the context of the entire file"* (R57) and compare the difference between the original code and code with multiple consecutive patches applied.

5. DISCUSSION

We now discuss several research directions that have emerged from this work and can help researchers and practitioners to plan their next research projects.

Reviewer recommender system. Mozilla developers need to control an overwhelming flow of information including bug reports, review requests, updates on their patches, etc. [5]. One way to help developers manage the increasing flow of information is to provide them with tools that can assist them with specific tasks. For example, for code review tasks, a reviewer recommender system could be able to help both reviewers and patch writers determine the right person to review a code change at hand considering reviewer code/module expertise, his or her current review loads and availability (schedule). For example, R8 asks for a tool to be able to "automatically identify potential reviewers based on similar types of code change (perhaps in other modules)". While there is a large body of research that addresses this problem of expertise recommendation and offers a variety of techniques [2, 3, 22, 30], most existing solutions are research tools that do not scale well or would otherwise be impractical to deploy within an industrial environment.

Next generation code review tool. Many reviewers expressed concerns with current support for tasks related to code review. Code review is an essential part of the development process at Mozilla; yet respondents complained about the lack of a good code review tool. We found that developers expressed interests in having an online code review tool that supports automatic static analysis, automatic format and style checker, as well as automatic spell checker. These features can help developers with their time-management by allowing them to focus on the code change and how they fit into the bigger picture rather than paying attention to the formatting and style nits. The next generation review tool should also support code indexing and navigation for reviewers to be able to better understand code modifications and their interactions with other areas of the code base. Another desired feature is related to developing better `diff` tools to enable the comparison of different versions of the code or tracking of individual code changes. Reviewers also commented on the importance of having the ability to compare code on "file-by-file" rather then "line-by-line" level and to determine the differences between multiple consecutive patches.

Reshaping OSS. While we only attracted and surveyed Mozilla developers, the results of our study can be applied to other OSS projects such as Linux, Apache, Red Hat, etc. Most recent studies were either conducted at Microsoft [4] or focused on the pull-based development model [12]. While pull-based development (e.g., via GitHub) is gaining popularity among distributed software development community, the need to continue studying and supporting the evolution of large long-lived OSS projects remains as important as ever. We noticed that some developers are interested in borrowing emerging technologies (e.g., GitHub) and bringing them to their own working environments. OSS projects are constantly reshaping themselves [1], and researchers can facilitate their growth by helping them address their practical needs and overcome the obstacles they face. Having said that, our study adds to the existing body of knowledge on code review.

6. THREATS AND LIMITATIONS

The first limitation lies in the validity of our findings from the qualitative study. While we carefully designed our survey questions to ensure their clarity, as with all exploratory studies, there is a chance we may have introduced the researcher bias when applying coding to the open ended questions. We tried to minimize this by coding the 20% of the card sorts extracted from each question independently, measuring the coder reliability on the next 20% and reporting these values in the paper (see Table 1).

As with any survey method, to control for sampling bias can be challenging. We targeted the core developers of the Mozilla community who actively participate in code review tasks either by evaluating patches of their peer developers or submitting their own code changes to reviewers for quality assessment.

We only survey developers from one large open source community, yet we targeted Mozilla's core developers who are full-time employees. While our findings might not generalize outside of Mozilla, we believe any medium and large open source project employ similar code review practices. Nevertheless, further research studies are needed to be able to provide greater insight into code review quality and develop an empirical body of knowledge on this topic. To encourage replication of our study, we documented our survey questions and card sort results in a technical report that is made available online [17]. We also made anonymized survey responses publicly available[1].

7. CONCLUSION

Code review is a vital element of any long-lived software development project. A high-quality execution of this process is essential to ensuring the ongoing quality of project's code base. This work explores the code review practices of a large, open source project and aims to understand the developers' perception of code review quality. To accomplish this, we surveyed 88 core contributors to the Mozilla project. The qualitative analysis of the survey responses provides insights into the factors that affect the time and decision of a review, the perceived review quality, and the challenges developers face when conducting code review tasks. Our findings suggest that the review quality is mainly associated with the thoroughness of the feedback, the reviewer's familiarity with the code, and the perceived quality of the code itself. We also found that developers often struggle with managing their personal priorities, maintaining their technical skill set, and mitigating context switching.

8. ACKNOWLEDGEMENTS

We thank all Mozilla developers participated in our study for their time, participation, and feedback.

[1] https://cs.uwaterloo.ca/~okononen/review_quality/

9. REFERENCES

[1] K. Amant and P. Zemliansky. *Internet-based Workplace Communications: Industry & Academic Applications*. Information Science Pub., 2005.

[2] J. Anvik. Automating bug report assignment. In *Proceedings of the 28th International Conference on Software Engineering*, pages 937–940, 2006.

[3] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *Proc. of the 28th Int. Conference on Software Engineering*, pages 361–370, 2006.

[4] A. Bacchelli and C. Bird. Expectations, outcomes, and challenges of modern code review. In *Proceedings of the International Conference on Software Engineering*, pages 712–721, 2013.

[5] O. Baysal, R. Holmes, and M. W. Godfrey. No issue left behind: Reducing information overload in issue tracking. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 666–677, 2014.

[6] 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*, pages 447–455, 2012.

[7] O. Baysal, O. Kononenko, R. Holmes, and M. W. Godfrey. The influence of non-technical factors on code review. In *Proc. of the Working Conference on Reverse Engineering*, pages 122–131, 2013.

[8] A. Begel and T. Zimmermann. Analyze this! 145 questions for data scientists in software engineering. In *Proceedings of the 36th International Conference on Software Engineering*, pages 12–23, 2014.

[9] M. E. Fagan. Design and code inspections to reduce errors in program development. *IBM Syst. J.*, 15(3):182–211, Sept. 1976.

[10] D. Freelon. ReCal2: Reliability for 2 coders. http://dfreelon.org/utils/recalfront/recal2/.

[11] G. Gousios, M. Pinzger, and A. v. Deursen. An exploratory study of the pull-based software development model. In *Proceedings of the 36th International Conference on Software Engineering*, pages 345–355, 2014.

[12] G. Gousios, A. Zaidman, M.-A. Storey, and A. van Deursen. Work practices and challenges in pull-based development: The integrator's perspective. In *Proceedings of the 37th International Conference on Software Engineering*, 2015.

[13] L. Hatton. Testing the value of checklists in code inspections. *IEEE Software*, 25(4):82–88, 2008.

[14] I. Herraiz, D. M. German, J. M. Gonzalez-Barahona, and G. Robles. Towards a simplification of the bug report form in eclipse. In *Proc. of the Int. Working Conf. on Mining Soft. Repos.*, pages 145–148, 2008.

[15] 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*, pages 101–110, 2013.

[16] 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.*, 35(4):534–550, July 2009.

[17] O. Kononenko and O. Baysal. A Qualitative Exploratory Study of How OSS Developers Define Code Review Quality. Technical Report CS-2015-14, University of Waterloo, Waterloo, Canada, August 2015.

[18] O. Kononenko, O. Baysal, L. Guerrouj, Y. Cao, and M. W. Godfrey. Investigating code review quality: Do people and participation matter? In *Proceedings of the International Conference on Software Maintenance and Evolution*, pages 111–120, 2015.

[19] J. Marlow, L. Dabbish, and J. Herbsleb. Impression formation in online peer production: Activity traces and personal profiles in github. In *Proceedings of the 2013 Conference on Computer Supported Cooperative Work*, pages 117–128, 2013.

[20] M. Miles and A. Huberman. *Qualitative Data Analysis: An Expanded Sourcebook*. SAGE Publications, 1994.

[21] A. Mockus, R. T. Fielding, and J. D. Herbsleb. Two case studies of open source software development: Apache and mozilla. *ACM Transactions on Software Engineering and Methodology*, 11(3):309–346, 2002.

[22] A. Mockus and J. Herbsleb. Expertise browser: a quantitative approach to identifying expertise. In *Proceedings of the 24rd International Conference on Software Engineering*, pages 503–512, May 2002.

[23] Mozilla. BMO/ElasticSearch. https://wiki.mozilla.org/BMO/ElasticSearch.

[24] Mozilla. Code-Review Policy. http://www.mozilla.org/hacking/reviewers.html, August 2015.

[25] P. C. Rigby and C. Bird. Convergent contemporary software peer review practices. In *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering*, pages 202–212, 2013.

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

[27] 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*, pages 541–550, 2011.

[28] L. Singer, F. F. Filho, and M.-A. Storey. Software engineering at the speed of light: How developers stay current using twitter. Technical Report DCS-350-IR, University of Victoria, Victoria, Canada.

[29] J. Tsay, L. Dabbish, and J. Herbsleb. Influence of social and technical factors for evaluating contribution in github. In *Proceedings of the 36th International Conference on Software Engineering*, pages 356–366, 2014.

[30] C. Weiss, R. Premraj, T. Zimmermann, and A. Zeller. How long will it take to fix this bug? In *Proceedings of the 4th International Workshop on Mining Software Repositories*, pages 1–1, May 2007.

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

[32] M. Wiki. MozReview. https://wiki.mozilla.org/Auto-tools/Projects/MozReview, August 2015.