What Makes a Code Change Easier To Review?

An Empirical Investigation on Code Change Reviewability

Achyudh Ram*
University of Waterloo
Waterloo, Canada
arkeshav@uwaterloo.ca

Anand Ashok Sawant*
Delft University of Technology
Delft, The Netherlands
a.a.sawant@tudelft.nl

Marco Castelluccio
Mozilla, London, United Kingdom
University of Napoli Federico II, Naples, Italy
mcastelluccio@mozilla.com

Alberto Bacchelli
University of Zurich
Zurich, Switzerland
bacchelli@ifi.uzh.ch

ABSTRACT

Peer code review is a practice widely adopted in software projects to improve the quality of code. In current code review practices, code changes are manually inspected by developers other than the author before these changes are integrated into a project or put into production. We conducted a study to obtain an empirical understanding of what makes a code change easier to review. To this end, we surveyed published academic literature and sources from gray literature (*e.g.*, blogs and white papers), we interviewed ten professional developers, and we designed and deployed a reviewability evaluation tool that professional developers used to rate the reviewability of 98 changes. We find that reviewability is defined through several factors, such as the change description, size, and coherent commit history. We provide recommendations for practitioners and researchers. Preprint [https://pure.tudelft.nl/portal/files/45941832/reviewability.pdf]. Data and Materials [https://doi.org/10.5281/zenodo.1323659].

CCS CONCEPTS

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

KEYWORDS

Code quality, code review, pull request

1 INTRODUCTION

Code review is a widespread practice aimed at reducing software defects and improving software maintainability [33]. The form of code review that is most widely adopted in both industrial [68, 69] and open-source software [67] (OSS) settings has been called *Modern Code Review* [39] (MCR) and is: (1) tool-based, (2) asynchronous, (3) informal, and (4) focused on reviewing new code changes rather than the whole, existing code base. The MCR process is implemented via tools such as Gerrit [19] and GitHub's pull-requests [51].

Ensuring an effective and efficient code review process is an open research area. Existing research has found that code change authors receive delayed or no feedback at all for their contributions [35, 52], while reviewers have problems staying on track with their increasing workload [53]. In such a context, it is in the interest of both authors and reviewers that the proposed code changes are easy to review. Nevertheless little evidence is available on what makes a code change easier to review, although this knowledge would provide valuable insights for both practitioners and researchers. Change authors can use it to help speed up the integration of their contributions and project reviewers can impose sound contribution guidelines. Researchers can focus their attention on how to support authors in automatically preparing an easy to review contribution.

We present an in-depth study, conducted to obtain an understanding of the factors that make a code change easier to review. Existing work on code review (*e.g.*, [38, 56, 62, 74]) investigated the factors of code changes associated with a higher acceptance or a lower time to review, but these factors are often dominated by aspects on which an author as little to no control, such as the identity of the author [62]. Instead, we expect reviewability to regard those factors related to the understandability and structure of the change rather than authorship and semantic aspects. Nevertheless, we do not make preset hypotheses on what reviewability is.

We conduct a study in three phases: In the first phase, we perform a multivocal literature review [47] to gain a sound foundation about what aspects of a code change reviewers focus on. From the analysis of 22 white literature sources published at international venues and 21 gray literature sources, ten distinct themes emerged that related to code change quality. In the second phase, to empirically determine which themes play a role in the reviewability of a change, we conduct task-based interviews with ten developers who work on either industry or open source projects. Our interviewees lead us to identify three primary factors that are critical to a change's reviewability. In the third phase, to triangulate our findings, we deploy a tool that allows reviewers to rate the reviewability of a change they have just reviewed. We collect real world data from 35 developers, and we observe that factors identified in the second phase of our study are confirmed. This allows us to put forth an empirically verified definition for code change reviewability.

*A. Ram and A. A. Sawant are both first authors, and contributed equally to the work.

ESEC/FSE '18, November 4–9, 2018, Lake Buena Vista, FL, USA

Figure 1: A section of garbled commit history

2 BACKGROUND

Peer code review is an accepted practice to reduce the number of software defects introduced during the development process [33]. In 1976, Fagan formalized a process where line-by-line code inspections [46] take place in a large group. However, Fagan inspections suffered from being synchronous, thus not having a place in the modern world where developers work in a more agile and distributed environment. This limitation has led to the widespread adoption of a more lightweight, informal, and asynchronous code review process knowns as *modern code review* (MCR) [33, 37, 57].

MCR is an iterative process, where each code change generates a feedback loop between the author and the reviewers. When an author submits a change, the reviewers can decide whether to directly accept/reject it or request further changes (*e.g.*, to make the change conform to a project's standards). This feedback loop is repeated until either the reviewers are satisfied or they deem the change unworthy of integration in the project. Most modern collaborative code review tools such as Gerrit and GitHub pull requests adhere to this model.

A considerable amount of literature investigated the factors that affect the review outcome of code changes (*e.g.*, merge decisions regarding pull requests [51]) and the variables that may delay this outcome. For example, smaller patches were found to be more likely accepted when sent by experienced developers [36, 58, 62, 76, 78, 81]. Jiang *et al.* found that contributors can shorten the time taken for reviewing their patches by participating actively in the development community and limiting the number of subsystems their patch alters [56]. Gousios *et al.* noted that the track record of the contributor and the number of lines in the pull request are important factors leading to the acceptance of a submitted change [51].

In this study, we are interested in examining the factors that lead to better reviewability of code changes. As a motivating example (Figure 1), we consider the (shortened) commit history from a pull request contributed to Mockito.[1] The contributor tried to address three independent issues in the same pull request, proposing a commit history that included several intermediate, temporary commits that did not convey meaningful information about the change. Both these aspects hindered the code review process and forced the reviewers to request multiple iterations over the patch. By commenting explicitly, the reviewers called out the contributor regarding the commit history and the presence of composite changes: "rework your commit history in order to see what your changes actually involves and split multiple fixes in separate PRs."

[1] https://github.com/mockito/mockito/pull/49/commits

While these aspects had an impact on the code review of the submitted change, we see that the outcome of the change is not adversely affected (this change was eventually accepted), as opposed to the effort required by the reviewers. This example illustrates as to why factors that lead to code changes that are easier to review may be only partially related to their acceptance and time to review.

3 RESEARCH METHODOLOGY

To obtain a deep understanding of the factors that make a code change easier to review, we conducted *mixed methods research* [42] to address the following research questions:

RQ_1: What makes a good code change, according to white and gray literature? We scope the initial focus of our investigation by surveying the existing literature regarding the characteristics associated with the quality of proposed code changes. We aim to collect a set of themes that can guide our subsequent empirical investigation on the factors that define code change reviewability. To obtain a comprehensive set of themes as an answer to this question, we not only focus on academic peer-reviewed literature (*i.e.*, *white* literature), but we also consider sources from the *gray* literature, such as blogs, white papers, and contribution guidelines (this approach is formally known as *multivocal literature review* [47]). In fact, for our practitioner-oriented field "synthesizing and combining both the state-of-the-art and -practice is very important" [47]; moreover, although there is a considerable amount of recent research on code review, the specific topic of quality of code changes has more diverse documents and scarcer academic investigations, which makes it a recommended candidate for multivocal synthesis [79].

RQ_2: What do reviewers associate with code change reviewability? Once we collect the candidate themes, we turn to practitioners to validate the themes' relevance to code change reviewability, to elicit missing, diverse factors, and to empirically determine the criticality of each factor. We consider two angles: (**RQ2.a**) we investigate general reviewers' perceptions of code changes that may have reviewability issues (according to the aforementioned themes) in a system they are not familiar with and (**RQ2.b**) we investigate specific reviewers' feedback on code changes that they just reviewed for the real-world systems they work on. With this approach, we aim at triangulating and validating the findings from different sources to obtain a more comprehensive view on code change reviewability.

In this study, we do not restrict ourselves to any single platform or programming language. Review of code changes is done universally, and we try to capture the perspective from both industrial and OSS settings to gain a thorough understanding as to what factors define code change reviewability. We focus primarily on the review process of a pull-request (PR) submitted on GitHub or of a code change submitted to code review systems, such as MozReview [24], Bugzilla [14], and Reviewboard [26]. Overall, our data sources include: (1) academic peer-reviewed literature, (2) industry white papers and guidelines, (3) task-based guided interviews with developers, and (4) reviewers' feedback on code changes submitted on the aforementioned code review systems.

3.1 White and gray literature survey (RQ1)

To form a basis for the factors that affect the reviewability of a patch, we conduct a multivocal literature review.

Identifying relevant white literature. We perform a keyword-based search on Google Scholar, IEEE Explore, and ACM Digital Library to create an initial selection of articles. As keywords we use: 'code review', 'code review practices', 'pull request acceptance', 'evaluating contributions on GitHub', 'pull-based development'. We initially restrict our search to work on the review of pull requests or code changes. We then expand the search terms to include papers on program comprehension related to the understandability of a code change, as it is reasonable to assume that reviewability is linked with this specific understanding. We also search for studies that focus on characterizing the code review process and the tools used in code review. All the resulting studies primarily evaluate code changes based on the merge decision or the time taken for this decision. This step resulted in an initial set of 12 papers.

We then perform backward snowballing [80] inspecting papers referred to in this initial selection of articles. At the same time, we surveyed the proceedings of all the relevant, top-level conference venues in Software Engineering such as ICSE [21], MSR [25], ICSME [22], ESEC/FSE [18] and journals such as IEEE TSE [31] and ACM TOSEM [29]. This step resulted in a further set of 28 papers.

We then filter the pool of papers that we have identified. Our primary filtration criterion is the relevance to our end-goal, which was assessed by reading the study setting, the methodology, and the conclusions made by the authors. When the relevance was not easy to ascertain from just this information, we add it to a list of selected articles that require further examination. This list was then discussed among all the authors to determine its relevance. This step resulted in a list consisting of 22 papers.

Identifying relevant gray literature We conduct the gray literature survey according to the guidelines proposed by Garousi *et al.* [48]. We search for documents from gray literature by using Google and filtering on a year by year basis. We use keywords such as 'code submission guidelines', 'code review practices', 'pull request guidelines', 'GitHub CONTRIBUTING.md', and 'pull request review preparation' to obtain a set of search results with content that might be relevant. We restrict ourselves to results only from the first page of Google. We take great care to ensure that these sources originate from reputable companies, organizations, and authors. We also take steps from preventing our personal search bias from playing a role in the search results, hence we search on Google using Google Chrome's incognito mode.

We restrict ourselves to white papers, reports, contribution guidelines, and developer blogs. We expand this initial selection of 55 results by performing backward snowballing [80] (*i.e.*, we follow the web links these sources contain that might refer to other documents on code review). We try to ensure that each of these results is relevant to our study as they must talk about factors that affect the review process. This step results in a selection of 21 sources.

Data extraction from the identified literature. For both the white and gray literature, we summarize each source and create an enumeration of the mentioned aspects that can impact the code review process and the outcome of a submitted code change. Subsequently, we conduct iterative *content analysis sessions* [61] to group summaries and aspects into the higher level *themes* that are related to the quality and reviewability of a code change. Iteratively, for each recommendation or outcome from a literature source, we verify whether we have previously identified a theme of this nature to which this document can be assigned or whether a new theme needs to be created or improved. Each document can fall into more than one theme. This iterative process resulted in ten themes related to the quality of code changes and, possibly, reviewability.

Table 1: Interview participants overview

ID	Domain	Company/project	Experience
P1	Industry	Microsoft	4 years
P2	Industry	HP	18 years
P3	Industry	SET GmbH	8 years
P4	Industry	SET GmbH	14 years
P5	Industry	Milvum B.V. NL	5 years
P6	Industry	Google	10 years
P7	Industry	Google	5 years
P8	Industry	Google	10 years
P9	OSS	Spring Framework	9 years
P10	OSS	Spring Framework	12 years

3.2 Task-based interviews (RQ2.a)

The goal of interviewing reviewers is three-fold: (1) to investigate whether and how the themes identified in the literature survey phase are factors of reviewability, (2) to determine any additional, diverse factor that might affect reviewability, and (3) to gain an understanding as to what reviewers consider as the key factors.

We strived to interview senior professional developers with a multi-year experience in both programming and code review, working on projects that routinely perform code review. To this aim, we contacted developers who work for large companies in three different Western countries (*i.e.*, the United States of America, The Netherlands, and Italy) and those that actively work on well-known OSS projects. We contacted industrial developers that were in our personal and professional network and those that work for large, established companies. In the case of open source developers, we mailed internal developers of large project asking for an interview. Overall this resulted in ten interview participants (identified as P1-10), whose background is summarized in Table 1.

The interview starts with the interviewee being asked to review a real-world code change, this acts as an incentive to help the interviewee provide open, yet specific answers on what can make the code change more or less straightforward to review. This task also has the added advantage of familiarizing the interviewee with the motivation behind this study. We select a set of three pull-requests from the Mockito project [23]. The actual reviewers have rejected two out of these changes due to issues related to themes emerged in the literature survey; one change, instead, has been accepted by the project without any discussion and we use it as a control. We ask the interviewees to be as detailed as possible about the issues they identify and to what extent they perceive it as affecting the ease with which the change can be reviewed. Before every interview, we randomize the code changes to mitigate any ordering effect.

After these tasks, we ask the interviewees a series of questions about their opinion on factors of a code change that make the review task easier or harder, by following a semi-structured interview format [75]. We ask the interviewees questions about specific factors and ask them to recollect problems to review a code change that they have previously encountered. After each interview, we transcribe and summarize its content[2]. Based on new factors uncovered or new discussion points, we iterate over the interview guideline. When, after the first five interviews, we reach a *saturation* point [49] (interviewees provide insights very similar to the earlier ones), we use a new set of code changes as a review task to increase the diversity in the responses. Overall, we used two different set of patches.

3.3 Developers' feedback on changes (RQ2.b)

After having collected information about reviewers' perceptions of factors that affect the ease with which they can review a code change, using pull-requests from systems they are not familiar with and asking about the experiences concerning reviewability issues they recall, we collect feedback about more specific cases.

To do so, we designed a tool that automatically asks for feedback from reviewers concerning a code change that they just reviewed. To design a usable tool, we employed the Rapid Iterative Testing and Evaluation (RITE) [63] method to evaluate the user experience and to uncover any usability issues associated with the initial setup and the feedback submission process. We choose five software developers who work in a commercial setting to use and test our tool. As per the norms of RITE, we declare the design to be successful after three consecutive iterations of the evaluation occur without the detection of any errors or failures. We follow the same issue categorization and resolution as Medlock *et al.* [63]. The result of this evaluation is the interface seen in Figure 2.

We create different integrations for this tool targeted at four different code review platforms. The GitHub integration consists of a GitHub app that integrates with the pull request mechanism in GitHub and posts a comment and link to a page where the reviewer can rate the PR, indicate how long it took to review, mention what was right about its reviewability and what should be improved. The integrations for Bugzilla [14], MozReview [24], and ReviewBoard [26] rely on a Mozilla Firefox plugin that allows reviewers to provide feedback on a code change submitted to these platforms by dynamically changing the pages.

Once our tool had passed the RITE evaluation, we conducted a pilot with the students and teaching assistants of the Software Engineering course at the Delft University of Technology. This pilot allowed us to find bugs in the prototype and fix them before deploying it to real developers.

We spread this tool out to developers by advertising it on social media platforms such as Twitter and Reddit. We also, contacted developers of some open source projects to ask them if they would like to try it out. Finally, Mozilla agreed to use it internally to rate the code changes reviewed for their open source projects. To incentivize the use of our feedback tool, we provided a dashboard to the contributors and project maintainers to view these responses as a report with graphical visualizations. Overall, we collect feedback on the reviewability of 98 real-world code changes[3].

[2]Interview guidelines and transcripts for which we got permission are available in the accompanying material [66]

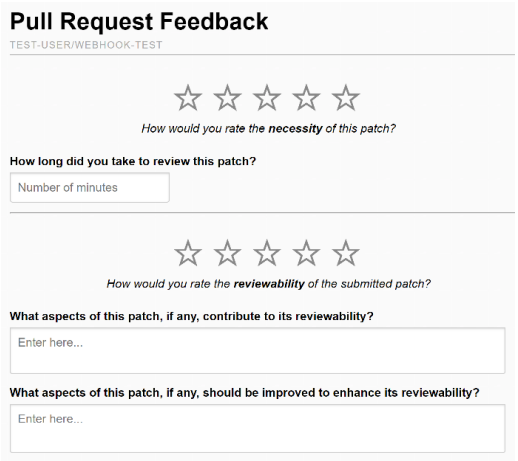

Figure 2: Developer feedback form (with trimmed text areas)

Table 2: Themes emerged from white and gray literature

Theme	White literature	Gray literature
Change description	[82], [74], [38]	[40], [28], [27], [10], [11], [13], [6], [12], [5], [3], [4], [2]
Change scope	[56], [58], [38], [72]	[27], [3]
Code churn	[76], [81], [78], [36], [56], [62], [70], [58], [38]	[40], [5], [3], [1]
Code quality	[53], [81], [71], [70], [43], [58], [52]	[28], [10], [11], [41], [6], [9], [4]
Code style	[54], [62], [43], [52]	[32], [6], [5], [4], [1]
Commit history	[53], [81], [72]	[27], [11], [32], [8], [4], [2], [1]
Composite changes	[74], [58], [52]	[28], [13], [41], [17], [5]
Nature of the change	[62], [58], [65]	
Subsystem hotness	[51], [53], [81], [56]	
Test inclusion	[53], [76], [81], [43], [77], [52]	[10], [11], [32], [6], [12], [8], [9], [7], [3], [4], [2]

4 RQ1 – A GOOD CODE CHANGE ACCORDING TO LITERATURE

We describe the themes (ordered alphabetically) that constitute a good code change according to the white and gray literature. Table 2 gives a high-level overview.

Change description. The quality of the change description has been shown to be associated with how well reviewers can understand a change [74]. Bosu *et al.* put forth a set of guidelines that change descriptions should follow [38]. Zhang *et al.* also find that,

[3]Data for these changes is available in the accompanying material [66]

on GitHub, a good change description should mention the reviewer who is most appropriate for that change[82].

These characteristics are echoed in developer guidelines, many of which require the authors to submit code changes with adequate documentation about the nature and impact of the change, as well as (for programming library projects) usage examples for the change [10, 11, 28]. GitHub's guidelines further recommend that a change should mention an appropriate reviewer [20]. Additionally, gray literature indicates that the description must link the addressed issue [2, 5, 6, 10–12, 27, 28].

Change scope. Jiang *et al.* find that the time to review a change in Linux increases with the number of modified subsystems and packages [56]. Kononenko *et al.* made similar observations at Mozilla [58] and Bosu *et al.* at Microsoft [38], while Soares *et al.* found no corresponding evidence among GitHub projects [72]. Marlow *et al.* note that there is less uncertainty when merging changes that are small in scope, as their impact is easier to understand [62].

Code churn. Code churn is the most frequent theme mentioned in the surveyed white literature. Kononenko *et al.* reports that developers indicate that size-related factors (*e.g.*, change size and number of modified files) considerably influence the reviewing time [58]; they also find that smaller changes undergo fewer rounds of revisions. Bosu *et al.* find that review effectiveness decreases with the size of the change [38]. They recommend smaller and incremental changes. A gray literature source, based on peer review at Cisco Systems, suggests the number of lines modified in a change to be less than 200 and to not exceed 400 [40].

Weißgerber *et al.* similarly show that small changes (at most four lines) have a higher chance of acceptance [78]. This point is also reflected in the work of Marlow *et al.* [62], who find that there is less uncertainty when deciding whether to merge small changes, and Yu *et al.* [81], who find that the size is one of the determinants of change evaluation latency on GitHub. However, Jiang *et al.*'s case study on the Linux Kernel found no impact for factors such as change size, spread, and fragmentation [56], thus also contradicting the Linux Foundation's guidelines [41].

Code quality. Gousios *et al.* [52, 53] find that factors such as code documentation, adherence to project conventions, style conformance, and code quality play a significant role in the acceptance of code changes. Kononenko *et al.* similarly find that the quality of code documentation, readability of the code, and adherence to variable-naming conventions negatively affect the understanding of a code change [58]. Silva *et al.* specifically study the topics discussed in code reviews and find that code design and adherence to project conventions are most commonly mentioned [71]. Due to the broad definition of quality, overlap exists with other themes such as code style, test inclusion, code churn and composite changes.

Code style. Marlow *et al.* find that changes that suffer from stylistic issues and adherence to poor coding practices require additional reviewer effort and often end up being rejected [62]. Gousios *et al.* go further to show that, apart from code quality, style conformance is the only factor that both the reviewers and authors are concerned with while working in the pull-based development model [52]. Hellendoorn *et al.* observed that changes that match the project's coding style are more likely to sail through a code review [54]. Gray literature sources, such as open source project contribution guidelines, typically specify their code style conventions on topics like variable naming and placement of new methods, which they expect to be followed by any submitted change [1, 3, 27].

Commit history. Yu *et al.* find that a higher number of commits for a code change lead to a more extended review period in GitHub pull requests [81]. Soares *et al.* show that a larger number of commits in a code change decreases its chances of acceptance [72]. Gousios *et al.* find that the format of the commit (*e.g.*, the message that summarizes the change) is a component of code quality and plays a role in the review process [53].

Several blogs and contribution guidelines echo these findings [1, 2, 4, 8, 32]. Some projects (*e.g.*, Spring Framework and Mockito) require to squash commits on the same logical change [11, 27].

Composite changes. Tao *et al.* find that addressing more than one issue in a code change (making it a composite or tangled change) increases the difficulty of understanding the change [74]. Kononenko *et al.* find that one of the factors that affects the time and outcome of a review is whether a code change is properly separated into self-contained pieces [58]. Gousios *et al.* recommend practitioners to keep their changes small and isolated [52].

The Linux Kernel requires code changes to be split into logically independent changes, each of which yields a properly functioning kernel [41]. Other open-source projects also recommend and require this practice [13, 17, 28].

Nature of the change. A code change can have several goals, such as introducing new functionality, fixing a bug, or maintaining existing code. The type of this change affects how reviewers perceive the change. Padhye *et al.* find that project maintainers readily accept code changes that fix a bug or update the documentation, rather than approving a new functionality into the project [65]. Vice versa, Marlow *et al.* find that the review period is shorter when the change addresses an urgent issue or fixes a known bug, as opposed to introducing new features that might conflict with existing functionalities [62]. Changes to interfaces between different subsystems furthermore increase the review time [58].

Subsystem hotness. Gousios *et al.* find that code changes in actively developed parts of OSS systems have a higher chance of being accepted [51, 53]. Yu *et al.* also find that "hotness" of the part of the system to which the contribution is being made reduces the review time [81]. However, Jiang *et al.* find a different result on the Linux kernel: there, code changes to more actively developed subsystems have a lower rate of acceptance [56]. They hypothesize that this could be due to multiple authors doing duplicate work by trying to address the same issue.

Test inclusion. In Gousios *et al.*'s findings, reviewers tend to consider changes with tests included as being technically sound, thus making the code review easier [53]. Others also find that the inclusion of test cases is considered an evaluation of a code change's technical proficiency and determines the amount of time needed to review the change [76, 81]. Tsay *et al.* observe that project core members may ask authors to provide specific test cases to motivate the need for the code change being submitted [77].

Some open-source projects require changes to be covered by automated tests [10, 11, 32]. Most of the gray literature sources also recommend that any change is accompanied by an adequate number of test cases [2–4, 6–9, 12].

The analysis of the white and the gray literature let emerge ten themes associated with the outcome and time to review a code change. For some themes, there is an indication that they may be related to code change reviewability.

5 RQ2 – DEVELOPERS' PERSPECTIVE ON CODE CHANGE REVIEWABILITY

We present the developers' perception on reviewability and the reviewers' feedback on specific code changes.

5.1 RQ2.a – Task-based interview results

Out of the first set of code changes shown to interviewees, one change had no issues with its reviewability. This code change has a succinct description, short commit history with clear commit messages, and a code coverage report that shows no drop in the coverage percentage. The second change attempts to fix a non-existent issue with no clear description as to why the change is needed and has no tests included. The third change tries to fix multiple issues and has several commits with garbled commit messages.

In the second set of code changes, the first change attempts to fix a bug without proper tests to verify whether the fix works, however, the build fails due to 'checkstyle' issues. The second change proposes to modify the inner workings of the project without proper motivations. Furthermore, the code change complicates the code base without strong advantages and the patch has a garbled commit history. The third patch aims at updating the code base to the latest version of a dependency. The accompanying change description motivated the change and guided its understanding.

5.1.1 Reviewability aspects in focus

We asked our interviewees as to what aspects of a change help them in understanding it. As they started to inspect the changes, we asked to explain what they were looking at and to also consider the meta-aspects of the change in addition to the technical aspects.

Change description. The change description is the most frequently brought up aspect that developers feel affect the reviewability. P2 immediately pointed out, for one of the changes, that the description did not mention where the code comes from and why, *i.e.*, it did not explain the change's context. P6 mentioned the same: "the first thing I want to see in a pull request is why you want [it]."

Nine out of the ten interviewees stressed that mentioning the motivation behind a specific change along with what is being changed is essential in aiding the code review process. P4 put this as: "[the patch description] is especially important for reviewing, as one has to understand why [and] not only what changes have been made but why they have been made." P9 was the only one in disagreement; in his opinion, "the code cannot lie and users do lie." Fundamentally, he does not trust what an author has mentioned in the change description and relies on the code itself.

Code churn. The consensus among our interviewees was that smaller the change the more reviewable it is.

P1 started his review by saying: "let me just take a look at the size of the change over here." P4 found one of the changes easier to review "because it is a really short change, one can read it fast." P10 feels that "its definitely easier to review small changes", a sentiment that was also echoed by P6. P7 and P8 mention that in their company, changes are bucketed into four sizes (extra small, small, medium and large), as the company had found that there is a considerable decrease in reviewability when a change goes from being medium in size to large. The exact number of lines that places a change into one of these categories is unknown. However, our interviewees stated that the general rule of thumb is that for a change to be reviewable it must be at most 250 lines long. However, P9 contends that one cannot generalize size being an issue or a helpful aspect of reviewability. In his view, a lot depends on the nature of the change. Some issues need larger volumes of code and this is, at times, inescapable. He prefers to measure the size of the change by "[counting] the functionality to see how much functionality is added", but he concedes that this is not trivial.

Commit history. The consensus among our interviewees was that the commit messages must be self-explanatory and cannot have non-descriptive idioms such as "Fix typo" or "Update".

This theme is especially important to interviewee P5, who remarked that "First I check if the commit history is normal … not 'fix this', 'fix that' in there because people can just use rebase to clean that up." P9 also mentions that an author who wants to make a PR more reviewable should "make each commit self-contained." All of our interviewees remarked on the issue of having multiple commits in a code change versus having one single merged commit. They were primarily in favor of having a single merged commit that could stand on its own. If this is not the case, then the interviewees did agree that it made it harder to review the code change. For instance, P4 remarked that he would have expected "the developer to clean up the commit structure before the pull request" as he did not see the value of having multiple commits in a single change. P6 mentioned that he would have kept minor fixes or changes in his local branch and then merged these commits and pushed the PR as a single commit, as he felt "[such commit history behavior made] it easier for the reviewer to follow the change logic." There are certain circumstances where multiple commits in a code change are deemed acceptable. P7 remarked, while looking at the commit history of a large PR he was reviewing, that "it has been extremely helpful to have small and focused commits." However, the commits must be self-contained and have meaningful commit messages. Furthermore, interviewees P9 and P10 felt that the ratio of commits in a change to the number of files changed should not be high.

Test inclusion. According to our interviewees, the integration of tests is *not* vital in aiding the understanding of a change under review. Nine interviewees say that the inclusion of tests is a quick check: They only look at the presence of tests and whether the change has impacted the test coverage.

P1 mentions that the lack of test inclusion does not impact the ability to understand or assess a PR. He could evaluate the code visually and appreciate its impact. P9 echoed a similar sentiment, because he understands the effects of most changes he sees in his project without the aid of tests. Out of our interviewees, only P5 mentioned that he looks at the tests first when reviewing a change to understand the nature of the change and what the developer is trying to do. All our interviewees mention that the presence of tests does have an impact on their decision to accept/reject the

change. P3 said: "Sometimes one has to reject it if the code is not documented properly and if there are too few test cases that cover the changes." P7 mentions that a PR with no tests that cover the change would not be reviewed. P9 put it as: "when test coverage goes down its a smell", so when he sees a change that has a negative impact on the code coverage, the patch is rejected.

Additional factors. All our interviewees mention that having multiple issues addressed in the same code change is something that they deem unacceptable. However, this is not necessarily due to a reduced understanding of the change, for example, P5 says that "you should have just one pull request per issue because it keeps your history cleaner, otherwise it is more difficult to revert." But P5 also concedes that splitting different code modifications into separate changes can aid the reviewability of the patch. P9 echoed a similar sentiment and said that in his project they might even reject a code change outright if this was not done.

P1 mentions that one reason he has rejected a code change in the past is that someone had "added something to a highly contested file and [added it] to the top." There is a preference that in a file to which many developers are contributing that a developer always adds new features at the bottom. P4 agrees with this perspective, he felt that an exception to this convention is only made if there is the addition of a global variable at the top. Additionally, he felt that semantically similar changes should be grouped as this aids in improving the reviewability of the change.

5.1.2 Impact of reviewability on acceptance

We asked our interviewees as to how serious they felt the aforementioned reviewability issues were when it came to accepting or rejecting a code change. Only P9 mentioned that they would take a serious view of such problems and outright reject a change. He felt that "especially if [in his] estimate that it will take 3 or 4 times to go back and forth to get to the end result", he would end up addressing the issue himself and discard the change.

Our industrial interviewees have a different perspective on this. P1 mentions that—in a commercial setting—a patch is not merely outright rejected if it fails some reviewability criteria, instead he speaks of a *temporary rejection*. In his opinion, this is because "there are open lines of communication", thus, a reviewer can talk to the developer who created the change and explain what needs to be fixed before accepting it. P5 too agrees with this sentiment: "it might be a soft reject … [giving] him a chance to clean it up first." P4 mentions too that for small issues such as fixing typos in commits or code, it is a convention that the reviewer makes the fixes himself. In their company, it is protocol that the original author only handles larger issues. However, in the company where P6 works, such issues while not deserving of an outright reject, would be expected to be fixed by the original author.

5.1.3 Improving the reviewability

We ask our interviewees as to how contributors can improve the reviewability of a code change. P1, P9-10 felt that in OSS development one should learn from previously accepted patches.

OSS projects have contribution guidelines, but P6 feels that these manual checks are not ideal. In his opinion, it would be beneficial to have "automating checks … so I don't have to go around commenting just because the syntax is not correct." He proposes that the first round of reviews is done with the aid of review bots before the change is sent to a human reviewer. This step should help the author clean up the code change and prevent him from submitting a code change where reviewability is an issue. This step is also beneficial for the reviewer, as he does not have to do a lot of boilerplate review work and can focus on the semantics of the change.

> Interviewed developers primarily focus on three aspects for reviewability, *i.e.*, code churn, change description and commit history. Surprisingly, test inclusion does not play a significant role in reviewability and the connection between reviewability and acceptance is weak.

5.2 RQ2.b – Feedback on reviewability

Overall, we received feedback on 98 real-world code changes from 35 reviewers working on six industry projects and three OSS projects.

5.2.1 Manual analysis

We manually analyzed each of these rated code changes from three angles: the (A1) *feedback on the reviewability* (as provided via our tool), the (A2) *review comments* (as provided through the standard code review tool), and the (A3) *content* of the code change and the accompanying information provided by the author.

A1 – *Feedback*: We read the feedback provided to us by the reviewer and tried to assign it to one or more of the themes previously emerged from RQ1. The list of themes was exhaustive enough to cover all the feedback: This strengthens the credibility of the completeness of the list. We assigned a score to each mentioned theme, according to whether it was mentioned positively ('+1'), neutrally ('0'), or negatively ('-1') with respect to reviewability. A score of '0' was also used when there was no comment on that theme.

A2 – *Review comments*: In addition to the analysis of the feedback, we also analyzed the comments that reviewers put in the review of the proposed code change through the code review tool. We assigned these comments to one or more themes (again, the list of themes was found to be exhaustive) and assigned a ('+1'/'0'/'-1') score to each theme using the same aforementioned semantic.

A3 – *Content*: Finally, we inspected the content of the source code change and the accompanying information, by analyzing it in terms of the themes emerged from RQ1 that we could objectively assess without being developers of these systems. Specifically, we examined 'change description' (whether the rationale and/or the behavior of the change was described), 'test inclusion' (whether tests accompany the changed production code), 'composite change' (whether one issue only is addressed), 'code churn' (how many lines/files were added/removed), and 'change scope' (how many packages and files are involved in the change). For the "binary" themes (*i.e.*, test inclusion, change description and composite changes), we assign a score of '+1' if the theme is present, '-1' otherwise. For the "numerical" themes (*i.e.*, 'code churn' and 'change scope'), we assign a score of '+1' when the value is less or equal than the median across all the changes, '0' when it is between the median and the upper quartile, and '-1' when is above the upper quartile.

5.2.2 Results

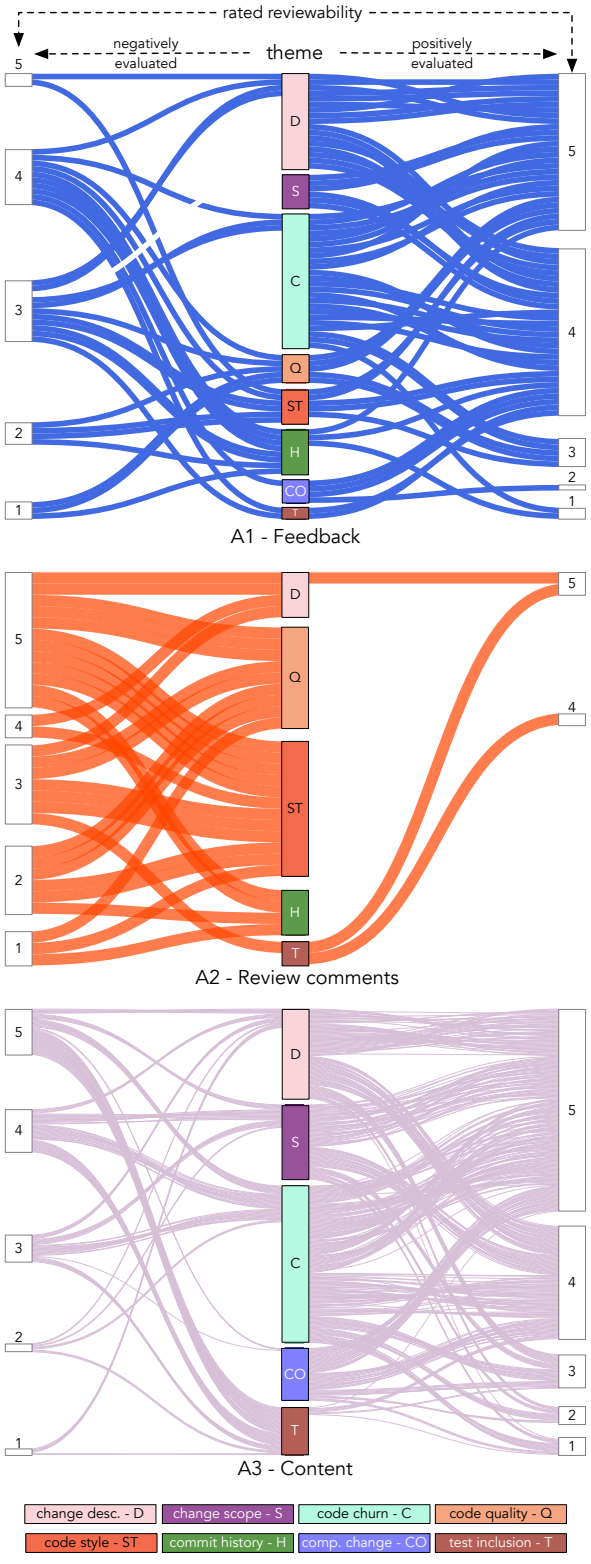

Figure 3: The themes and the rated reviewability.

Figure 3 summarizes the results. In this figure, if a code change has a negative score for one of the themes from one of the three angles (*e.g.*, the feedback remarks about missing tests, *i.e.*, 'test inclusion' for feedback (A1) would be '-1'), we draw a line from the block denoting this theme to the final reviewability rating (as indicated by the reviewers in their feedback) of the change. For positive scores, a line is drawn to the right from the theme to the rated reviewability. The figure is divided in three parts denoting the source of the remark (*e.g.*, A1, feedback, is the first with blue lines). In case of a neutral score ('0'), we do not draw a line but increase the height of the block representing the theme.

Never mentioned themes. Analyzing the results from a high-level perspective, we found that no concepts related to the themes of 'subsystem hotness' and 'nature of the change' were ever mentioned in either the feedback or the review comments.

Negative scores. Figure 3 shows that most review comments (A2, orange lines) are in a negative sense and predominantly focus on code style and code quality aspects. The association between the negative rating of these factors on the reviewability rating is not prominent. For code style, the negative comments are evenly divided across 1–4 star rated changes; for code quality, the negative ratings are distributed across 2–4 star changes. The feedback (A1) provided by the reviewer on these changes mirrors the comments that the reviewers post in the review comments.

Sometimes feedback (A1, blue lines) provides an unfavorable view of the commit history, however, overall this is not associated with a low star rating. There are only isolated cases in which code churn, composite changes, test inclusion and change description are mentioned in the feedback (A1) or the reviewer comments (A2) with a negative connotation. We see that in these cases the lowest the change is rated is 3 stars. Through the analysis of the change (A3, purple lines), we found that most changes do not include tests. We observe that reviewers rarely comment (A2) on this and little feedback (A1) mentions it. Despite the lack of testing, the corresponding code changes are still rated with 4 or 5 stars.

Positive scores. Predominantly positive feedback (A1, blue lines) is provided in the cases of code churn and code description and these changes are mostly scored with either 4 or 5 stars. In the analysis of the content (A3, purple lines), the same themes are largely scored positively and are associated with a higher reviewability rating. In the case of test inclusion and change description, there are few positive reviewer comments in the review and in both cases, the positive scores correspond to 4 or 5 star rated changes.

For most of the themes, when there is a positive score, it is for a change with a 4 or 5 star rating. In a few cases, a positive score relates to a 1 or 2 star. One such instance is where a positive score for code churn relates to a 1 star rated change. This specific change is an outlier as the reviewer appears to understand the patch as the positive comment mentions the patch was small, however, the change had no description, thus it was rated poorly.

> Positive characteristics concerning code churn and change description are mostly found in changes that are rated highly when it comes to reviewability. Most changes that lack of test inclusion still receive a high reviewability rating.

6 CODE CHANGE REVIEWABILITY

In the first phase of this study (RQ1), we found ten themes that are (considered to be) related to the outcome of a code review. Considering the semantic of these themes, it was reasonable to think that some of these themes could contribute to the degree of reviewability of a change. From the analysis of the reviewers' perception from the interviews (RQ2.a), three factors (from the list of the ten themes) were considered more important to define reviewability - change description, code churn and commit history. Factors such as composite changes and code style were mentioned to a lesser degree. Triangulating this finding with the analysis of the real-world reviewed code changes for which we collected feedback on reviewability (RQ2.b), we found that change description and code churn are perceived as major factors of reviewability.

> Code change reviewability can be empirically defined as how well a change is (1) explained ('change description' and 'commit history'), (2) properly sized and self-contained ('code churn' and 'composite changes'), and (3) aligned with the project's style ('code style'). The three factors that most significantly define reviewability are the quality of the description and the commit history, as well as the size.

7 DISCUSSION

In this section, we discuss our main findings, suggest recommendations for practitioners, and consider the implications for researchers. Subsequently, we discuss the evaluation of the results and the quality criteria for our study.

7.1 Reviewability and rejection decisions

We found that—in the industrial setting—reviewability aspects are not the principal drivers behind the overall decision made on the code change. In fact, our industrial interviewees consider reviewability aspects as easily fixable and only temporarily reject a change.

Our OSS interviewees, instead, take a more extreme view of reviewability factors and consider it a good enough reason to reject a code change. In their opinion, when a contributor submits a change, it should already be of sufficiently high quality so that it is easily reviewable. If they sense that they themselves will have to improve certain aspects of a patch, they are inclined to rejection.

Despite the difference in the two settings, we see that there is an additional reviewer overhead that is caused by the presence of reviewability issues. One way to avoid either a temporary or permanent rejection of a code change is to have an automated way of assessing important reviewability aspects of a code change. This would give a contributor immediate feedback on the code change and allow him to improve it before an actual reviewer assesses it.

Currently, many automated tools aid a reviewer in analyzing the impact of a change. For example, tools such as Codecov [15], Coveralls [16] and Operias [64] automatically analyze the code coverage change on a GitHub pull request and provide a detailed report on the same to the contributor and reviewer. Continuous integration tools such as Travis CI [30] that help a contributor and reviewer understand if a GitHub pull request breaks the build.

Recommendations and implications. Currently, no tool performs automated checks for reviewability factors. Our interviewees agreed that having a tool that performed the boilerplate review and checked the reviewability aspects in an automated manner would reduce reviewer burden and allow a reviewer to focus on the semantics of a change. Given the empirical definition of code change reviewability that we found and the key factors, the possibility of designing and deploying such a tool seems to be in the reach, based on current research results (*e.g.*, in change decomposition [34, 45]). We envision a *tool* that would score a change based on its change descriptions' quality, commit history, size, and adherence to the project style; this tool will give authors an opportunity to improve before the review. This is a ripe opportunity to gain more practical impact with current software engineering research.

Our results also show that there is a considerable difference between the acceptance criteria in industry and in OSS. Given that most studies that investigate the acceptance of a patch focus on open source projects, we question if the same results would hold in industry given the difference in socio-technical factors. Further studies can be designed and conducted to investigate this dichotomy between two different development settings further.

7.2 Review comments vs. collected feedback

In the collected reviewability feedback data, for factors such as code style, code quality and commit history, the comments are only negative, instead, the feedback provided via our tool was also positive for these factors. Despite being possible reviewers never directly post a review comment commending the author's effort; at times the same happened when the reviewer had a negative remark.

Positive reinforcement has been shown to play a role in inducing employees of companies to achieve peak productivity and quality of work [44]. Praising change authors for the positive aspects of their contribution could be beneficial irrespective of the authors' seniority within the project, although this could be even more valuable for newcomers [59]. Furthermore, authors could get a better understanding of what aspects of their code change were considered as good by reviewers, which would lead them to adhere to similar standards with their future contributions.

Recommendations and implications. In the current deployment of our code change reviewability feedback collection tool, we provide authors with a dashboard where they can see how reviewers rate their contributions and can read the feedback written by the reviewers (in anonymized form). Authors can see how long it has taken to review their contributions and what factors contribute positively and negatively to the reviewability of their contributions. However, this report dashboard is only a beginning: Investigating whether and how this information is useful and how it should be integrated into existing code review platforms is an open research problem which we leave as future work. Moreover, further research can be conducted on how to incentivize reviewers in also providing positive feedback when reviewing code changes.

7.3 Test inclusion and reviewability

Past research found that the integration of tests is critical to the acceptance of a code change. For example, Gousios *et al.* [53] find that reviewers use lack of testing as a reason to reject a code change.

Tsay *et al.* [76] showed that project maintainers always ask for test cases to be included in the code change so that a reviewer can confirm if the code change does solve the issue it claims that it fixes. Even gray literature sources state that the anatomy of a good code change includes good tests that cover the change and adequately test the behavioral aspects of the change. Contribution guidelines for large open source projects such as Spring Framework [27], JUnit [10], and Mockito [11] all state that any contribution must be well tested. Mockito even has an automated code coverage tool installed on its repository on GitHub that checks the impact a contribution has on the overall code coverage of the project, thereby making it easier for reviewers to spot a change that is untested or insufficiently tested.

Our interviewees mention that for them test inclusion is simply a binary check, they see if there are tests or not. However, they do not consider the test cases as vital to understanding the change and can simply review the change even in the absence of test cases. The data we collect from our code change reviewability feedback collection tool echoes this behavior.

We see that for a majority of the code changes there are no tests included. Reviewers do not appear to mind this behavior, there is only one isolated case where the reviewer commented on the lack of testing. Our findings are in line with those reported by Spadini *et al.* [73]. For the other changes, it seems that by simply inspecting the change the reviewer can judge its correctness.

Recommendations and implications. Previous work has shown that inclusion of tests in a submitted change is pivotal to its chances of acceptance; conversely, we observed an unexpected lack of concern about test inclusion, particularly when it comes to the reviewability aspects of a change. Future research work should be designed and conducted to, on the one hand, understand the reason for these developers' perception and, on the other hand, provide further empirical evidence on the benefits of including tests in a code change as well as how to persuade practitioners in the importance of testing.

7.4 Results evaluation and quality criteria.

Credibility. To strengthen the credibility of our study, we triangulated insights from different data sources: We performed a multivocal literature survey involving both white (22) and gray (21) sources, we gathered qualitative data from 10 professional developers who work in industrial and OSS settings, and we gathered feedback on reviewability of 98 real-world code changes. Two authors did the manual analysis of the 98 rated code changes. To measure their inter-rater reliability (Krippendorf α value [60]), they had an overlapping set of 15 instances along seven dimensions: On these, the reliability was above 0.9 (*i.e.*, very good).

Originality. Previous studies in the context of code review study the quality of a code change based on its acceptance. However, other factors can have an impact on the reviewer and the ease of the review. With this study, we identify factors that affect a change's reviewability and provide its first empirically-based definition.

Usefulness. In this study, the factors that we identify that contribute to reviewability is actionable for change authors and reviewers alike. Furthermore, our study highlights the need for a new tool that automatically assesses reviewability and provides an author with feedback before the code review process starts.

Interviewer bias. The results might be influenced by our own biases that might have led interviewees giving us the answers we wanted, *e.g.*, by driving interviewees to provide more desirable answers [55]. To mitigate this issue we collected independent data from developers by asking them to give us feedback on real-world changes, without our involvement in the feedback process.

Prior knowledge bias. Previous knowledge of the field helps a researcher interpret select lines of inquiry and events, however, it might blind the researcher to alternative explanations for a phenomenon [50]. We address this issue by ensuring apart from the first author, the other authors were involved in the process of interpreting the literature, interview, and feedback data.

External validity. In this study, we collect data from three different sources - literature, developer interviews and developer feedback. We do not have a specific focus on a single programming language, ecosystem or platform. Furthermore, we strive to include data from both industrial and academic sources in our analysis.

8 CONCLUSION

In this paper, we present a study we conducted to gain an empirical understanding of what makes a code change easier to review: We seek a definition of code change reviewability. Starting from the themes that have an impact on the acceptance of a change, we find that there are three primary factors (code churn, change description and commit history) that define a change's reviewability. Our novel findings include:

(1) The first empirical definition of reviewability (code churn, change description and commit history) based on several sources. This can be used to inform the creation of a tool that is able to automatically assess the reviewability of a change.
(2) Observe that reviewability factors are seen in different light in industry as opposed to OSS projects, thus questioning the transferability of existing research conducted in OSS settings to industrial settings.
(3) Empirical evidence on the substantial difference between reviewability and acceptability. This is important for research (*e.g.*, considering "acceptance" as a proxy for discovering features of a good patch would lead to biased results).
(4) Evidence that reviewers provide little to no feedback on reviewability to authors. This is important for practitioners (who must rethink positive/negative reinforcement) and research (to investigate ways to trigger this feedback).
(5) Evidence that investigating the gray literature contributes to a more complete synthesis of information than is possible using only the white literature.

Based on our findings, we provide recommendations to both researchers and practitioners alike. We hope that the insights we have uncovered lead to a more efficient code review process.

ACKNOWLEDGMENTS

A. Bacchelli gratefully acknowledges the support of the Swiss National Science Foundation through the SNF Project No. PP00P2_170529.

REFERENCES

[1] 2014. Ionic. The Art of a Successful Pull Request. https://blog.ionic.io/pull-requests/. (2014). Accessed: 2018-01-31.

[2] 2014. Radify. Perfect Pull Requests. http://radify.io/blog/perfect-pull-requests/. (2014). Accessed: 2018-01-31.

[3] 2015. M. Seemann. Ten tips for better Pull Requests. http://blog.ploeh.dk/2015/01/15/10-tips-for-better-pull-requests/. (2015). Accessed: 2018-01-31.

[4] 2015. Structured Procrastination. Why and how to correctly amend GitHub pull requests. https://blog.adamspiers.org/2015/03/24/why-and-how-to-correctly-amend-github-pull-requests/. (2015). Accessed: 2018-01-31.

[5] 2016. Alpha's Manifesto. How to create a good pull request. https://blog.alphasmanifesto.com/2016/07/11/how-to-create-a-good-pull-request/. (2016). Accessed: 2018-01-31.

[6] 2016. Dev. Pull Requests: The Good, The Bad and The Ugly. https://dev.to/backendandbbq/pull-requests-the-good-the-bad-and-the-ugly. (2016). Accessed: 2018-01-31.

[7] 2016. Elastic. The Art of a Pull Request. https://www.elastic.co/blog/art-of-pull-request. (2016). Accessed: 2018-01-31.

[8] 2017. Anorgan's Blog. Preparing Your Pull Request For Code Review. https://blog.anorgan.com/2017/04/27/preparing-your-pull-request-for-code-review/. (2017). Accessed: 2018-01-31.

[9] 2017. D. Merejkowsky. Lessons Learned From A Failed Pull Request. https://dmerej.info/blog/post/lessons-learned-from-a-failed-pull-request/. (2017). Accessed: 2018-01-31.

[10] 2017. JUnit Team. Guide for contributors. https://github.com/junit-team/junit5/blob/master/CONTRIBUTING.md. (2017). Accessed: 2018-01-31.

[11] 2017. Mockito. Guide for contributors. https://github.com/mockito/mockito/blob/release/2.x/.github/CONTRIBUTING.md. (2017). Accessed: 2018-01-31.

[12] 2017. S. Nonnenberg. Top ten pull request review mistakes. https://blog.scottnonnenberg.com/top-ten-pull-request-review-mistakes/. (2017). Accessed: 2018-01-31.

[13] 2018. Atlassian Blog. The (written) unwritten guide to pull requests. https://www.atlassian.com/blog/git/written-unwritten-guide-pull-requests. (2018). Accessed: 2018-01-31.

[14] 2018. Bugzilla. https://www.bugzilla.org/. (2018). last accessed March 2018.

[15] 2018. Codecov tool. https://codecov.io/. (2018). last accessed March 2018.

[16] 2018. Coveralls tool. https://coveralls.io/. (2018). last accessed March 2018.

[17] 2018. Django Documentation. Committing code. https://docs.djangoproject.com/en/dev/internals/contributing/committing-code/. (2018). Accessed: 2018-01-31.

[18] 2018. ESEC/FSE - ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. http://www.esec-fse.org/index. (2018). Accessed: 2018-07-31.

[19] 2018. Gerrit Code Review. https://www.gerritcodereview.com/. (2018). Accessed: 2018-01-31.

[20] 2018. GitHub Blog. How to write the perfect pull request. https://github.com/blog/1943-how-to-write-the-perfect-pull-request. (2018). Accessed: 2018-01-31.

[21] 2018. ICSE - International Conference on Software Engineering. http://www.icse-conferences.org. (2018). Accessed: 2018-07-31.

[22] 2018. ICSME - International Conference on Software Maintenance and Evolution. http://conferences.computer.org/icsm/. (2018). Accessed: 2018-07-31.

[23] 2018. Mockito. Guide for contributors. https://github.com/mockito/mockito. (2018). Accessed: 2018-07-31.

[24] 2018. MozReview. http://mozilla-version-control-tools.readthedocs.io/en/latest/mozreview.html. (2018). last accessed March 2018.

[25] 2018. MSR - International Conference on Mining Software Repositories. http://www.msrconf.org. (2018). Accessed: 2018-07-31.

[26] 2018. Reviewboard. https://wiki.mozilla.org/ReviewBoard. (2018). last accessed March 2018.

[27] 2018. Spring Framework. Contribution guidelines. https://github.com/spring-projects/spring-framework/blob/master/CONTRIBUTING.md. (2018). Accessed: 2018-01-31.

[28] 2018. The Apache Software Foundation. Guide for new project contributors. https://apache.org/dev/contributors.html. (2018). Accessed: 2018-01-31.

[29] 2018. TOSEM - ACM Transactions on Software Engineering and Methodology. https://tosem.acm.org. (2018). Accessed: 2018-07-31.

[30] 2018. Travis Continuous Integration tool. https://travis-ci.org/. (2018). last accessed March 2018.

[31] 2018. TSE - IEEE Transactions on Software Engineering. https://www.computer.org/web/tse. (2018). Accessed: 2018-07-31.

[32] 2018. Yeomen. Contributing: pull request guidelines. http://yeoman.io/contributing/pull-request.html. (2018). Accessed: 2018-01-31.

[33] Alberto Bacchelli and Christian Bird. 2013. Expectations, outcomes, and challenges of modern code review. In *Proceedings of the 2013 international conference on software engineering*. IEEE Press, 712–721.

[34] Mike Barnett, Christian Bird, Joao Brunet, and Shuvendu K Lahiri. 2015. Helping developers help themselves: Automatic decomposition of code review changesets. In *Proceedings of the 2015 International Conference on Software Engineering*. IEEE Press.

[35] Olga Baysal, Oleksii Kononenko, Reid Holmes, and Michael W Godfrey. 2012. The secret life of patches: A firefox case study. In *Reverse Engineering (WCRE), 2012 19th Working Conference on*. IEEE, 447–455.

[36] Olga Baysal, Oleksii Kononenko, Reid Holmes, and Michael W Godfrey. 2016. Investigating technical and non-technical factors influencing modern code review. *Empirical Software Engineering* 21, 3 (2016), 932–959.

[37] Moritz Beller, Alberto Bacchelli, Andy Zaidman, and Elmar Juergens. 2014. Modern Code Reviews in Open-Source Projects: Which Problems Do They fix?. In *Proceedings of the 11th Working Conference on Mining Software Repositories*. 202–211.

[38] Amiangshu Bosu, Michaela Greiler, and Christian Bird. 2015. Characteristics of useful code reviews: An empirical study at microsoft. In *Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference on*. IEEE, 146–156.

[39] Jason Cohen. 2010. Modern Code Review. In *Making Software*, Andy Oram and Greg Wilson (Eds.). O'Reilly, Chapter 18, 329–338.

[40] Jason Cohen, Eric Brown, Brandon DuRette, and Steven Teleki. 2006. *Best kept secrets of peer code review*. Smart Bear.

[41] J Corbet. 2018. How to participate in the linux community. http://ldn.linuxfoundation.org/book/how-participate-linux-community. (2018). Accessed: 2018-01-31.

[42] Vicki L Creswell JW, Clark P, John W. J.W. Creswell, V.L. Vicki L Plano Clark, Vicki L.P. Plano Clark, and V.L. Vicki L Plano Clark. 2007. *Designing and Conducting Mixed Methods Research*. 275 pages. https://doi.org/10.1111/j.1753-6405.2007.00096.x

[43] Laura Dabbish, Colleen Stuart, Jason Tsay, and Jim Herbsleb. 2012. Social coding in GitHub: transparency and collaboration in an open software repository. In *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work*. ACM, 1277–1286.

[44] Aubrey C Daniels. 1989. *Performance management: Improving quality productivity through positive reinforcement*. Performance Management Pub.

[45] Martin Dias, Alberto Bacchelli, Georgios Gousios, Damien Cassou, and Stéphane Ducasse. 2015. Untangling Fine-Grained Code Changes. In *Proceedings of the 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering*. 341–350.

[46] Michael Fagan. 2002. Design and code inspections to reduce errors in program development. In *Software pioneers*. Springer, 575–607.

[47] Vahid Garousi, Michael Felderer, and Mika V Mäntylä. 2016. The need for multivocal literature reviews in software engineering: complementing systematic literature reviews with grey literature. In *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*. ACM, 26.

[48] Vahid Garousi, Michael Felderer, and Mika V Mäntylä. 2017. Guidelines for including the grey literature and conducting multivocal literature reviews in software engineering. *arXiv preprint arXiv:1707.02553* (2017).

[49] Barney Glaser. 1998. *Doing Grounded Theory: Issues and Discussions*. Sociology Press.

[50] Barney G Glaser. 1998. *Doing grounded theory: Issues and discussions*. Sociology Press.

[51] Georgios Gousios, Martin Pinzger, and Arie van Deursen. 2014. An exploratory study of the pull-based software development model. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 345–355.

[52] Georgios Gousios, Margaret-Anne Storey, and Alberto Bacchelli. 2016. Work practices and challenges in pull-based development: The contributor's perspective. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*. IEEE, 285–296.

[53] Georgios Gousios, Andy Zaidman, Margaret-Anne Storey, and Arie Van Deursen. 2015. Work practices and challenges in pull-based development: the integrator's perspective. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 358–368.

[54] Vincent J Hellendoorn, Premkumar T Devanbu, and Alberto Bacchelli. 2015. Will they like this?: Evaluating code contributions with language models. In *Proceedings of the 12th Working Conference on Mining Software Repositories*. IEEE Press, 157–167.

[55] Donald C Hildum and Roger W Brown. 1956. Verbal reinforcement and interviewer bias. *The Journal of Abnormal and Social Psychology* 53, 1 (1956), 108.

[56] Yujuan Jiang, Bram Adams, and Daniel M German. 2013. Will my patch make it? and how fast? case study on the linux kernel. In *Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on*. IEEE, 101–110.

[57] Sami Kollanus and Jussi Koskinen. 2009. Survey of software inspection research. *The Open Software Engineering Journal* 3, 1 (2009), 15–34.

[58] Oleksii Kononenko, Olga Baysal, and Michael W Godfrey. 2016. Code review quality: how developers see it. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, 1028–1038.

[59] Vladimir Kovalenko and Alberto Bacchelli. 2018. Code review for newcomers: is it different?. In *Proceedings of the 11th International Workshop on Cooperative and*

Human Aspects of Software Engineering. 29–32.

[60] Klaus Krippendorff. 2011. Computing Krippendorff's alpha-reliability. (2011).

[61] William Lidwell, Kritina Holden, and Jill Butler. 2010. *Universal Principles of Design, Revised and Updated: 125 Ways to Enhance Usability, Influence Perception, Increase Appeal, Make Better Design Decisions, and Teach through Design* (2nd ed.). Rockport Publishers.

[62] Jennifer Marlow, Laura Dabbish, and Jim Herbsleb. 2013. Impression formation in online peer production: activity traces and personal profiles in github. In *Proceedings of the 2013 conference on Computer supported cooperative work*. ACM, 117–128.

[63] Michael C Medlock, Dennis Wixon, Mark Terrano, Ramon Romero, and Bill Fulton. 2002. Using the RITE method to improve products: A definition and a case study. *Usability Professionals Association* 51 (2002).

[64] Sebastiaan Oosterwaal, Arie van Deursen, Roberta Coelho, Anand Ashok Sawant, and Alberto Bacchelli. 2016. Visualizing code and coverage changes for code review. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 1038–1041.

[65] Rohan Padhye, Senthil Mani, and Vibha Singhal Sinha. 2014. A study of external community contribution to open-source projects on GitHub. In *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 332–335.

[66] Achyudh Ram, Anand Ashok Sawant, Marco Castelluccio, and Alberto Bacchelli. 2018. Data and Material for "What Makes A Code Change Easier To Review?". https://doi.org/10.5281/zenodo.1323659. (2018).

[67] P. Rigby, B. Cleary, F. Painchaud, M.A. Storey, and D. German. 2012. Open Source Peer Review–Lessons and Recommendations for Closed Source. *IEEE Software* (2012).

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

[69] Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, and Alberto Bacchelli. 2018. Modern Code Review: A Case Study at Google. In *Proceedings of the 40th ACM/IEEE International Conference on Software Engineering, Software Engineering in Practice Track*. 181–190.

[70] Walt Scacchi. 2007. Free/open source software development: Recent research results and methods. *Advances in Computers* 69 (2007), 243–295.

[71] Marcelino Campos Oliveira Silva, Marco Tulio Valente, and Ricardo Terra. 2016. Does Technical Debt Lead to the Rejection of Pull Requests? *arXiv preprint arXiv:1604.01450* (2016).

[72] Daricélio Moreira Soares, Manoel Limeira de Lima Júnior, Leonardo Murta, and Alexandre Plastino. 2015. Acceptance factors of pull requests in open-source projects. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*. ACM, 1541–1546.

[73] Davide Spadini, Mauricio Aniche, Margaret Storey, Magiel Bruntink, and Alberto Bacchelli. 2018. When Testing Meets Code Review: Why and How Developers Review Tests. In *Proceedings of the 40th ACM/IEEE International Conference on Software Engineering*. 677–687.

[74] Yida Tao, Yingnong Dang, Tao Xie, Dongmei Zhang, and Sunghun Kim. 2012. How do software engineers understand code changes?: an exploratory study in industry. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 51.

[75] B.C. Taylor and T.R. Lindlof. 2010. *Qualitative communication research methods*. Sage Publications, Incorporated.

[76] Jason Tsay, Laura Dabbish, and James Herbsleb. 2014. Influence of social and technical factors for evaluating contribution in GitHub. In *Proceedings of the 36th international conference on Software engineering*. ACM, 356–366.

[77] Jason Tsay, Laura Dabbish, and James Herbsleb. 2014. Let's talk about it: evaluating contributions through discussion in GitHub. In *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*. ACM, 144–154.

[78] Peter Weißgerber, Daniel Neu, and Stephan Diehl. 2008. Small patches get in!. In *Proceedings of the 2008 international working conference on Mining software repositories*. ACM, 67–76.

[79] William Foote Ed Whyte. 1991. *Participatory action research*. Sage Publications, Inc.

[80] Claes Wohlin. 2014. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *Proceedings of the 18th international conference on evaluation and assessment in software engineering*. ACM, 38.

[81] Yue Yu, Huaimin Wang, Vladimir Filkov, Premkumar Devanbu, and Bogdan Vasilescu. 2015. Wait for it: determinants of pull request evaluation latency on GitHub. In *Mining software repositories (MSR), 2015 IEEE/ACM 12th working conference on*. IEEE, 367–371.

[82] Yang Zhang, Gang Yin, Yue Yu, and Huaimin Wang. 2014. A exploratory study of@-mention in github's pull-requests. In *Software Engineering Conference (APSEC), 2014 21st Asia-Pacific*, Vol. 1. IEEE, 343–350.