# When Testing Meets Code Review: Why and How Developers Review Tests

Davide Spadini
Delft University of Technology
Software Improvement Group
Delft, The Netherlands
d.spadini@sig.eu

Maurício Aniche
Delft University of Technology
Delft, The Netherlands
m.f.aniche@tudelft.nl

Margaret-Anne Storey
University of Victoria
Victoria, BC, Canada
mstorey@uvic.ca

Magiel Bruntink
Software Improvement Group
Amsterdam, The Netherlands
m.bruntink@sig.eu

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

## ABSTRACT

Automated testing is considered an essential process for ensuring software quality. However, writing and maintaining high-quality test code is challenging and frequently considered of secondary importance. For production code, many open source and industrial software projects employ code review, a well-established software quality practice, but the question remains whether and how code review is also used for ensuring the quality of test code. The aim of this research is to answer this question and to increase our understanding of what developers think and do when it comes to reviewing test code. We conducted both quantitative and qualitative methods to analyze more than 300,000 code reviews, and interviewed 12 developers about how they review test files. This work resulted in an overview of current code reviewing practices, a set of identified obstacles limiting the review of test code, and a set of issues that developers would like to see improved in code review tools. The study reveals that reviewing test files is very different from reviewing production files, and that the navigation within the review itself is one of the main issues developers currently face. Based on our findings, we propose a series of recommendations and suggestions for the design of tools and future research.

## CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**;

## KEYWORDS

software testing, automated testing, code review, Gerrit

**ACM Reference Format:**
Davide Spadini, Maurício Aniche, Margaret-Anne Storey, Magiel Bruntink, and Alberto Bacchelli. 2018. When Testing Meets Code Review: Why and How Developers Review Tests. In *Proceedings of ICSE '18: 40th International Conference on Software Engineering , Gothenburg, Sweden, May 27-June 3, 2018 (ICSE '18),* 11 pages.

*ICSE '18, May 27-June 3, 2018, Gothenburg, Sweden*


https://doi.org/10.1145/3180155.3180192

## 1 INTRODUCTION

Automated testing has become an essential process for improving the quality of software systems [15, 31]. Automated tests (hereafter referred to as just 'tests') can help ensure that production code is robust under many usage conditions and that code meets performance and security needs [15, 16]. Nevertheless, writing effective tests is as challenging as writing good production code. A tester has to ensure that test results are accurate, that all important execution paths are considered, and that the tests themselves do not introduce bottlenecks in the development pipeline [15]. Like production code, test code must also be maintained and evolved [49].

As testing has become more commonplace, some have considered that improving the quality of test code should help improve the quality of the associated production code [21, 47]. Unfortunately, there is evidence that test code is not always of high quality [11, 49]. Vazhabzadeh *et al.* showed that about half of the projects they studied had bugs in the test code [46]. Most of these bugs create false alarms that can waste developer time, while other bugs cause harmful defects in production code that can remain undetected. We also see that test code tends to grow over time, leading to bloat and technical debt [49].

As code review has been shown to improve the quality of source code in general [12, 38], one practice that is now common in many development projects is to use Modern Code Review (MCR) to improve the quality of test code. But how is test code reviewed? Is it reviewed as rigorously as production code, or is it reviewed at all? Are there specific issues that reviewers look for in test files? Does test code pose different reviewing challenges compared to the review of production code? Do some developers use techniques for reviewing test code that could be helpful to other developers?

To address these questions and find insights about test code review, we conducted a two-phase study to understand how test code is reviewed, to identify current practices and reveal the challenges faced during reviews, and to uncover needs for tools and features that can support the review of test code. To motivate the importance of our work, we first investigated whether being a test changes the chances of a file to be affected by defects. Having found no relationship between type of file and defects, then in the first

phase, we analyzed more than 300,000 code reviews related to three open source projects (Eclipse, OpenStack and Qt) that employ extensive code review and automated testing. In the second phase, we interviewed developers from these projects and from a variety of other projects (from both open source and industry) to understand how they review test files and the challenges they face. These investigations led to the following research contributions:

**(1)** To motivate our study, we first explored whether the type of code (production or test) is associated with future defects. Our results show that there is no association, which suggests that test files are no less likely to have defects in the future and should benefit from code review.

**(2)** We investigated how test code is reviewed and found empirical evidence that test files are not discussed as much as production files during code reviews, especially when test code and production code are bundled together in the same review. When test files are discussed, the main concerns include test coverage, mocking practices, code maintainability, and readability.

**(3)** We discovered that developers face a variety of challenges when reviewing test files, including dealing with a lack of testing context, poor navigation support within the review, unrealistic time constraints imposed by management, and poor knowledge of good reviewing and testing practices by novice developers. We discuss recommendations for practitioners and educators, and implications for tool designers and researchers.

**(4)** We created GerritMiner, an open source tool that extracts code reviews from projects that use Gerrit. When performing a review, GerritMiner provides information regarding files, comments, and reviewers. We designed this tool to help us collect a dataset of 654,570 code reviews from three popular open source, industry-supported software systems. GerritMiner and the dataset we studied are publicly available [7].

## 2 BACKGROUND AND MOTIVATION

Past research has shown that both test code and production code suffer from quality issues [11, 32, 49]. We were inspired by the study by Vahabzadeh *et al.* [46] who showed that around half of all the projects they studied had bugs in the test code, and even though the vast majority of these test bugs were false alarms, they negatively affected the reliability of the entire test suite. They also found that other types of test bugs (*silent horrors*) may cause tests to miss important bugs in production code, creating a false sense of security. This study also highlighted how current bug detection tools are not tailored to detect test bugs [46], thus making the role of effective test code review even more critical.

Some researchers have examined MCR practices and outcomes and showed that code review can improve the quality of source code. For example, Bacchelli *et al.* [12] interviewed Microsoft developers and found that code reviews are important not only for finding defects or improving code, but also for transferring knowledge, and for creating a bigger and more transparent picture of the entire system. McIntosh *et al.* [30] found that both code review coverage and participation share a significant link with software quality, producing components with up to two and five additional post-release defects, respectively. Thongtanunam *et al.* [43] evaluated the impact that characteristics of MCR practices have on software quality, studying MCR practices in defective and clean source code files. Di Biase *et al.* [22] analyzed the Chromium system to understand the impact of MCR on finding security issues, showing that the majority of missed security flaws relate to language-specific issues. However, these studies, as well as most of the current literature on contemporary code review, either focus on production files only or do not explicitly differentiate production from test code.

Nevertheless, past literature has shown that test code is substantially different from production code. For instance, van Deursen *et al.* [21] showed that when refactoring test code, there is a unique set of code smells—distinct from that of production code—because improving test code involves additional test-specific refactoring. Moreover, test files have their own libraries that lead to specific coding practices: for example, Spadini *et al.* [41] studied a test practice called mocking, revealing that the usage of mocks is highly dependent on the responsibility and the architectural concern of the production class.

Furthermore, other literature shows that tests are constantly evolving together with production code. Zaidman *et al.* [49] investigated how test and production code co-evolve in both open source and industrial projects, and how test code needs to be adapted, cleaned and refactored together with production code.

Due to the substantial differences between test code and production code, we hypothesize that how they should be reviewed may also differ. However, even though code review is now widely adopted in both open source and industrial projects, how it is conducted on test files is unclear. We aim to understand how developers review test files, what developers discuss during review sessions, what tools or features developers need when reviewing test files, and what challenges they face.

## 3 SHOULD TEST FILES BE REVIEWED?

Even though previous literature has raised awareness on the prevalence of bugs in test files, such as the work by Vahabzadeh *et al.* [46], it may well be that these type of bugs constitute such a negligible number compared to defects found in production code that investing resources in reviewing them would not be advisable. If test code tends to be less affected by defects than production code, pragmatic developers should focus their limited time on reviewing production code, and research efforts should support this.

To motivate our research and to understand whether test code should be reviewed at all, we conducted a preliminary investigation to see whether test and production code files are equally associated with defects. To study this, we used a research method proposed by McIntosh *et al.* [30] where they analyzed whether code review coverage and participation had an influence on software quality. They built a statistical model using the post-release defect count as a dependent variable and metrics highly correlated with defect-proneness as explanatory variables. They then evaluated the effects of each variable by analyzing their importance in the model [17].

We applied the same idea in our case study; however, as our aim was to understand whether test files are less likely to have bugs than production files, we added the type of file (*i.e.*, either test or production) as an additional explanatory variable. If a difference existed, we would expect the variable to be significant for the model. If the variable was not relevant, a test file should neither increase

or decrease increase the likelihood of defects, indicating that test files should be reviewed with the same care as production files (assuming one also cares about defects with tests).

Similar to McIntosh *et al.* [30], we used the three most common families of metrics that are known to have a relationship with defect proneness as control variables, namely *product metrics* (size), *process metrics* (prior defects, churn, cumulative churn), and *human factors* (total number of authors, minor authors, major authors, author ownership). To determine whether a change fixed a defect (our dependent variable), we searched version-control commit messages for co-occurrences of defect identifiers with keywords like 'bug', 'fix', 'defect', or 'patch'. This approach has been used extensively to determine defect-fixing and defect-inducing changes [27, 28].

We considered the same systems that we used for the main parts of our study (Qt, Eclipse, and Openstack) as they perform considerable software testing and their repositories are available online. Due to the size of their code repositories, we analyzed a sample of sub-projects from each one of them. For QT and Openstack, we chose the five sub-projects that contained more code reviews: *qt*, *qt3d*, *qtbase*, *qtdeclarative*, *qtwebengine* (Qt), and *cinder*, *heat*, *neutron*, *nova*, and *tempest* (Openstack). Eclipse, on the other hand, does not use identifiers in commit messages. Therefore, we used the dataset provided by Lam *et al.* [29] for the *Platform* sub-project. This dataset includes all the bugs reported in the Eclipse bug tracker tool, together with the corresponding commit hash, files changed, and other useful information.

We measured dependent and independent variables during the six-month period prior to a release date in a release branch. We chose the release that gave us at least 18 months of information to analyze. In contrast to McIntosh *et al.* 's [30] work, we calculated metrics at the file level (not package level) to measure whether the file being test code vs. production code had any effect.

We observed that the number of buggy commits was much smaller compared to the number of non-buggy commits, *i.e.*, classes were imbalanced, which would bias the statistical model. Therefore, we applied SMOTE (Synthetic Minority Over-sampling TEchnique) [18] to make both classes balanced. All R scripts are available in our online appendix [7].

To rank the attributes, we used Weka [6], a suite of machine learning software written in Java. Weka provides different algorithms for identifying the most predictive attributes in a dataset—we chose *Information Gain Attribute Evaluation (InfoGainAttributeEval)*, which has been extensively used in previous literature [9, 26, 40]. InfoGainAttributeEval is a method that evaluates the worth of an attribute by measuring the information gain with respect to the class. It produces a value from 0 to 1, where a higher value indicates a stronger influence.

The precision and recall of the resulting model were above 90%, indicating that it is able to correctly classify whether most of the files contain defects, strengthening the reliability of the results.

We ran the algorithm using 10-fold cross-validation. Table 1 shows the results of the importance of each variable in the model as evaluated by *InfoGainAttributeEval*. The variable *is test* was consistently ranked as the least important attribute in the model, while *Churn*, *Author ownership*, and *Cumulative churn* were the most important attributes (in that order) for predicting whether a file will likely contain a bug. This is in line with previous literature.

**Table 1: Ranking of the attributes, by decreasing importance**

| Attribute | Average merit | Average Rank |
|---|---|---|
| Churn | 0.753 ± 0.010 | 1 ± 0 |
| Author ownership | 0.599 ± 0.002 | 2.2 ± 0.4 |
| Cumulative churn | 0.588 ± 0.013 | 2.8 ± 0.4 |
| Total authors | 0.521 ± 0.001 | 4 ± 0 |
| Major authors | 0.506 ± 0.001 | 5 ± 0 |
| Size | 0.411 ± 0.027 | 6 ± 0 |
| Prior defects | 0.293 ± 0.001 | 7 ± 0 |
| Minor authors | 0.149 ± 0.001 | 8 ± 0 |
| Is test | 0.085 ± 0.001 | 9 ± 0 |

From this preliminary analysis, we found that the decision to review a file should not be based on whether the file contains production or test code, as this has no association with defects. Motivated by this result, we conducted our investigation of practices, discussions, and challenges when reviewing tests.

## 4 RESEARCH METHODOLOGY

The main *goal* of our study is to increase our understanding of how test code is reviewed. To that end, we conducted *mixed methods research* [19] to address the following research questions:

**RQ$_1$: How rigorously is test code reviewed?** Previous literature has shown that code changes reviewed by more developers are less prone to future defects [35, 38], and that longer discussions between reviewers help find more defects and lead to better solutions [30, 45]. Based on our preliminary study (Section 3) that showed how the type of file (test vs. production) does not change its chances of being prone to future defects, and to investigate the amount of effort developers expend reviewing test code, we measured how often developers comment on test files, the length of these discussions, and how many reviewers check a test file before merging it.

**RQ$_2$: What do reviewers discuss in test code reviews?** In line with Bacchelli & Bird [12], we aimed to understand the concerns that reviewers raise when inspecting test files. Bacchelli & Bird noted that developers discuss possible defects or code improvements, and share comments that help one understand the code or simply acknowledge what other reviewers have shared. We investigated if similar or new categories of outcomes emerge when reviewing test code compared with production code to gather evidence on the key aspects of test file reviews and on the reviewers' needs when working with this type of artifact.

**RQ$_3$: Which practices do reviewers follow for test files?** Little is known about developer practices when reviewing test files. To identify them, we interviewed developers from the 3 open source projects analyzed in the first two RQs, as well as developers from other projects (including closed projects in industry). We asked them how they review test files and if their practices are different to those they use when reviewing production files. This helped discover review patterns that may guide other reviewers and triangulate reviewers' needs.

**RQ$_4$: What problems and challenges do developers face when reviewing tests?** We elicited insights from our interviews to

highlight important issues that both researchers and practitioners can focus on to improve how test code is reviewed.

In the following subsections, we discuss the three data collection methods used in this research. Section 4.1 describes the three open source projects we studied and Section 4.2 explains how we extracted quantitative data related to the prevalence of code reviews in test files. Section 4.3 discusses the manual content analysis we conducted on a statistically significant data sample of comments pertaining to reviews of test code. Section 4.4 describes the interview procedure used to collect data about practices, challenges, and needs of practitioners when reviewing test files.

## 4.1 Project Selection

To investigate what the current practices in reviewing test files are, we aimed at choosing projects that (1) test their code, (2) intensively review their code, and (3) use Gerrit, a modern code review tool that facilitates a traceable code review process for git-based projects [30].

The three projects we studied in our preliminary analysis (discussed in Section 3), match these criteria: **Eclipse**, **Openstack** and **Qt** and we continue to study these projects to answer our research questions. Moreover, these projects are commonly studied in code review research [24, 30, 43]. Table 2 lists their descriptive statistics.

**Table 2: Subject systems' details after data pre-processing**

| | # of prod. files | # of test files | # of code reviews | # of reviewers | # of comments |
|---|---|---|---|---|---|
| Eclipse | 215,318 | 19,977 | 60,023 | 1,530 | 95,973 |
| Openstack | 75,459 | 48,676 | 199,251 | 9,221 | 894,762 |
| Qt | 159,894 | 8,871 | 114,797 | 1,992 | 19,675 |
| Total | 450,671 | 77,524 | 374,071 | 12,743 | 1,010,410 |

## 4.2 Data Extraction and Analysis

To investigate how developers review test files, we extracted code review data from the Gerrit review databases of the systems under study. Gerrit explicitly links commits in a Version Control System (VCS) to their respective code review. We used this link to connect commits to their relevant code review, obtaining information regarding which files have been modified, the modified lines, and the number of reviewers and comments in the review.

Each review in Gerrit is uniquely identified by a hash code called Change-ID. After a patch is accepted by all the reviewers, it is automatically integrated into the VCS. For traceability purposes, the commit message of the integrated patch contains the Change-ID; we extracted this Change-ID from commit messages to link patches in the VCS with the associated code review in Gerrit.

To obtain code review data, we created GerritMiner, a tool that retrieves *all* the code reviews from Gerrit for each project using the Gerrit REST API [4]. The tool saves all review-related data (*e.g.*, Change-ID, author, files, comments, and reviewers) in a MySQL database. Through GerritMiner, we retrieved a total of 654,570 code reviews pertaining to the three systems. Since we were interested in just production and test files, we only stored code reviews that changed *source code* files (*e.g.*, we did not consider '.txt' file, 'README', JSON files, configuration files). After this process, we were left with 374,071 reviews. Table 2 presents the statistics.

To answer $RQ_1$, we selected only reviews that contained less than 50 files and had at least one reviewer involved who was different than the author [8, 34, 37]. In fact, as explained by Rigby *et al.* [36], a code review should ideally be performed on changes that are small, independent, and complete: a small change lets reviewers focus on the entire change and maintain an overall picture of how it fits into the system. As we were interested in code reviews where reviewers actually examined the code closely, we did not consider reviews where the author was the only reviewer. We also did not consider bots as reviewers (*e.g.*, Jenkins and Sanity Bots). At the end, the distribution of the number of reviewers per review (excluding the author) is the following: 27% have 1 reviewer, 26% have 2, 16% have 3, 10% have 4, and 20% have more than 4.

To understand how often and extensively discussions are held during reviews about test files, we considered the following metrics as proxies, which have been validated in previous literature [42]: number of comments in the file, number of files with comments, number of different reviewers, and the length of the comments. We only considered code reviews that contained at least one comment because we were interested in understanding whether there was any difference between review discussions of test and production files, and reviews that do not contain any discussion are not useful for this investigation.

We used the production file metrics as a baseline and separately analyzed the three different review scenarios based on what needed to be reviewed: (1) both production and test files, (2) only production files, or (3) only test files.

## 4.3 Manual Content Analysis

To answer $RQ_2$, we focused on the previously extracted comments (Section 4.2) by practitioners reviewing test files. To analyze the content of these comments, we performed a manual analysis similar to Bacchelli & Bird [12]. Due to the size of the total number of comments (1,010,410), we analyzed a statistically significant random sample. Our sample of 600 comments was created with a confidence level of 99% and error ($E$) of 5% [44].

The manual analysis was conducted by the first two authors of this paper, using the following process: (1) Each researcher was responsible for coding 300 comments. (2) All the sampled comments were listed in a shared spreadsheet. (3) The researchers worked together to categorize and explain comments, using a *negotiated agreement* technique [39] to achieve agreement. As agreement is negotiated on-the-fly, there is no inter-rater agreement value. Agreement was found after 60 comments, at which point the work continued in parallel. (4) As a starting point, researchers used the same categories described by Bacchelli & Bird [12], including *code improvement*, *understanding*, *social communication*, *defect*, *knowledge transfer*, *miscellaneous*, *testing*, *external impact*, and *review tool*. Furthermore, researchers did a second pass to retrieve more fine–grained information for each category, obtaining more details on what developers discuss. (5) In case of doubt, *i.e.*, the category of a specific comment was not clear to one of the researchers, the category was then analyzed by both researchers together.

### 4.4 Interviews

To answer RQs 3 and 4, guided by the results of the previous RQs, we designed an interview in which the goal was to understand *which practices* developers apply when reviewing test files. The interviews were conducted by the first author of this paper and were semi-structured, a form of interview often used in exploratory investigations to understand phenomena and seek new insights [48].

Each interview started with general questions about code reviews, with the aim of understanding why the interviewee performs code reviews, whether they consider it an important practice, and how they perform them. Our interview protocol also contained many questions derived from the results of previous research questions. Our full interview protocol is available in the appendix [7]. We asked interviewees the following main questions:

(1) What is the importance of reviewing these files?
(2) How do you conduct reviews? Do you have specific practices?
(3) What are the differences between reviewing test files and production files?
(4) What challenges do you face when reviewing test files? What are your needs related to this activity?

During each interview, the researcher summarized the answers, and before finalizing the meeting, these summaries were presented to the interviewee to validate our interpretation of their opinions. We conducted all interviews via Skype. With the participants' consent, the interviews were recorded and transcribed for analysis. We analyzed the interviews by initially assigning codes [25] to all relevant pieces of information, and then grouped these codes into higher-level categories. These categories formed the topics we discuss in our results (Section 5).

We conducted 12 interviews (each lasting between 20 and 50 minutes) with developers that perform code reviews as part of their daily activities. Three of these developers worked on the projects we studied in the previous RQs. In addition, we had one participant from another open source project and 8 participants from industry. Table 3 summarizes the interviewees' demographics.

**Table 3: Interviewees' experience (in years) and working context (OSS project, or company)**

| ID | Years as developer | Years as reviewer | Working context |
|---|---|---|---|
| P1 | 5 | 5 | OSS |
| P2 | 10 | 10 | Eclipse |
| P3 | 10 | 10 | Company A |
| P4 | 20 | 6 | Qt |
| P5 | 13 | 8 | Company B |
| P6 | 5 | 5 | Openstack |
| P7 | 7 | 5 | Company C |
| P8 | 5 | 1 | Company D |
| P9 | 11 | 3 | Company E |
| P10 | 9 | 9 | Company F |
| P11 | 7 | 2.5 | Company D |
| P12 | 6 | 4.5 | Company D |

### 4.5 Threats to Validity and Limitations

We describe the threats to validity and limitations to the results of our work, as posed by the research methodology that we applied.

*Construct validity.* When building our model we assume that each post-release defect has the same importance, when in reality this could not be the case. We mitigate this issue analyzing only the release branch of the systems, which are more controlled than a development branch, to ensure that only the appropriate changes will appear in the upcoming release [30].

The content analysis was performed manually, thus giving rise to potentially subjective judgement. To mitigate this threat, we employ the negotiated agreement technique [39] between the first and second authors until agreement was reached (after 60 comments).

*Internal validity – Credibility.* Threats to *internal validity* concern factors we did not consider that could affect the variables and the relations being investigated. In our study, we interview developers from the studied software to understand how they review test files. Every developer has a specific way of reviewing, which may differ from the practices of other practitioners. We try to mitigate this issue by interviewing a range of developers from different open-source and industry projects. In addition, their interviewees' opinions may also be influenced by other factors, such as current literature on MCR, which could may have led them to social desirability bias [23], or by practices in other projects that they participate in. To mitigate this issue, we constantly reminded interviewees that we were discussing the code review practices specifically of their project. At the end of the interview, we asked them to freely talk about their ideas on code reviews in general.

*Generalizability – Transferability.* Our sample contains three open-source systems, which is small compared to the overall population of software systems that make use of code reviews. We reduce this issue by considering diverse systems and by collecting opinions from other open-source projects as well as from industry.

## 5 RESULTS

In this section, we present the results to our research questions that aimed to understand how rigorously developers review tests, what developers discuss with each other in their reviews of test code, and what practices and challenges developers use and experience while performing these code reviews.

### $RQ_1$. How rigorously is test code reviewed?

In Table 4, we show the distribution of comments in code reviews for both production and test files when they are in the same review, and for code reviews that only contain either type.

**Discussion in code reviews of test files.** The number of test file reviews that contain at least one comment ranges from 29% (in reviews that combine test and production files) to 48% (in reviews that only look at test files). The number of production files that contain at least a single comment ranges from 43% (when together with test files) to 44% (in reviews that only look at production files).

In a code review that contains both types of files, the odds of a production file receiving a comment is 1.90 [1.87 – 1.93] higher than with test files. On the other hand, when a review only contains

**Table 4: The prevalence of reviews in test files vs production files (baseline)**

| Code review | # of files | # of files w/ comments | # of files wo/ comments | odds ratio | # of comments | Avg number of comments | Avg # of reviewers | Avg length of comments |
|---|---|---|---|---|---|---|---|---|
| Production | 157,507 | 68,338 (43%) | 89,169 (57%) | 1.90 | 472,020 | 3.00 | 5.49 | 19.09 |
| Test | 102,266 | 29,327 (29%) | 72,939 (71%) | | 129,538 | 1.27 | | 15.32 |
| Only production | 74,602 | 32,875 (44%) | 41,725 (56%) | 0.86 | 122,316 | 1.64 | 3.95 | 18.13 |
| Only test | 22,732 | 10,808 (48%) | 11,924 (52%) | | 52,370 | 2.30 | 5.15 | 17.01 |

one type of file, the odds of a test file receiving a comment is higher than that of a production file: 1.15 [1.11 − 1.18].

We also observed a large number of files that did not receive any discussion. The number of code files that did not receive at least a single comment ranges from 52% (in reviews that only look at test files) to 71% (in reviews that combine test and production files).

**Discussion intensity in test files.** In the code reviews that contain both types of files, production files received more individual comments than test files (3.00 comments per file for production, 1.27 comments for tests). The difference is statistically significant but small (Wilcoxon p-value< $2.2^{e-16}$, Cliff's Delta=-0.1643); this is due to the large number of files with no comments (median = 0 in both test and production, $3_{rd}$ quantile = 1). The difference is larger when we analyze only files with at least a single comment (Wilcoxon p-value< $2.2^{e-16}$, Cliff's Delta=-0.1385).

Again, numbers change when both files are not bundled in the same review. Code reviews on only production files contain fewer individual comments on average than reviews on only test files (1.64 comments for production, 2.30 comments for tests). The difference is statistically significant but small (Wilcoxon p-value< $2.2^{e-16}$, Cliff's Delta=0.0585).

Production files receive longer comments than test files on average, both when they are in the same review (an average of 19.08 characters per comment in a production file against 15.32 in a test) and when they are not (18.13 against 17.01). The difference is again statistically significant but small (Wilcoxon p-value< $2.2^{e-16}$, Cliff's Delta=0.0888).

**Reviewers of test files.** The number of reviewers involved in reviews containing both files and only tests is slightly higher compared to reviews containing production files. However, from the Wilcoxon rank sum test and the effect size, we observe that the overall difference is statistically significant but small (Wilcoxon p-value< $2.2^{e-16}$, Cliff's Delta=−0.1724).

> ***Finding 1.*** *Test files are almost 2 times less likely to be discussed during code review when reviewed together with production files. Yet, the difference is small in terms of the number and length of the comments, and the number of reviewers involved.*

## RQ$_2$. What do reviewers discuss when reviewing test code?

In Figure 1, we report the results of our manual classification of 600 comments. When compared to the study by Bacchelli & Bird [12] that classified production and test code together, we exclude the 'Testing' category as all our comments were related to test code. In addition, we did not observe any comments related to 'External impact' and 'Review tool'. Interestingly, the magnitude of the remaining outcomes is the same as found by Bacchelli & Bird [12].

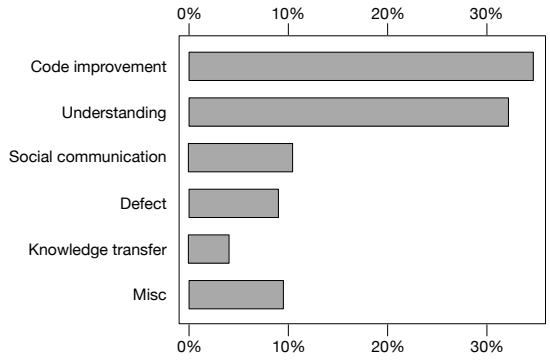


**Figure 1: The outcomes of comments in code review of test files, after manual analysis in 600 comments (CL=99%, CI=5).**

**Code improvements (35%).** This is the most frequently discussed topic by reviewers when inspecting test files. This category includes suggestions to use better coding practices, fix typos, write better Java-docs, and improve code readability. Interestingly, the code improvements that we found are also similar to the ones found by Bacchelli & Bird [12]. Yet the reviewers mostly discuss improvements focused on testing, as opposed to generic code quality practices, such as maintainability, which are the focus of reviews on production code [12, 14].

More specifically, 40% of the code improvement comments concern improvements to testing practices, such as better mocking usage and test cohesion. In addition, we found that in 12% of the cases, developers were discussing better naming for test classes, test methods, and variables. Interestingly, our interviewees mentioned that naming is important when reviewing test files, as P9 put it: "Most of the time I complain of code styling. Sometimes it's difficult to understand the name of the variables, so I often complain to change the names."

Of the code improvement comments, 14% are about untested and tested paths. According to our interviewees, this is an important concern when reviewing test files. Some examples of such review discussion comments are "Where is the assert for the non synchronized case?", "Add a test for context path of /.", and "I wouldn't do this test. That's an implementation detail."

Another 6% of the comments concern wrong assertions. As we discuss in the following RQs, developers often complain about readability of the assertions, namely the assertion has to be as specific as possible to let the developer better understand why the test failed. As an example, a developer asked to change an *assertTrue(equals())* to an *assertEqual()* in one comment.

Finally, we observed that 12% of the comments concern unused or unnecessary code, and 17% of them mention code styling. These kinds of comments are in line with those found on reviews of production code [12, 14].

**Understanding (**32%**).** This category represents all the comments where reviewers ask questions to better understand the code, including posing questions asking for explanations, suggestions, and motivating examples. In this category, we included comments such as "why do you need this for?", "what does this variable name mean?" and "why is this class static?". Interestingly, as opposed to review comments related to code improvements, the comments in this category did not reveal any differences from what we found in the test and production files analyzed in our previous work, *i.e.*, there were no comments that were specifically related to testing practices, such as assertion and mocking. This provides additional evidence on the importance of understanding when performing code reviews [12], regardless of the types of files under review.

**Defect finding (**9%**).** Within this category, we see discussion concerning test defects such as wrong assert handling, missing tests, misuse of test conventions, and incorrect use of mocks. We observe three different categories of defects: severe, not severe, and wrong assertions. More specifically, 43% of the comments are about severe issues, *i.e.*, tests that completely fail because of a wrong variable initialization, a wrong file path, or incorrect use of mocks. On the other hand, 41% of the comments are focused on less severe issues, such as missing test configurations. Finally, 14% of the comments concern wrong assertion handling, such as assertions of wrong scenarios. Interestingly, as opposed to the results reported by Bacchelli & Bird who found that "review comments about defects … mostly address 'micro' level and superficial concerns" [12], a large portion of the defects discussed in test file code reviews concern rather high-level and severe issues. A hypothesis for this difference may be that a good part of the severe, high-level defects that affect test files are localized (*i.e.*, visible just looking at the changed lines), while production files are affected by more delocalized defects that may be harder to detect by simply inspecting the changed lines.

**Knowledge transfer (**4%**).** This category, which also emerged in previous work [12], represents all the comments where the reviewers direct the committer of the code change to an external resource (*e.g.*, internal documentation or websites). We observe two different types of comments in this category: comments that link to external resources and that contain examples. More specifically, 55% of these comments contain links to external documentation (*e.g.*, Mockito website, python documentation), to other classes of the project (*e.g.*, other tests), and to other patches. The rest of the comments are examples where the reviewer showed how to tackle the issue with an example, within the review comment itself, of how s/he would do it.

**Social communication (**11%**).** Finally, this category, in line with the work by Bacchelli & Bird [12], includes all the comments that are social in nature and not about the code, examples such as "Great suggestion!" and "Thank you for your help".

> ***Finding 2.*** *Reviewers discuss better testing practices, tested and untested paths, and assertions. Regarding defects, half of the comments regard severe, high-level testing issues, as opposed to results reported in previous work, where most of the comments on production code regarded low level concerns.*

## $RQ_3$. Which practices do reviewers follow for test files?

We analyze the answers obtained during our interviews with developers. We refer to individual interviewees using ($P_{\#}$).

**Test driven reviews.** Regardless of having test files in the patch, all participants agreed that before diving into the source code, they first get an idea of what the change is about, by reading the commit message or any documentation attached to the review request [$P_{1,2,6-10,12}$]. $P_2$ added: "I look at what it says and I think to myself 'how would I implement that?', just to give a sense of what are the files that I would be expecting to be changed, what are the areas of the system that I'm expecting that they touch." If the files that are attached to the patch look much different from what they expected, they immediately reject the code change [$P_{2,7,9,10}$].

Once they understood what the change is about, developers start to look at the code. In this case, we identified two different approaches: some developers prefer to *read test files first* followed by production files [$P_{2,4,5}$], while other developers prefer to *read production files first* and then review tests [$P_{1,3,6-10}$].

When starting from tests, developers say they can understand what the feature should do even before looking at its implementation [$P_{2,4,5}$]. $P_5$ says: "It is similar to read interfaces before implementations. I start from the test files because I want to understand the API first." In this approach, developers first see what the code is tested for, then check whether the production code does only what is tested for or if it does more than what is necessary [$P_{2,4,5}$]: "If I start to find something in production that is much different of what I am inferring from the tests, those are my first questions" [$P_5$].

More developers instead start reviewing the change in its production files first [$P_{1,3,6-10}$]. As $P_8$ explained: "I start first from the production code, because I can have a sense of what should be tested." The advantage of such an approach is that developers do not waste time validating whether the test covers every path for a change that is wrong in first place [$P_{1,3,6-10}$]. $P_7$ also said that he would prefer to start reviewing the tests, but the poor quality of the tests makes this not realistic: "I would prefer to starting with the tests, the problem is that tests are usually very bad. And the reason is because they usually tend to test what they implemented and not what they should have implemented. In TDD I would also start to review tests first but, as you know, this is not the case."

***Finding 3***. *Similarly to when writing new code, when reviewing some developers prefer to start from tests, others from production. Reviewers who start inspecting tests use them to determine what the production code should do and whether it does only that. Reviewers who start inspecting production code prefer to understand the logic of production code before validating whether its tests cover every path.*

**Reviewers look for different problems when reviewing tests.** According to the interviewees, reviewing test files require different practices to those used for reviewing production files; this is in line with the differences we found in the content of the subcategories of comments left for test files vs. production files for RQ2. Interviewees explain that it is especially different in terms of *what* to look for. $P_{10}$ said that when reviewing production code they discuss more about the design and cleanliness of the code, whereas for reviews of tests they discuss more on tested/untested paths, testing practices like mocking and the complexity of the testing code.

A main concern for all the developers is understanding if all the possible paths of the production code are tested, especially corner cases [$P_{8-12}$]. "If the method being tested receives two variables, and with these two variables it can have on average five to ten different returns, I'll try to see whether they cover a sufficient number of cases so that I can prove that that method won't fail."[$P_{11}$]

Interviewees explained that they often complain about maintainability and readability of the test (*i.e.*, complex or duplicated code, if the test can be simpler, or divided into two smaller tests), but especially the name of the tests and the assertions [$P_{7-12}$]. "I often complain on the assertions, some assertions are really difficult to understand, we should try to be as specific as possible."[$P_8$]

***Finding 4***. *A main concern of reviewers is understanding whether the test covers all the paths of the production code and to ensure tests' maintainability and readability.*

**Having the contextual information about the test.** As we will discuss in the next RQ, a main concern for developers is the small amount of information provided in the code review [$P_{1,6,8-10,12}$]. Indeed, within a code review, reviewers can only see files that are changed and only the lines that have been modified, while interviewees complain that they do not have the possibility to, for example, automatically switch between production code and its test code, or to check other related test cases that are not modified in the patch [$P_{1,8,9}$].

For this reason, two developers explained that they check out the code under review and open it with another tool, for example a local IDE [$P_{9,12}$]. In this way, they can have the full picture of what is changed and get full support of other tools: "I particularly like opening the pull request to know what has come in again, which classes have been edited. I [open] GitHub, I access the PR and open it [in] my IDE. So I can look at the code as a whole, not just the files changed as it is in GitHub."[$P_{12}$] Another advantage of checking out the commit with the patch is that developers can see the tests running: "Most of the time I pull the pull request to see the feature and tests running, so I can have a sense of what have changed and how."[$P_9$] Nevertheless, this practice can only be accomplished when the code base is limited in size and the coding environment can be easily pulled to the local machine of the reviewers.

***Finding 5***. *Due to the lack on test-specific information within the code review tool, we observed that developers check out the code under review and open it in a local IDE: This allows them to navigate through the dependencies, have a full picture of the code, and run the test code. However this workaround is limited to small scale code bases.*

## $RQ_4$. What problems and challenges do developers face when reviewing tests?

**Test code is substantially different than production code.** According to our interviewees, even if sometimes writing tests is simpler than writing production code [$P_{1,3,4,6-9,11}$], this changes when reviewing tests [$P_{3,4,6-9}$]: "Imagine you have to test a method that does an arithmetic sum. The production code only adds two numbers, while the test will have to do a positive test, a negative and have about fifteen different alternatives, for the same function."[$P_{11}$]

According to our interviewees, reviewing test files becomes complicated due to lack of context [$P_{1,6,8-10,12}$]. When reviewing a test, code review tools do not allow developers to have production and test files side-by-side [$P_{12}$]: "Having the complete context of the test is difficult, we often use variables that are not initialized in the reviewed method, so often I have to go around and understand where the variable is initialized."[$P_8$] Another developer said that "It's hard to understand which test is actually testing this method, or this path in the function."[$P_1$] and that when the test involves a lot of other classes (it is highly coupled) s/he never knows whether and how the other classes are tested [$P_9$].

Furthermore, one of the main challenges experienced by our interviewees is that often reviewing a test means reviewing code additions, which is more complicated than reviewing just code changes [$P_{2,11,12}$]. "Code changes make me think, why is this line changing from the greater than side to a less than side? While code additions you have to think what's going on in there, and tests are almost all the time new additions."[$P_2$] According to developers, test code is theoretically written once and if it is written correctly it will not change [$P_2$]. The reason is that while the implementation of the feature may change (*e.g.*, how it is implemented), both the result and the tests will stay the same [$P_{11}$].

***Finding 6***. *Reviewing test files requires developers to have context about not only the test, but also the production file under test. In addition, test files are often long and are often new additions, which makes the review harder to do.*

**The average developer believes test code is less important.** "I will get a call from my manager if there is a bug in production while I'm not going to get a call if there's a bug in the test right?"[$P_2$] According to our interviewees, developers choose saving time to the detriment of quality. This is due to the fact that there is no *immediate* value on having software well tested; as $P_7$ explained "it is only

good for the long term." For a product that is customer-driven, it is more important to release the feature on time without bugs, because that is the code that will run on the client's machine [$P_{2,4-7,9}$]. $P_7$ said: "If I want to get a good bonus by the end of the year I will make sure that my features make it into production level code. If instead we would start to get punished for bugs or bad code practices, you will see a very very different approach, you would see way more discussions about tests than production code." Interviewees affirmed that the main problem is the developers mindset [$P_{1,2,7,8}$]: "It is the same reason as why people write bad tests, testing is considered as secondary class task, it is considered not but it is not."[$P_7$] Developers see test files as less important, because a bug in a test is a developer's problem while a bug in production code is a client's problem. As explained by $P_7$, developers are not rewarded for writing good code, but for delivering features the clients want.

Furthermore, according to our interviewees, during a review sometimes they do not even *look* at the test files, their presence is enough [$P_{1,3-6}$]. As $P_6$ said "Sometimes you don't look at the test because you see there is the file, you know that the code is doing what it has to do and you trust the developer who wrote it (maybe we trust too much sometimes)."

> ***Finding 7.*** *Developers have a limited amount of time to spend on reviewing and are driven by management policies to review production code instead of test code which is considered less important.*

**Better education on software testing and reviewing.** Most interviewees agreed on the need to convince developers and managers that reviewing and testing are highly important for the software system overall quality [$P_{1,2,4,6-8,10}$]. Educating developers on good and bad practices and the dangers of bad testing [$P_{2,7,10}$]. "I would love to see in university people teaching good practice on testing. Furthermore, people coming from university they have no freaking clue on how a code review is done. Educating on testing and reviewing, how to write a good test and review it."[$P_7$]

Furthermore, with the help of researchers, developers could solve part of the education problem: one interviewee said that research should focus more on developers' needs, so that tool designers can take advantage of these needs and improve their tools [$P_6$]. "I think it is important to give this feedback to the people who write [code review] tools so they can provide the features that the community wants. Having someone like you in the middle, collecting this feedback and sending them to tool developers, this could be very helpful."[$P_6$]

> ***Finding 8.*** *Novice developers and managers are not aware of what is the impact of poor testing and reviewing on software quality, education systems should fix this. Moreover, research should focus more on developers' needs and expose them to tool makers to have an impact.*

**Tool improvements.** Part of the interview was focused on what can be improved in the current code review tools.

According to our interviewees, the navigation between the production and the test files within the review is difficult [$P_{2,9,10,12}$]. "We don't have a tool to easily switch between your tests and your production code, we have to go back and forth, then you have to look for the same name and trying to match them."[$P_2$] As mentioned before, the context of the review is limited to the files attached to the review itself, and this makes it difficult to have a big picture of the change. For example, test files are usually highly coupled to several production classes: however, developers can not navigate to the dependencies of the test, or other test in general, without opening a new window [$P_{2,9}$]. "If we could click on the definition of a class and go to its implementation would be amazing. That's why I pull the PR every-time and I lose a lot of time doing it."[P9] P12' said: "It's very unproductive to review in GitHub, because you first visualize all the codes, and then at the end are all the tests, and it ends up being more difficult having to keep going in the browser several times."

In addition, adding fine-grained information about code coverage during the review is considered helpful [$P_{1,7,11,12}$]. More specifically, which tests cover a specific line [$P_{1,7,11,12}$], what paths are already covered by the test suite [$P_{2,12}$], and whether tests exercise exceptional cases [$P_{12}$]. Regarding the latter, $P_{12}$ says: "I think it's harder to automate, but it is to ensure that not only the "happy" paths are covered. It is to ensure that the cover is in the happy case, in case of errors and possible variations. A lot of people end up covering very little or too much."

Tool features that are not related to test also emerged during our interviewees. For example, enabling developers to identify the importance of each file within the code review [$P_{7,8,11,12}$] and splitting the code review among different reviewers [$P_7$].

> ***Finding 9.*** *Review tools should provide better navigation between test and production files, as well as in-depth code coverage information.*

## 6 DISCUSSION

We discuss how our results lead to recommendations for practitioners and educators, as well as implications for future research.

### 6.1 For Practitioners and Educators

**Underline the importance of reviewing test code.** The results of both our quantitative and qualitative analysis indicate that most reviewers deem reviewing tests as less important than reviewing production code. Especially when inspecting production and test files that are bundled together, reviewers tend to focus more on production code with the risk of missing bugs in the tests. However, previous research has shown that bugs in test files can lower the quality of the corresponding production code, because a bug in the test can lead to serious issues such as 'silent horrors' or 'false alarms' [46]. Moreover, our analysis provided empirical evidence that being a test does not change the chances of a file to have future defects (Section 3). For this reason, practitioners should be instructed and keep in mind to put the same care when reviewing test or production code.

**Set aside sufficient time for reviewing test files.** Our interviewees agreed that reviewing test files is a non-trivial task, because changes involving tests are more often code *additions* rather than code *modifications* and several test options must be analyzed. This indicates that correctly reviewing test files would hardly take less time than reviewing production code. A good team culture must be developed, in which the time spent on reviewing test code is considered as important as the time spent on reviewing production code and scheduled accordingly. In fact, as previous work already pointed out [12], good reviewing effectiveness is found mostly within teams that value the time spent on code review; tests should not be treated differently.

**Educate developers on how to review test code.** Many books and articles have been written by practitioners on best practices for code review [1, 2, 5] and researchers continue to conduct studies to increase our empirical understanding of code review [30, 42]. Nevertheless, best practices for reviewing test code have not been discussed nor proposed yet. Our work, as a first step, collects current best practices for reviewing of tests. These practices show us that developers should learn how to look for possible false alarms, to check that the tests will be easily understandable and maintainable, and to check whether all the possible paths of the production code are tested. Novice reviewers may also consider the practice of reviewing test code before production to (1) make sure to give it enough time and (2) to better understand the goals of the production code under test, since novices may not know them beforehand.

## 6.2 For Tool Designers and Researchers

**Providing context to aid in reviewing of tests.** The lack of context when reviewing test code is a concern for many developers. Specifically, developers argue that it is important to understand and inspect the classes that are under test as well as which dependencies are simulated by tests (*i.e.*, mock objects). However, knowing which classes are executed by a test normally requires dynamically executing the code during review, which is not always feasible, especially when large code bases are involved [20]. This is an opportunity to adapt and extend existing research that determines the coverage of tests using static analysis [10]. Moreover, developers would like to be able to easily navigate through the test and tested classes; future research studies could investigate how to improve file navigation for code review in general (an existing open research problem [13]) but also to better support review of tests in particular.

**Providing detailed code coverage information for tests.** As our results show, one of the most important tasks during the review of a test code is to make sure the test covers all the possible paths of the production code. Although external tools provide code coverage support for developers (*e.g.*, Codecov [3]), this information is usually not "per test method", *i.e.*, coverage reports focus on the final coverage after the execution of the entire test suite, and not for a single test method. Therefore, new methods should be devised to not only provide general information on code coverage, but also provide information that is specific to each test method. An effort in this direction has been presented by Oosterwaal *et al.* [33]; our analysis points to the need for further research in this area.

**Understanding how to review test code and benefits of test reviews.** Our research highlights some of the current practices used by developers when reviewing test files, such as test driven review (review tests before production code). Nevertheless, the real effect of these practices on code review effectiveness and on the eventual test code quality is not known: some practices may be beneficial, other may simply waste reviewers' time. This calls for in-depth, empirical experiments to determine which practices should be suggested for adoption by practitioners.

# 7 CONCLUSIONS

Automated testing is nowadays considered to be an essential process for improving the quality of software systems. Unfortunately, past literature showed that test code, similarly to production code, can often be of low quality and may be prone to contain defects [46]. To maintain a high code quality standard, many software projects employ code review, but is test code typically reviewed and if so, how rigorously? In this paper we investigated whether and how developers employ code review for test files. To that end, we studied three OSS projects, analyzing more than 300,000 reviews and interviewing three of their developers. In addition, we interviewed another 9 developers, both from OSS projects and industry, obtaining more insights on how code review is conducted on tests. Our results provide new insights on what developers look for when reviewing tests, what practices they follow, and the specific challenges they face.

In particular, after having verified that a code file that is a test does not make it less likely to have defects—thus little justification for lower quality reviews—we show that developers tend to discuss test files significantly less than production files. The main reported cause is that reviewers see testing as a secondary task and they are not aware of the risk of poor testing or bad reviewing. We discovered that when inspecting test files, reviewers often discuss better testing practices, tested and untested paths, and assertions. Regarding defects, often reviewers discuss severe, high-level testing issues, as opposed to results reported in previous work [12], where most of the comments on production code regarded low level concerns.

Among the various review practices on tests, we found two approaches when a review involves test and production code together: some developers prefer to start from tests, others from production. In the first case, developers use tests to determine what the production code should do and whether it does only that, on the other hand when starting from production they want to understand the logic before validating whether its tests cover every path. As for challenges, developers' main problems are: understanding whether the test covers all the paths of the production code, ensuring maintainability and readability of the test code, gaining context for the test under review, and difficulty reviewing large code additions involving test code.

We provide recommendations for practitioners and educators, as well as viable directions for impactful tools and future research. We hope that the insights we have discovered will lead to improved tools and validated practices which in turn may lead to higher code quality overall.

# ACKNOWLEDGMENTS

D. Spadini gratefully acknowledges the support of SENECA - EU MSCA-ITN-2014-EID no.642954. A. Bacchelli gratefully acknowledges the support of the SNF Project No. PP00P2_170529.

## REFERENCES


[1] [n. d.]. Best Practices: Code Reviews. https://msdn.microsoft.com/en-us/library/bb871031.aspx. ([n. d.]). [Online; accessed 25-August-2017].

[2] [n. d.]. Best Practices for Code Reviews. https://smartbear.com/learn/code-review/best-practices-for-peer-code-review/. ([n. d.]). [Online; accessed 25-August-2017].

[3] [n. d.]. Codecov. https://codecov.io. ([n. d.]). [Online; accessed 25-August-2017].

[4] [n. d.]. Gerrit REST APIs. https://gerrit-review.googlesource.com/Documentation/rest-api.html. ([n. d.]). [Online; accessed 25-August-2017].

[5] [n. d.]. Modern Code Review. https://www.slideshare.net/excellaco/modern-code-review. ([n. d.]). [Online; accessed 25-August-2017].

[6] [n. d.]. WEKA. http://www.cs.waikato.ac.nz/ml/weka/. ([n. d.]). [Online; accessed 25-August-2017].

[7] 2018. Appendix. https://doi.org/10.5281/zenodo.1172419. (2018).

[8] R T Fielding A. Mockus and J D Herbsleb. 2002. Two case studies of open source software development: Apache and Mozilla. *ACM Trans. Softw. Eng. Meth.* 11, 3 (2002), 309–346.

[9] Megha Aggarwal and Amrita. 2013. Performance Analysis Of Different Feature Selection Methods In Intrusion Detection. *International Journal Of Scientific & Technology Research* 2, 6 (2013), 225–231.

[10] Tiago L Alves and Joost Visser. 2009. Static estimation of test coverage. In *Source Code Analysis and Manipulation, 2009. SCAM'09. Ninth IEEE International Working Conference on*. IEEE, 55–64.

[11] Dimitrios Athanasiou, Ariadi Nugroho, Joost Visser, and Andy Zaidman. 2014. Test Code Quality and Its Relation to Issue Handling Performance. *IEEE Trans. Software Eng.* 40, 11 (2014), 1100–1125. https://doi.org/10.1109/TSE.2014.2342227

[12] Alberto Bacchelli and Christian Bird. 2013. Expectations, outcomes, and challenges of modern code review. In *Proceedings - International Conference on Software Engineering*. 712–721. https://doi.org/10.1109/ICSE.2013.6606617

[13] Mike Barnett, Christian Bird, João Brunet, and Shuvendu K Lahiri. 2015. Helping developers help themselves: Automatic decomposition of code review changesets. In *Proceedings of the 37th International Conference on Software Engineering*. IEEE Press, 134–144.

[14] 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*. ACM, 202–211.

[15] Antonia Bertolino. 2007. Software testing research: Achievements, challenges, dreams. In *2007 Future of Software Engineering*. IEEE Computer Society, 85–103.

[16] George Candea, Stefan Bucur, and Cristian Zamfir. 2010. Automated software testing as a service. In *Proceedings of the 1st ACM symposium on Cloud computing*. ACM, 155–160.

[17] J.M. Chambers and T. Hastie. 1992. *Statistical Models in S*. Wadsworth & Brooks/Cole Advanced Books & Software.

[18] Nitesh V. Chawla, Kevin W. Bowyer, Lawrence O. Hall, and W. Philip Kegelmeyer. 2002. SMOTE: Synthetic minority over-sampling technique. *Journal of Artificial Intelligence Research* 16 (2002), 321–357. https://doi.org/10.1613/jair.953 arXiv:1106.1813

[19] 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

[20] Jacek Czerwonka, Rajiv Das, Nachiappan Nagappan, Alex Tarvo, and Alex Teterev. 2011. Crane: Failure prediction, change analysis and test prioritization in practice–experiences from windows. In *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on*. IEEE, 357–366.

[21] Arie Van Deursen, Leon Moonen, Alex Van Den Bergh, and Gerard Kok. 2001. Refactoring Test Code. *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes XP2001* (2001), 92–95. https://doi.org/10.1109/ICSEA.2007.57

[22] Marco di Biase, Magiel Bruntink, and Alberto Bacchelli. 2016. A Security Perspective on Code Review: The Case of Chromium. *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)* (2016), 21–30. https://doi.org/10.1109/SCAM.2016.30

[23] Adrian Furnham. 1986. Response bias, social desirability and dissimulation. *Personality and individual differences* 7, 3 (1986), 385–400.

[24] Kazuki Hamasaki, Raula Gaikovina Kula, Norihiro Yoshida, Camargo Cruz Ana Erika, Kenji Fujiwara, and Hajimu Iida. 2013. Who does what during a Code Review ? An extraction of an OSS Peer Review Repository. *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR' 13)* (2013), 49–52. https://doi.org/10.1109/MSR.2013.6624003

[25] Bruce Hanington and Bella Martin. 2012. *Universal methods of design: 100 ways to research complex problems, develop innovative ideas, and design effective solutions*. Rockport Publishers.

[26] Truong Ho-Quang, Michel R.V. Chaudron, Ingimar Samuelsson, Joel Hjaltason, Bilal Karasneh, and Hafeez Osman. 2014. Automatic Classification of UML Class Diagrams from Images. *2014 21st Asia-Pacific Software Engineering Conference* December (2014), 399–406. https://doi.org/10.1109/APSEC.2014.65

[27] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi. 2013. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering* 39, 6 (June 2013), 757–773. https://doi.org/10.1109/TSE.2012.70

[28] Sunghun Kim, E. James Whitehead, and Yi Zhang. 2008. Classifying software changes: Clean or buggy? *IEEE Transactions on Software Engineering* 34, 2 (2008), 181–196. https://doi.org/10.1109/TSE.2007.70773

[29] An Ngoc Lam, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. 2015. Combining Deep Learning with Information Retrieval to Localize Buggy Files for Bug Reports (N). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. 476–481. https://doi.org/10.1109/ASE.2015.73

[30] Shane Mcintosh, Yasutaka Kamei, Bram Adams, and Ahmed E Hassan. 2014. The Impact of Code Review Coverage and Code Review Participation on Software Quality Categories and Subject Descriptors. *Proceedings of the 11th Working Conference on Mining Software Repositories* (2014), 192–201. https://doi.org/10.1145/2597073.2597076

[31] Glenford Myers. 2004. *The Art of Software Testing, Second edition*. Vol. 15. 234 pages. https://doi.org/10.1002/stvr.322 arXiv:arXiv:gr-qc/9809069v1

[32] Helmut Neukirchen and Martin Bisanz. 2007. Utilising Code Smells to Detect Quality Problems in TTCN-3 Test Suites. *Proceedings of the 19th IFIP TC6/WG6.1 International Conference, and 7th International Conference on Testing of Software and Communicating Systems* (2007), 228–243. https://doi.org/10.1007/978-3-540-73066-8_16

[33] 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.

[34] D. L. Parnas and D. M. Weiss. 1987. Active design reviews: Principles and practices. *The Journal of Systems and Software* 7, 4 (1987), 259–265. https://doi.org/10.1016/0164-1212(87)90025-2

[35] Eric S. Raymond. 1999. The cathedral and the bazaar. *First Monday* 12, 3 (1999), 23–49. https://doi.org/10.5210/fm.v3i2.578

[36] Peter Rigby, Brendan Cleary, Frederic Painchaud, Margaret-Anne Storey, and Daniel German. 2012. Contemporary Peer Review in Action: Lessons from Open Source Development. *IEEE Software* 29, 6 (nov 2012), 56–61. https://doi.org/10.1109/MS.2012.24

[37] Peter C Rigby. 2011. Understanding Open Source Software Peer Review: Review Processes, Parameters and Statistical Models, and Underlying Behaviours and Mechanisms. *ProQuest Dissertations and Theses* (2011), 194. http://search.proquest.com.proxy1.ncu.edu/docview/898609390?accountid=28180

[38] Peter C. Rigby, Daniel M. German, and Margaret-Anne Storey. 2008. Open source software peer review practices. *Proceedings of the 13th International Conference on Software Engineering* (2008), 541. https://doi.org/10.1145/1368088.1368162

[39] Johnny Saldaña. 2015. *The coding manual for qualitative researchers*. Sage.

[40] P M Soni, Varghese Paul, and M Sudheep Elayidom. 2016. Effectiveness of Classifiers for the Credit Data Set : an Analysis. (2016), 78–83.

[41] Davide Spadini, Maurício Aniche, Magiel Bruntink, and Alberto Bacchelli. 2017. To Mock or Not To Mock? An Empirical Study on Mocking Practices. *Proceedings of the 14th International Conference on Mining Software Repositories* (2017), 11.

[42] Patanamon Thongtanunam, Shane McIntosh, Ahmed E. Hassan, and Hajimu Iida. 2017. Review participation in modern code review. *Empirical Software Engineering* 22, 2 (Apr 2017), 768–817.

[43] Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Raula Gaikovina Kula, Norihiro Yoshida, Hajimu Iida, and Ken Ichi Matsumoto. 2015. Who should review my code? A file location-based code-reviewer recommendation approach for Modern Code Review. *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015 - Proceedings* (2015), 141–150. https://doi.org/10.1109/SANER.2015.7081824

[44] Mario Triola. 2006. *Elementary Statistics* (10th ed.). Addison-Wesley.

[45] Jason Tsay, Laura Dabbish, and James Herbsleb. 2014. Let's talk about it: evaluating contributions through discussion in GitHub. *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (2014), 144–154. https://doi.org/10.1145/2635868.2635882

[46] Arash Vahabzadeh and Ali Mesbah. 2015. An Empirical Study of Bugs in Test Code. (2015), 101–110.

[47] Eva Van Emden and Leon Moonen. 2002. Java quality assurance by detecting code smells. *Proceedings - Working Conference on Reverse Engineering, WCRE* 2002-Janua (2002), 97–106. https://doi.org/10.1109/WCRE.2002.1173068

[48] R.S. Weiss. 1995. *Learning from strangers: The art and method of qualitative interview studies*. Simon and Schuster.

[49] a. Zaidman, B. Van Rompaey, S. Demeyer, and a. Van Deursen. 2008. Mining Software Repositories to Study Co-Evolution of Production & Test Code. *2008 1st International Conference on Software Testing, Verification, and Validation* 3 (2008), 220–229. https://doi.org/10.1109/ICST.2008.47