Investigating Code Review Practices in Defective Files: An Empirical Study of the Qt System

Patanamon Thongtanunam*, Shane McIntosh†, Ahmed E. Hassan†, Hajimu Iida*
*Nara Institute of Science and Technology, Japan †Queen's University, Canada
patanamon-t@is.naist.jp, {mcintosh, ahmed}@cs.queensu.ca, iida@itc.naist.jp

***Abstract*—Software code review is a well-established software quality practice. Recently, Modern Code Review (MCR) has been widely adopted in both open source and proprietary projects. To evaluate the impact that characteristics of MCR practices have on software quality, this paper comparatively studies MCR practices in defective and clean source code files. We investigate defective files along two perspectives: 1) files that will eventually have defects (i.e., future-defective files) and 2) files that have historically been defective (i.e., risky files). Through an empirical study of 11,736 reviews of changes to 24,486 files from the Qt open source project, we find that both future-defective files and risky files tend to be reviewed less rigorously than their clean counterparts. We also find that the concerns addressed during the code reviews of both defective and clean files tend to enhance evolvability, i.e., ease future maintenance (like documentation), rather than focus on functional issues (like incorrect program logic). Our findings suggest that although functionality concerns are rarely addressed during code review, the rigor of the reviewing process that is applied to a source code file throughout a development cycle shares a link with its defect proneness.**

***Index Terms*—Code Review, Software Quality**

I. Introduction

Software code review is a well-established software quality practice. Boehm and Basili argue that code review is one of the best investments for defect reduction [1]. Moreover, Shull *et al.* find that code reviews often catch more than half of a product's defects [2]. One of the main goals of code review is to identify weakness in software changes early on the project development cycle. The traditional software inspection is a formal and rigidly structured review activity involving in-person meetings [3]. Prior work has shown that traditional software inspections can successfully improve the overall quality of a software product [3–5]. However, traditional software inspections have received limited adoption in the domain of globally-distributed software development teams [6].

Unlike the formal software inspections of the past, Modern Code Review (MCR) is a lightweight variant of the code review process [7]. MCR is now widely adopted in open source and proprietary projects [8]. Since the code review process of MCR is in stark contrast to the traditional software inspections of the past, many recent studies revisit the findings of the past to better understand the performance of MCR [7–12].

In this paper, we investigate the code review practices of MCR in terms of: 1) code review activity, and 2) concerns addressed during code review. We characterize code review practices using 11 metrics grouped along three dimensions, i.e., review intensity, review participation, and reviewing time. We then comparatively study the difference of these code review practices in defective and clean source code files. We also investigate defective files along two perspectives: 1) files that will eventually have defects (called *future-defective files*), and 2) files that have historically been defective (called *risky files*). Using data collected from the Qt open source system, we address the following two research questions:

(RQ1) Do developers follow lax code review practices in files that will eventually have defects?

Motivation: Our prior work has shown that lax code review practices are correlated with future defects in the corresponding software components [12]. For example, components with many changes that have no associated review discussion tend to have post-release defects. While these prior findings suggest that a total lack of code review activity may increase the risk of post-release defects, little is known about how much code review activity is "enough" to mitigate this risk.

Results: We find that future-defective files tend to undergo reviews that are less intensely scrutinized, having less team participation, and a faster rate of code examination than files without future defects. Moreover, most of the changes made during the code reviews of future-defective files are made to ease future maintenance rather than fix functional issues.

(RQ2) Do developers adapt their code review practices in files that have historically been defective?

Motivation: Since the number of prior defects is a strong indicator of the incidence of future defects [13], the files that have historically been defective may require additional attention during the code review process. In other words, to reduce the likelihood of having future defects, developers should more carefully review changes made to these risky files than changes made to files that have historically been defect-free. However, whether or not developers are actually giving such risky files more careful attention during code review remains largely unexplored.

Results: We find that developers are likely to review changes of risky files with less scrutiny and less team participation than files that have historically been defect-free (i.e., normal files). Developers tend to address evolvability and functionality concerns in the reviews of risky files more often than they do in normal files. Moreover, such risky files are more likely to have future defects if they undergo a large number of lax reviews that focus on evolvability concerns.

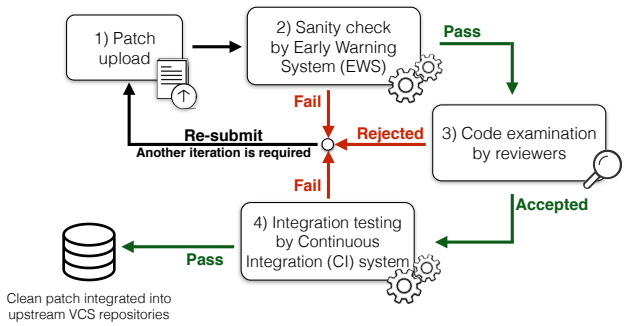


Fig. 1. Gerrit-based code review process of the Qt system

Our results lead us to conclude that lax code review practices could lead to future defects in software systems. Developers are not as careful when they review changes made to risky files despite their historically defective nature. These findings suggest that files that have historically been defective should be given more careful attention during the code review process, since the rigor of the reviewing process shares a link with defect proneness.

Paper organization. The remainder of the paper is organized as follows. Section II describes the code review process of the studied Qt system. Section III describes the design of our empirical study, while Section IV presents the results with respect to our two research questions. Section V discusses broader implications of our findings. Section VI discloses the threats to the validity of our empirical study. Section VII surveys related work. Finally, Section VIII draws conclusions.

II. Qt Code Review Process

The code review process of the studied Qt system is based on Gerrit, which is a popular web-based code review tool that tightly integrates with Version Control Systems (VCSs). Gerrit helps developers to expedite the code review process by interfacing with automated quality assurance bots that check for regressions in system behavior, and automatically integrating changes into upstream VCS repositories after reviewers (and bots) are satisfied. Figure 1 illustrates the Gerrit-based code review process, which consists of four main steps:

1) Patch upload. An author uploads a patch (i.e., a set of proposed changes) to Gerrit and invites a set of reviewers to critique it. During code review, the original author or other authors can submit several revisions to improve the patch.

2) Sanity check by Early Warning System (EWS). The EWS is an automated testing bot developed by the Qt team. The goal of the EWS is to provide quick feedback that can be easily revised before reviewers examine the patch. The EWS runs a set of tests that detect basic errors, such as violations of the Qt coding and documentation style. If the patch does not pass the EWS testing, the patch is rejected. The author can later revise the patch and upload a new revision.

3) Code examination by reviewers. Reviewers examine the technical content of a patch and provide feedback by posting a message to a general discussion thread or inserting inline comments within the patch itself. To decide whether a patch should be integrated into upstream VCS repositories or abandoned, reviewers assign a score ranging from -2 to +2 in order to indicate agreement (positive value) or disagreement (negative value). The author revises the patch to address reviewer feedback until at least one reviewer agrees with the patch and assigns it a score of +2.

4) Integration testing by Continuous Integration (CI) system. The CI bot performs more rigorous regression testing prior to eventually integrating clean patches into upstream VCS repositories. If the patch does not pass this integration testing, the patch needs to be revised according to the CI bot report, and a new revision needs to be uploaded. Once the patch satisfies the requirements of the CI bot, the patch is automatically integrated into upstream VCS repositories.

TABLE I
An overview of the studied Qt system.

Version	LOC	Commits		Files	
		w/ Review	Total	w/ 100% review coverage	Changed Files
Qt 5.0.0	5,560,317	9,677	10,163	11,689	25,615
Qt 5.1.0	5,187,788	6,848	7,106	12,797	19,119

III. Case Study Design

In this section, we describe the studied system, and our data preparation and analysis approaches.

A. Studied System

In order to address our research questions, we perform an empirical study on a large, rapidly-evolving open source system with a globally-distributed development team. In selecting the subject system, we identified two important criteria:

Criterion 1: Active MCR Practices — We want to study a subject system that actively uses MCR for the code review process, i.e., a system that examines and discusses software changes through a code review tool. Hence, we only study a subject system where a large number of reviews are performed using a code review tool.

Criterion 2: High Review Coverage — Since we will investigate the differences of MCR practices in defective and clean files, we need to ensure that a lack of code review is not associated with defects [12]. Hence, we focus our study on a system that has a large number of files with 100% review coverage, i.e., files where every change made to them is reviewed.

Due to the human-intensive nature of carefully studying the code reviewing process, we decide to perform an in-depth study on a single system instead of examining a large number of projects. With our criteria in mind, we select Qt, an open source cross-platform application and UI framework developed by the Digia corporation. Table I shows that the Qt system satisfies our criteria for analysis. In terms of criterion 1, the development process of the Qt system has achieved a large proportion of commits that are reviewed. In terms of criterion 2, the Qt system has a large number of files where 100% of the integrated changes are reviewed.

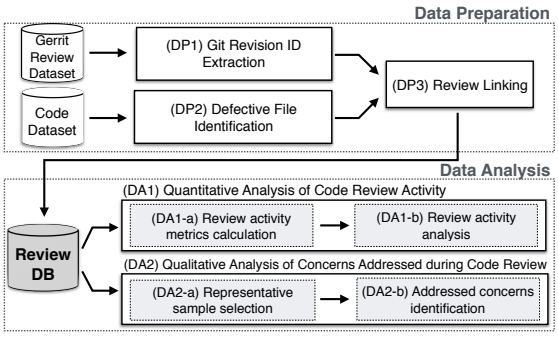


Fig. 2. An overview of data preparation and data analysis approaches.

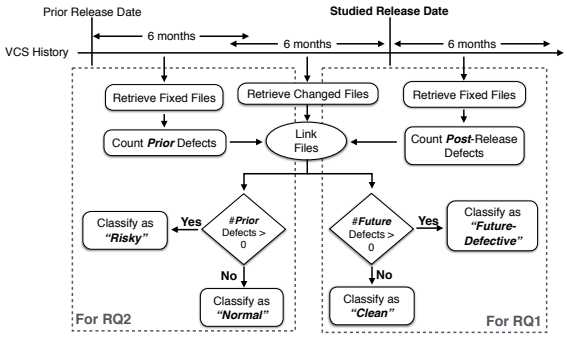


Fig. 3. Our approach to identify defective files.

B. Data Preparation

We used the Gerrit review and code datasets that are provided by prior work [12, 14]. The Gerrit review dataset of Hamasaki *et al.* describes patch information, reviewer scoring, the personnel involved, and review discussion history [14]. The code dataset of McIntosh *et al.* describes the recorded commits on the `release` branch of the Qt VCSs during the development and maintenance of each studied Qt release [12]. For each of our research questions, we construct a review database by linking the Gerrit review and code datasets. Figure 2 provides an overview of our dataset linking process, which is broken down into three steps that we describe below.

(DP1) Git Revision ID Extraction

In order to link the Gerrit review and code datasets, we extract the Git revision ID from the message that is automatically generated by the Qt CI bot. After a patch is accepted by reviewers, it is merged into the `release` branch and the bot adds a message to the review discussion of the form: *"Change has been successfully cherry-picked as <Git Revision ID>"*.

(DP2) Defective File Identification

To identify the future-defective and risky files, we count the number of post-release defects of each file. Similar to our prior work [12], we identify post-release defects using defect-fixing commits that are recorded during the six-month period after the release date.

Figure 3 illustrates our approach to identify the future-defective files for RQ1 and the risky files for RQ2. We classify the files that have post-release defects as *future-defective files*. Files that do not have post-release defect are classified as *clean files*. Similarly, we classify the files that had post-release defects in the prior release as *risky files*, while *normal files* are files that did not have post-release defects in the prior release.

(DP3) Review Linking

To find the reviews that are associated with changes made to the studied files, we first link the commits to reviews using the Git revision ID. We only study the reviews of changes that can be linked between code and review datasets. We then produce a review database by selecting pre-release reviews of the studied files. We select reviews of future-defective (or risky) files from those reviews that are associated with at least one future-defective (or risky) file. The reviews that are not associated with any future-defective (or risky) files are linked to clean (or normal) files.

We conservatively link the reviews that are associated with both future-defective (or risky) files and clean (or normal) files to future-defective (or risky) files, when these reviews could have been linked to clean (or normal) files. We do so because we feel that the worst-case scenario where we mistakenly link some review activity to future-defective (or risky) files is more acceptable than the worst-case scenario where we mistakenly link some review activity to clean (or normal) files.

C. Data Analysis

To address our research questions, we perform quantitative (DA1) and qualitative (DA2) analyses. Figure 2 provides an overview of our analyses. We describe each analysis below.

(DA1) Quantitative Analysis of Code Review Activity

We study the differences in code review activity between future-defective (or risky) and clean (or normal) files. To do so, we calculate several code review activity metrics and measure the difference in code review activity metrics using a statistical approach. We describe each step in this process below.

(DA1-a) Review activity metrics calculation. Table II provides an overview of the 11 metrics that we use to measure code review activity. Since code review activity is often correlated with patch size [24–26], we normalize each raw code review activity metric by the patch size. Our metrics are grouped into three dimensions: 1) *Review Intensity* measures the scrutiny that was applied during the code review process, 2) *Review Participation* measures how much the development team invests in the code review process, and 3) *Reviewing Time* measures the duration of a code review.

(DA1-b) Review activity analysis. We use a statistical approach to determine whether code review activity in future-defective (or risky) files is significantly different from code review activity in clean (or normal) files.

To statistically confirm the difference of the code review activity in future-defective (or risky) and clean (or normal) files, we first test for normality in our data using the Shapiro-Wilk test ($\alpha = 0.05$). We observe that the distributions of code

TABLE II
A TAXONOMY OF THE CODE REVIEW ACTIVITY METRICS. THE METRICS NORMALIZED BY PATCH SIZE ARE MARKED WITH A DAGGER SYMBOL (†).

	Metric	Description	Rationale
Review Intensity	Number of Iterations†	Number of review iterations for a patch prior to its integration.	Fixing a defect found in each round of multiple iterations of a review would reduce the number of defects more than a single iteration of review [15].
	Discussion Length†	Number of general comments and inline comments written by reviewers.	Reviewing proposed changes with a long discussion would find more defects and provide a better solution [10, 16].
	Proportion of Revisions without Feedback	Proportion of iterations that are not inspired by a reviewer neither posting a message nor a score.	Although a code review of MCR can be done by bots, the suggestion can be either superficial or false positives [17]. More revisions that are manually examined by reviewers would lead to a lower likelihood of having future defects.
	Churn during Code Review†	Number of lines that were added and deleted between revisions.	More lines of codes that were revised during code review would lead to a lower likelihood of having a future defect [18].
Review Participation	Number of Reviewers†	Number of developers who participate in a code review, i.e., posting a general comment, or inline comment, and assigning a review score.	Changes examined by many developers are less likely to have future defects [9, 19].
	Number of Authors†	Number of developers who upload a revision for proposed changes.	Changes revised by many authors may be more defective [13, 20].
	Number of Non-Author Voters†	Number of developers who assign a review score, excluding the patch author.	Changes that receive a review score from the author may have essentially not been reviewed [12].
	Proportion of Review Disagreement	A proportion of reviewers that vote for a disagreement to accept the patch, i.e., assigning a negative review score.	A review with a high rate of acceptance discrepancy may induce a future fix.
Reviewing Time	Review Length†	Time in days from the first patch submission to the reviewers acceptance for integration.	The longer time of code review, the more defects would be found and fixed [9, 15].
	Response Delay	Time in days from the first patch submission to the posting of the first reviewer message.	Reviewing a patch promptly when it is submitted would reduce the likelihood that a defect will become embedded [9].
	Average Review Rate	Average review rate (KLOC/Hour) for each revision.	A review with a fast review rate may lead the changes that are defective [21–23].

TABLE III
A CONTINGENCY TABLE OF A CODE REVIEW ACTIVITY METRIC (m), WHERE a AND c REPRESENT THE NUMBER OF REVIEWS OF DEFECTIVE FILES, AND b AND d REPRESENT THE NUMBER OF REVIEWS OF THEIR CLEAN COUNTERPARTS.

	Low Metric Value $m \leq \text{median}_m$	High Metric Value $m > \text{median}_m$
Have defects	a	c
No defects	b	d

review activity metrics do not follow a normal distribution ($p < 2.2 \times 10^{-16}$ for all of the metrics). Thus, we use a non-parametric test, i.e., the one-tailed Mann-Whitney U test to check for significant differences in the code review activity metrics of future-defective (or risky) files and clean (or normal) files ($\alpha = 0.05$).

We also measure the relative impact in order to understand the magnitude of the relationship. We estimate the relative impact using the odds ratio [27]. We compare the odds of future-defective (or risky) files that undergo reviews with high metric values (greater than the median) and reviews with low metric values (less than or equal to the median). From the contingency table constructed for a code review activity metric (m) as shown in Table III, we can measure the relative impact using a calculation of $\text{imp}(m) = \frac{(c/d)-(a/b)}{(a/b)}$. A positive relative impact indicates that a shift from low metric values to high metric values is accompanied by an increase in the likelihood of future (or past) defect proneness, whereas a negative relative impact indicates a decrease in that likelihood.

(DA2) Qualitative Analysis of Concerns Addressed during Code Review

We compare the concerns that were addressed during code review of future-defective (or risky) files and clean (or normal) files. Similar to Beller *et al.* [28], we identify the concerns by manually labelling the types of changes made to a patch between revisions. We describe each step below.

(DA2-a) Representative sample selection. As the full set of review data is too large to manually examine in its entirety, we randomly select a statistically representative sample for our analysis. To select a representative sample, we determine the sample size using a calculation of $s = \frac{z^2 p(1-p)}{c^2}$, where p is the proportion that we want to estimate, $z = 1.96$ to achieve a 95% confidence level, and $c = 0.1$ for 10% bounds of the actual proportion [29, 30]. Since we did not know the proportion in advance, we use $p = 0.5$. We further correct the sample size for the finite population of reviews P using $ss = \frac{s}{1+\frac{s-1}{P}}$. Since we consider only changes that occur during code review, we randomly select the representative sample from those reviews that have at least two revisions.

(DA2-b) Addressed concerns identification. To identify the concerns that were addressed during code review, we manually label the changes that occurred between revisions using the file-by-file comparison view of the Gerrit system. Each change is labelled as being inspired by reviewer feedback or not, as well as the corresponding type. For the type of a change, we use the change classification defined by Mäntylä and Lassenius [31], with the addition of the traceability category. For each type of change, we count how many reviews make such changes. Since many changes can be made during a

TABLE IV
AN OVERVIEW OF THE REVIEW DATABASE OF RQ1.

	Qt 5.0.0		Qt 5.1.0	
	Future-defective	**Clean**	**Future-defective**	**Clean**
Studied Files	1,176	10,513	866	11,931
Related Reviews	3,470	2,727	2,849	2,690
Review Sample	93 (405 revisions)	93 (344 revisions)	93 (371 revisions)	93 (342 revisions)

review, the sum of the frequencies of each type can be larger than the total number of reviews. Below, we briefly describe each type in our change classification schema.

Evolvability refers to changes made to ease the future maintenance of the code. This change type is composed of three sub-types: 1) *Documentation* refers to changes in parts of the source code that describe the intent of the code, e.g., identifier names or code comments, 2) *Structure* refers to code organization, e.g., refactoring large functions, and 3) *Visual representation* refers to changes that improve code readability, e.g., code indentation or blank line usage.

Functionality refers to changes that impact the functionality of the system. This change type is composed of six sub-types: 1) *Larger defects* refer to changes that add missing functionality or correct implementation errors, 2) *Check* refers to validation mistakes, or mistakes made when detecting an invalid value, 3) *Resource* refers to mistakes made with data initialization or manipulation, 4) *Interface* refers to mistakes made when interacting with other parts of the software, such as other internal functions or external libraries, 5) *Logic* refers to computation mistakes, e.g., incorrect comparison operators or control flow, and 6) *Support* refers to mistakes made with respect to system configuration or unit testing.

Traceability refers to bookkeeping changes for VCSs. We add this type into our classification schema because the software development of large software systems is also concerned with the management of source code repositories [32, 33]. For instance, developers should describe the proposed change using a detailed commit message, and the proposed change must be applied to the latest version of the codebase.[1]

IV. CASE STUDY RESULTS

In this section, we present the results of our case study with respect to our two research questions. For each research question, we present and discuss the results of quantitative (DA1) and qualitative (DA2) analyses.

(RQ1) Do developers follow lax code review practices in files that will eventually have defects?

Table IV provides an overview of the review database that we construct to address RQ1. We conjecture that files that are intensely scrutinized, with more team participation, that are reviewed for a longer time, and often address functionality concerns are less likely to have defects in the future.

We now present our empirical observations, followed by our general conclusion.

[1]http://qt-project.org/wiki/Gerrit-Introduction

TABLE V
RESULTS OF ONE-TAILED MANN-WHITNEY U TESTS ($\alpha = 0.05$) FOR CODE REVIEW ACTIVITY METRICS OF FUTURE-DEFECTIVE (FD) AND CLEAN (C) FILES.

Metric	Statistical Test		Relative Impact (%)	
	Qt 5.0.0	Qt 5.1.0	Qt 5.0.0	Qt 5.1.0
Review Intensity				
#Iterations†	FD < C***	FD < C***	-24 ↓	-20 ↓
Discussion Length†	FD < C***	FD < C**	-19 ↓	-14 ↓
Revisions without Feedback	FD > C***	FD > C*	7 ↑	5 ↑
Churn during Code Review†	**FD > C** ***	**FD > C*****	11 ↑	11 ↑
Review Participation				
#Reviewers†	FD < C***	FD < C***	-26 ↓	-20 ↓
#Authors†	**FD < C*****	**FD < C*****	-33 ↓	-27 ↓
#Non-Author Voters†	FD < C***	FD < C***	-30 ↓	-26 ↓
Review Disagreement	FD > C*	FD > C*	3 ↑	4 ↑
Reviewing Time				
Review Length†	-	FD > C***	-	18% ↑
Response Delay	-	FD > C*	-	11% ↑
Average Review Rate	FD > C***	FD > C***	42% ↑	16% ↑

† The metrics are normalized by patch size.
Statistical significance: * $p < 0.05$, ** $p < 0.01$, *** $p < 0.001$.

(RQ1-DA1) Quantitative Analysis of Code Review Activity

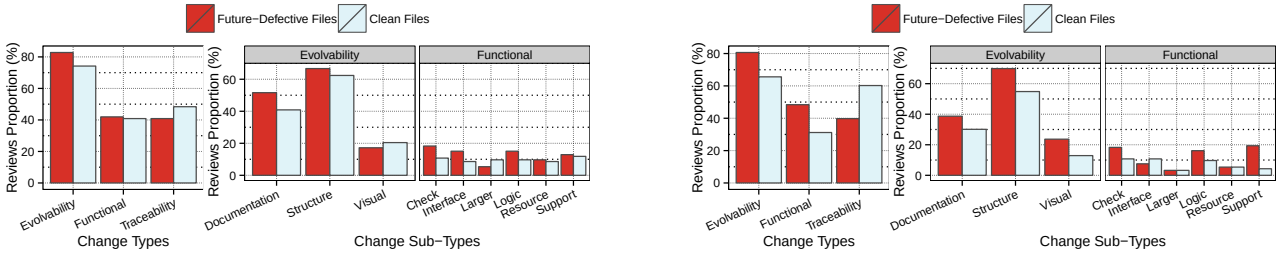
Observation 1 – Future-defective files tend to undergo less intense code review than clean files do. As we suspected, Table V shows that future-defective files tend to undergo reviews that have fewer iterations, shorter discussions, and have more revisions that are not inspired by reviewer feedback than clean files do. Mann-Whitney U tests confirm that the differences are statistically significant ($p < 0.001$ for all of the review intensity metrics).

However, future-defective files tend to change more during code review than clean files. Table V shows that the churn during code review of future-defective files is higher than that of clean files ($p < 0.001$). This may be because changes made to future-defective files tend to be more controversial. For example, review ID 27977[2] proposes a controversial change that reviewers disagreed with, asking the author to revise the proposed changes many times before reluctantly allowing the integration of the changes into the upstream VCSs.

Observation 2 – Future-defective files tend to undergo reviews with less team participation than clean files do. Table V shows that future-defective files tend to be reviewed by fewer reviewers, involve fewer non-author voters, and have a higher rate of review disagreement than clean files do. Mann-Whitney U tests confirm that the differences are statistically significant ($p < 0.001$ for the number of reviewers and the non-author voters metrics, and $p < 0.05$ for the proportion of review disagreement metric).

Furthermore, Table V shows that future-defective files tend to have less authors who upload revisions than clean files do ($p < 0.001$). We observe that additional authors often help the original author to improve the proposed changes. For example, in review ID 35360,[3] the additional author updated the proposed changes to be consistent with his changes from another patch. This suggests that the proposed change can be improved and complex integration issues can be avoided when it is revised by many developers during code review.

[2]https://codereview.qt-project.org/#/c/27977
[3]https://codereview.qt-project.org/#/c/35360



(a) Qt 5.0.0 (b) Qt 5.1.0

Fig. 4. Distribution of change types that occurred during the code review of future-defective and clean files. The sum of review proportion is higher than 100%, since a review can contain many types of changes.

Observation 3 – Future-defective files tend to undergo reviews with a faster rate of code examination than clean files do. Table V shows that future-defective files tend to be reviewed with a faster average review rate than clean files. A Mann-Whitney U test confirms that the difference is statistically significant ($p < 0.001$). We also observe that in Qt 5.1.0, the future-defective files tend to receive longer response delay than the clean files do, while we cannot confirm the statistical significance of the difference in Qt 5.0.0.

Surprisingly, Table V shows that the review length of future-defective files tends to be longer than clean files in Qt 5.1.0 ($p < 0.001$). We observe that some of the reviews take a longer time due to a lack of reviewer attention. For example, in review ID 32926,[4] there is little prompt discussion about a proposed change. Since the patch already received a review score of +1 from a reviewer and there were no other comments after waiting for 9 days, the author presumed that the change was clean and self-approved his own change. This lack of reviewer attention may in part be due to long review queues [34]. This finding complements observation 2 — files that are reviewed with little team participation tend to have future defects.

Table V shows that the metrics with the largest relative impacts are the average review rate for Qt 5.0.0 (42%) and the number of authors for Qt 5.1.0 (-27%), while the proportion of review disagreement metric has the smallest relative impact for both Qt versions (an average of 3.5%). Furthermore, the number of reviewers, authors, and non-author voters also have a large relative impact, which ranges between -33% and -20%.

(RQ1-DA2) Qualitative Analysis of Concerns Addressed during Code Review

Observation 4 – For evolvability changes, future-defective files tend to more frequently address documentation and structure concerns than clean files do. Figure 4 shows that the proportion of reviews in future-defective files that make documentation and structure changes is higher than the corresponding proportion in the clean files. There are differences of 11 and 9 percentage points (52%−41% and 39%−30%) in documentation changes, and 5 and 15 percentage points (67%−62% and 70%−55%) in structure changes for Qt 5.0.0 and Qt 5.1.0, respectively. We observe that the documentation changes involve updating copyright terms, expanding code comments, and renaming identifiers. The structure changes relate to removing dead code and reusing existing functions rather than implementing new ones. Moreover, we find that the documentation changes were inspired by reviewers in future-defective files (21%) more often than clean files (10%), indicating that reviewers often focus on documentation issues in future-defective files.

Furthermore, we find that evolvability is the most frequently addressed concern during code review of both clean and future-defective files. Figure 4 shows that, similar to prior work [28, 31], evolvability changes account for the majority of changes in the reviews, i.e., 82% of the reviews of future-defective files and 70% of the reviews of clean files on average.

Observation 5 – For functionality changes, future-defective files tend to more frequently address check, logic, and support concerns than clean files do. Figure 4 shows that the proportion of reviews in future-defective files that make check, logic, and support changes is higher than the corresponding proportion in the clean files. There are differences of 7 and 6 percentage points (18%−11% and 16%−10%) in check and logic changes, respectively. We observe that many check changes involve validating variable declarations. The logic changes mainly relate to changing comparison expressions. For the support changes, the proportion of reviews in Qt 5.1.0 shows a clear difference of 15 percentage points (19%−4%) between future-defective and clean files. Moreover, in Qt 5.1.0, the support changes are addressed by the author in future-defective files more often than clean files. The proportion of reviews is 15% and 2% in future-defective files and clean files, respectively. On the other hand, there are few reviews where reviewers inspire functionality changes. The proportion of reviews ranges between 1% - 9% (average of 5%) in future-defective files and between 1% - 6% (average of 4%) in clean files.

We also find that the reviews of clean files tend to more frequently address traceability concerns than the reviews of future-defective files do. Figure 4 shows that the proportion of reviews that address traceability concern in clean files is higher than the corresponding proportion in the future-defective files with the differences of 7 and 21 percentage points (48%−41% and 60%−39%) for Qt 5.0.0 and Qt 5.1.0, respectively.

[4]https://codereview.qt-project.org/#/c/32926

TABLE VI
AN OVERVIEW OF THE REVIEW DATABASE TO ADDRESS RQ2.

	Qt 5.1.0			
	Risky	**Normal**	**Risky & Future-Defective**	**Risky & Clean**
Studied Files	1,168	11,629	206	962
Related Reviews	2,671	2,868	1,299	1,372
Review Sample	93 (399 revisions)	93 (309 revisions)	44 (205 revisions)	49 (194 revisions)

The reviews of files that will eventually have defects tend to be less rigorous and more frequently address evolvability concerns than the reviews of files without future defects do.

(RQ2) Do developers adapt their code review practices in files that have historically been defective?

To address RQ2, we use post-release defects of Qt 5.0.0 as prior defects for Qt 5.1.0 and investigate the review activity of changed files in Qt 5.1.0. Table VI shows an overview of the review database that we use to address RQ2. We conjecture that reviews of risky files should be more intensely scrutinized, have more team participation, and take a longer time to complete than the reviews of normal files.

We now present our empirical observations, followed by our general conclusion.

(RQ2-DA1) Quantitative Analysis of Code Review Activity

Observation 6 – Risky files tend to undergo less intense code reviews than normal files do. Table VII shows that risky files tend to undergo reviews that have fewer iterations, shorter discussions, and more revisions without reviewer feedback than normal files do. Mann-Whitney U tests confirm that the differences are statistically significant ($p < 0.001$ for the number of iterations, discussion length metrics, and $p < 0.01$ for the proportion of revisions without feedback metric).

Table VII also shows that the reviews of risky files tend to churn more during code review than those in normal files. Similar to observation 1, changes with more churn during code review are likely to be controversial. For example, in review ID 29929,[5] reviewers provide many suggested fixes and the author needs to revise the proposed changes many times before the change was accepted for integration.

Observation 7 – Risky files tend to be reviewed with less team participation than normal files. Table VII shows that risky files tend to be reviewed by fewer reviewers and non-author voters than normal files do. Mann-Whitney U tests confirm that the differences are statistically significant ($p < 0.001$ for the number of reviewers and the number of non-author voters metrics). Moreover, there tend to be fewer authors in the reviews of risky files than those reviews of normal files. The low number of authors in risky files is also worrisome, since observation 2 suggests that multiple authors revising the proposed change in a review tend to avoid problems that could lead to future defects.

Observation 8 – Risky files tend to undergo reviews that receive feedback more slowly and have faster review rate than normal files. Although Table VII shows that risky files tend to undergo reviews that have a longer review length than normal files do, its relative impact is only 6%. On the other hand, we find that the reviews of risky files have a longer response delay and faster review rate than the reviews of normal files. Mann-Whitney U tests confirm that the differences are statistically significant ($p < 0.001$ for the response delay and the average review rate metrics). This result is also worrisome, since observation 3 suggests that that files that undergo reviews with longer response delay and faster review rate are likely to have defects in the future.

[5]https://codereview.qt-project.org/#/c/29929

TABLE VII
RESULTS OF ONE-TAILED MANN-WHITNEY U TESTS ($\alpha = 0.05$) FOR CODE REVIEW ACTIVITY METRICS OF RISKY AND NORMAL FILES.

Metric	Statistical Test	Relative Impact (%)
Review Intensity		
#Iterations†	Risky < Normal***	-28 ↓
Discussion Length†	Risky < Normal***	-26 ↓
Revisions without Feedback	Risky > Normal**	6 ↑
Churn during Code Review†	Risky > Normal***	7 ↑
Review Participation		
#Reviewers†	Risky < Normal***	-32 ↓
#Authors†	Risky < Normal***	-34 ↓
Non-Author Voters†	Risky < Normal***	-33 ↓
Review Disagreement	-	-
Reviewing Time		
Review Length†	Risky > Normal*	6 ↑
Response Delay	Risky > Normal***	15 ↑
Average Review Rate	Risky > Normal***	17 ↑

† The metrics are normalized by patch size.
Statistical significance: * $p < 0.05$, ** $p < 0.01$, *** $p < 0.001$.

Table VII shows that the metric with the largest relative impact is the number of authors (-34%), while the proportion of revisions without feedback and the review length metrics have the smallest relative impact (6%). Furthermore, we find that the number of iterations, the discussion length, the number of reviewers, and non-author voters metrics also have large relative impacts, ranging between -33% and -26%.

(RQ2-DA2) Qualitative Analysis of Concerns Raised during Code Review

Observation 9 – Risky files tend to more frequently have evolvability concerns addressed during code review than normal files do. Figure 5 shows that evolvability concerns are addressed in the reviews of risky files more often than normal files with a proportion of reviews that is 29 percentage points higher (87%−58%). The proportion of reviews that make structure changes shows an obvious difference of 32 percentage points (78%−46%) in risky and normal files. We observe that structure changes in the reviews of risky files relate to removing dead code, while changes in the reviews of normal files relate to re-implementing solutions using alternative approaches and small fixes for runtime errors.

Observation 10 – Risky files tend to more frequently have functionality concerns addressed during code review than normal files do. Figure 5 shows that there is a difference of 14 percentage points (47%−33%) between risky and normal files. The proportion of reviews that make check and logic

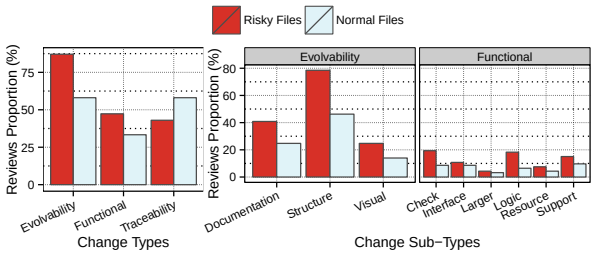


Fig. 5. Distribution of change types that occurred during the code review of risky and normal files. The sum of review proportion is higher than 100%, since a review can contain many types of changes.

changes shows a clear difference of 11 percentage points (19%−8%) between risky and normal files. We observe that the check changes relate to validating variable values, and logic changes relate to updates to comparison expressions.

On the other hand, Figure 5 shows that normal files tend to address traceability concerns more often than risky files do. There is a difference of 15 percentage points (58% - 43%) between risky and normal files. Additionally, this concern is often addressed by authors. Moreover, we observe that changes to normal files are more rebased than risky files are.

Developers are not as careful when they review changes made to files that have historically been defective and often address the concerns of evolvability and functionality.

We study this phenomenon further to investigate the relationship between the code review practices in the risky files and future defects. We use the same data analysis approaches, i.e., DA1 and DA2. We separate the risky files into two groups: 1) risky files that will eventually have defects (called *risky & future-defective files*) and 2) risky files that will eventually be defect-free (called *risky & clean files*) when Qt 5.1.0 is released. We also separate the randomly selected reviews of risky files to reviews of risky & future-defective files and reviews of risky & clean files in order to compare concerns addressed during code review. Table VI shows an overview of the review database that we use to perform this study.

Observation 11 – Risky & future-defective files tend to be less carefully reviewed than risky & clean files. Table VIII shows that risky & future-defective files tend to undergo less intense code review with less team participation, i.e., fewer iterations, shorter discussions, more revisions without reviewer feedback, fewer reviewers, authors, and non-author voters than risky & clean files. Mann-Whitney U tests confirm that the differences are statistically significant ($p < 0.001$ for all metrics of the review intensity and participation dimensions). Moreover, we find that risky & future-defective files tend to undergo reviews that have a longer response delay and faster review rate than risky & clean files. Mann-Whitney U tests confirm that the differences are statistically significant ($p < 0.001$ for the response delay and the average review rate metrics). These results indicate that changes made to the risky

TABLE VIII
RESULTS OF ONE-TAILED MANN-WHITNEY U TESTS ($\alpha = 0.05$) FOR CODE REVIEW ACTIVITY METRICS OF RISKY & FUTURE-DEFECTIVE FILES (R&FD) AND RISKY & CLEAN FILES (R&C).

Metric	Statistical Test	Relative Impact (%)
Review Intensity		
#Iterations†	R&FD < R&C***	-27 ↓
Discussion Length†	R&FD < R&C***	-29 ↓
Revisions without Feedback	R&FD < R&C***	10 ↑
Churn during Code Review†	R&FD > R&C*	7 ↑
Review Participation		
#Reviewers†	R&FD < R&C***	-27 ↓
#Authors†	R&FD < R&C***	-31 ↓
Non-Author Voters†	R&FD < R&C***	-30 ↓
Review Disagreement	-	-
Reviewing Time		
Review Length†	-	-
Response Delay	R&FD > R&C***	34 ↑
Average Review Rate	R&FD > R&C***	25 ↑

† The metrics are normalized by patch size.
Statistical significance: * $p < 0.05$, ** $p < 0.01$, *** $p < 0.001$.

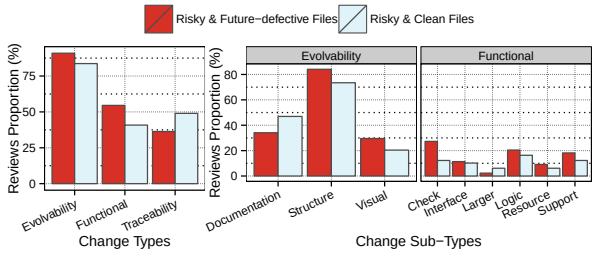


Fig. 6. Distribution of change types that occurred during the code review of risky & future-defective and risky & clean files. The sum of review proportion is higher than 100%, since a review can contain many types of changes.

files that are also defective are reviewed with less intensity, less team participation, faster review rate, and receive slower feedback than the risky files that are defect-free.

Table VIII shows that the metric with the largest relative impact is the response delay (34%). On the other hand, the churn during code review metric has the smallest relative impact (7%). We also find that the number of iterations, the discussion length, the number of reviewers, authors, and non-author voters metrics have a large negative relative impacts ranging between -31% to -27%, and the average review rate metric has a large positive relative impact (25%).

Observation 12 – Risky & future-defective files tend to have structure, visual representation, and check concerns addressed more often during code reviews than risky & clean files do. Figure 6 shows that for evolvability changes, the proportion of reviews in risky & future-defective files that make structure and visual representation changes is higher than the corresponding proportion in risky & clean files. There are differences of 11 and 10 percentage points (84%−73% and 30%−20%) in structure and visual representation changes, respectively. For functionality changes, the proportion of reviews that make check changes shows an obvious difference of 14 percentage points (55%−41%) between risky & future-defective files and risky & clean files. We also observe that reviewers inspire evolvability changes in risky & future-

defective files more often than risky & clean files. The proportion of reviews is 43% and 31% in risky & future-defective files and risky & clean files, respectively. However, few functionality changes are inspired by reviewers in the reviews of risky & future-defective files. The proportion of reviews ranges between 0% - 11% (average of 5%) in risky & future-defective files and between 4% - 10% (average of 5%) in risky & clean files for each type of functionality changes. This finding suggests that reviewers do not focus much on functionality during code review of risky files.

Files that have historically been defective and will eventually have defects tend to undergo less rigorous code reviews that more frequently address evolvability concerns than the files that have historically been defective, but will eventually be defect-free.

V. Discussion

In this section, we discuss the broader implications of our empirical observations.

Review intensity. Observations 1 and 6 have shown that the reviews of changes made to clean and normal files often have longer discussions and more iterations than the reviews of future-defective and risky files do. Prior work reports that the focus of MCR discussion has shifted from defect-hunting to group problem-solving [7, 8, 10]. Examining a patch in multiple review iterations would likely uncover more problems than a single review iteration [15]. Hence, the reviews that have long discussions and many iterations seem to improve the patch and avoid problems that could lead to future defects.

Review participation. Observations 2 and 7 have shown that the reviews of changes made to clean and normal files often have more participants than the reviews of future-defective and risky files do. Corresponding to Linus' law [19], our findings suggest that code reviews should be performed by multiple reviewers to reduce the likelihood of having future defects. Development teams should take the number of review participants into consideration when making integration decisions. For example, the MCR tools should be configured to require acceptance votes from multiple reviewers.

Review speed. Observations 3 and 8 have shown that each review iteration of clean and normal files is performed slower than future-defective and risky files. Similar to the traditional code review practices [22, 23], our findings suggest that reviewers will be able to uncover more problems hidden in a patch if they perform a careful code examination with an appropriate code reading rate.

Reviewing concerns. Observations 4, 5, 9, and 10 have shown that the reviews of changes made to future-defective files and risky files focus on evolvability concerns rather than functional fixes. Indeed, functionality concerns are still rarely addressed during code review. Although MCR practices seem to focus on improving maintainability, our prior findings suggest that the rigor of the reviewing process that is applied to a source code file throughout a development cycle could help development teams to avoid future defects.

Code review of risky files. Observations 11 and 12 have shown that risky & future-defective files tend to undergo less rigorous code reviews that more frequently address evolvability concerns than the risky & clean files. One contributing reason for the less careful review of risky files could be that it is difficult for practitioners to determine the risk of code changes during code review [35]. A supporting tool could help practitioners notice such risky files to perform code review more rigorously when necessary.

VI. Threats to Validity

We now discuss potential threats to validity of our study.

Construct validity. We study the code review practices of files derived from a set of reviews instead of studying on a single review because uncovering defects is not the sole intent of MCR practices [7] and the number of defects found during code review is not explicitly recorded in a MCR tool [8]. Therefore, we use the collection of code review activity of files during the development of a release to evaluate the impact that the MCR practices have on software quality.

The change classification method was conducted by the first author who are not involved in the code review process of the studied system. The results of manual classification by the team members might be different. However, our classification schema is derived from prior work [28, 31] and the comments that we use to classify code reviews are originally written by team members who participated in the code review process. Furthermore, we repeatedly label changes several times before perform our study to ensure the uniformity of the change classification, and a subset of change classification results is verified by the second author.

Internal validity. Some of our code review activity metrics are measured based on heuristics. For example, we assume that the review length is the elapsed time between when a patch has been uploaded and when it has been approved for integration. However, there are likely cases where reviewers actually examined a patch for a fraction of this review length. Unfortunately, reviewers do not record the time that they actually spent reviewing a patch. Since there is a limitation of measuring the actual code review activities, we must rely on heuristics to recover this information. Furthermore, our rationales for using metrics are supported by prior work [9, 10, 15, 16, 36].

There might be confounding factors that also impact software quality. For example, Meneely *et al.* find that novice reviewers are more likely to allow defects to seep through the code review process than experts [37]. On the other hand, only core developers (who are elected by the Qt development community) are given voting privileges in the Gerrit system.[6] Thus, each change has been examined by at least one expert before integration. Nevertheless, taking such factors into account may allow us to gain more insight into MCR practices in defective files.

[6]http://qt-project.org/wiki/The_Qt_Governance_Model

External validity. Although the Qt system is an open source project that actively assesses software changes through an MCR tool, the analysis of the studied dataset does not allow us to draw conclusions for all open source projects. Since the code review process of MCR is a relatively new development, finding systems that satisfy our selection criteria is a challenge (*cf.* Section III-A). Naggapan *et al.* also argue that if care is not taken when selecting which projects to analyze, then increasing the sample size does not actually contribute to the goal of increased generality [38]. Nonetheless, additional replication studies are needed to generalize our results.

VII. Related Work

In order to discuss the related work on code inspections and code reviews, we group them into those that perform quantitative studies and those that perform qualitative studies.

Quantitative studies. Understanding the code review process would help practitioners to perform code review more effectively. Porter *et al.* report that the number of reviewers and authors were significant sources of variation that impact software inspection performance [15]. Ferreira *et al.* find that in software inspection process, reading code changes with rate higher than 200 lines of code per hour leads to defects [22]. Rigby *et al.* report that in the broadcast-based code review process of the Apache project, a defect will become embedded in the software if proposed changes are not reviewed by the time it is submitted [9]. Our study arrives at similar conclusions, i.e., files that will have future defects tend to undergo reviews with fewer reviewers, a faster review rate, and receive late feedback.

Our prior work studies the relationship between lax participation in the MCR process and future defects in software components [12]. Our study aims to complement the prior work by examining the code review activity in different dimensions as well as observing concerns raised during the code review process of defective and clean files.

Besides observing the number of defects, many studies investigate factors that impact code review time and integration decisions. Jiang *et al.* study the relationship of patch characteristics and code review time in the Linux kernel [26]. Baysal *et al.* study the influence of the non-technical factors on the code review time and the proportion of accepted patches in the WebKit open source project [34]. Thongtanunam *et al.* investigate the impact of the reviewer assignment problem on the code review time of four open source projects [39]. Gousios *et al.* explore the relationship of code review participation and integration decisions in GitHub projects [11]. Inspired by these studies, we design our code review activity metrics to comparatively study code review practices in defective and clean files.

Qualitative studies. To better understand code review, several studies explore review discussions to uncover the addressed concerns during code review. Mäntylä and Lassenius observe a 75:25 ratio of maintainability-related and functional defects raised during the code reviews of student and industrial development projects [31]. Beller *et al.* found that a similar 75:25 ratio of issues were fixed during the MCR processes of two large systems [28]. Tao *et al.* report that reviewers of Eclipse and Mozilla projects are seriously concerned about inconsistent and misleading documentation of a patch over other problems [40]. Similar to code review practices in GitHub, Tsay *et al.* report that stakeholders are concerned about the appropriateness of a code solution and often provide alternative solutions during code review [10]. Complementing these studies, we study the difference of concerns addressed during the reviews of defective and clean files.

VIII. Conclusion

Although Modern Code Review (MCR) is now widely adopted in both open source and industrial projects, the impact that MCR practices have on software quality is still unclear. In this study, we comparatively study the MCR practices in defective and clean files, i.e., 1) files that will eventually have defects (called future-defective files), and 2) files that have historically been defective (called risky files). Due to the human-intensive nature of code reviewing, we decide to perform an in-depth study on a single system instead of examining a large number of projects. Using data collected from the Qt open source system, we empirically study 11,736 reviews of changes to 24,486 files and manually examine 558 reviews. The results of our study indicate that:

- The code review activity of future-defective files tends to be less intense with less team participation and with a faster rate of code examination than the reviews of clean files (Observations 1-3).
- Developers more often address concerns about: 1) documentation and structure to enhance evolvability, and 2) checks, logic, and support to fix functionality issues, in the reviews of future-defective files than the reviews of clean files (Observations 4-5).
- Despite their historically defective nature, the code review activity of risky files tends to be less intense with less team participation than files that have historically been defect-free. Reviews of risky files also tend to receive feedback more slowly and have a faster review rate than the reviews of normal files (Observations 6-8).
- In the reviews of risky files, developers address concerns about evolvability and functionality more often than the reviews of normal files do (Observations 9-10).
- Risky files that will have future defects tend to undergo less careful reviews that more often address concerns about evolvability than the reviews of risky files without future defects do (Observations 11-12).

Our results suggest that rigorous code review could lead to a reduced likelihood of future defects. Files that have historically been defective should be given more careful attention during code review, since such files are more likely to have future defects [13].

References

[1] B. Boehm and V. R. Basili, "Software Defect Reduction Top 10 List," *IEEE Computer*, vol. 34, no. 1, pp. 135–137, 2001.

[2] F. Shull, V. Basili, B. Boehm, A. W. Brown, P. Costa, M. Lindvall, D. Port, I. Rus, R. Tesoriero, and M. Zelkowitz, "What We Have Learned About Fighting Defects," in *Proceedings of the 8th International Software Metrics Symposium (METRICS)*, 2002, pp. 249–258.

[3] M. E. Fagan, "Design and Code Inspections to Reduce Errors in Program Development," *IBM System Journal*, vol. 15, no. 3, pp. 182–221, 1976.

[4] A. F. Ackerman, L. S. Buchwald, and F. H. Lewski, "Software Inspections: An Effective Verification Process," *IEEE Software*, vol. 6, no. 3, pp. 31–36, 1989.

[5] A. Aurum, H. Petersson, and C. Wohlin, "State-of-the-Art: Software Inspections After 25 Years," *Software Testing, Verification and Reliability*, vol. 12, no. 3, pp. 133–154, Sep. 2002.

[6] L. G. Votta, "Does Every Inspection Need a Meeting?" in *Proceedings of the 1st International Symposium on the Foundations of Software Engineering (FSE)*, 1993, pp. 107–114.

[7] A. Bacchelli and C. Bird, "Expectations, Outcomes, and Challenges of Modern Code Review," in *Proceedings of the 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 712–721.

[8] P. C. Rigby and C. Bird, "Convergent Contemporary Software Peer Review Practices," in *Proceedings of the 9th joint meeting of the European Software Engineering Conference and the International Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2013, pp. 202–212.

[9] P. C. Rigby, D. M. German, and M.-A. Storey, "Open Source Software Peer Review Practices: A Case Study of the Apache Server," in *Proceedings of the 30th International Conference on Software Engineering (ICSE)*, 2008, pp. 541–550.

[10] J. Tsay, L. Dabbish, and J. Herbsleb, "Let's Talk About It: Evaluating Contributions through Discussion in GitHub," in *Proceedings of the 22nd International Symposium on the Foundations of Software Engineering (FSE)*, 2014, pp. 144–154.

[11] G. Gousios, M. Pinzger, and A. van Deursen, "An Exploratory Study of the Pull-based Software Development Model," in *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, 2014, pp. 345–355.

[12] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan, "The Impact of Code Review Coverage and Code Review Participation on Software Quality," in *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR)*, 2014, pp. 192–201.

[13] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, "Predicting Fault Incidence Using Software Change History," *Transactions on Software Engineering (TSE)*, vol. 26, no. 7, pp. 653–661, 2000.

[14] K. Hamasaki, R. G. Kula, N. Yoshida, C. C. A. Erika, K. Fujiwara, and H. Iida, "Who does what during a Code Review? An extraction of an OSS Peer Review Repository," in *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR)*, 2013, pp. 49–52.

[15] A. Porter, H. Siy, A. Mockus, and L. Votta, "Understanding the Sources of Variation in Software Inspections," *Transactions On Software Engineering and Methodology (TOSEM)*, vol. 7, no. 1, pp. 41–79, 1998.

[16] R. Morales, S. McIntosh, and F. Khomh, "Do Code Review Practices Impact Design Quality? A Case Study of the Qt, VTK, and ITK Projects," in *Proceedings of the 22nd International Conference on Software ANalysis, Evolution, and Reengineering (SANER)*, 2015, pp. 171–180.

[17] S. Panichella, V. Arnaoudova, M. D. Penta, and G. Antoniol, "Would Static Analysis Tools Help Developers with Code Reviews?" in *Proceedings of the 22nd International Conference on Software ANalysis, Evolution, and Reengineering (SANER)*, 2015, pp. 161–170.

[18] A. Bosu, J. C. Carver, M. Hafiz, P. Hilley, and D. Janni, "Identifying the Characteristics of Vulnerable Code Changes: An Empirical Study," in *Proceedings of the 22nd International Symposium on the Foundations of Software Engineering (FSE)*, 2014, pp. 257–268.

[19] E. S. Raymond, "The Cathedral and the Bazaar," *Knowledge, Technology & Policy*, vol. 12, no. 3, pp. 23–49, 1999.

[20] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu, "Don't Touch My Code! Examining the Effects of Ownership on Software Quality," in *Proceedings of the 8th joint meeting of the European Software Engineering Conference and the International Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2011, pp. 4–14.

[21] G. W. Russell, "Experience with Inspection in Ultralarge-Scale Developments," *IEEE Software*, vol. 8, no. 1, pp. 25–31, 1991.

[22] A. L. Ferreira, R. J. Machado, L. Costa, J. G. Silva, R. F. Batista, and M. C. Paulk, "An Approach to Improving Software Inspections Performance," in *Proceedings of the 25th International Conference on Software Maintenance (ICSM)*, 2010, pp. 1–8.

[23] C. F. Kemerer and M. C. Paulk, "The Impact of Design and Code Reviews on Software Quality: An Empirical Study Based on PSP Data," *Transactions on Software Engineering (TSE)*, vol. 35, no. 4, pp. 1–17, 2009.

[24] J. Tsay, L. Dabbish, and J. Herbsleb, "Influence of Social and Technical Factors for Evaluating Contribution in GitHub," in *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, 2014, pp. 356–366.

[25] P. Weißgerber, D. Neu, and S. Diehl, "Small Patches Get

In!" in *Proceedings of the 5th Working Conference on Mining Software Repositories (MSR)*, 2008, pp. 67–75.

[26] Y. Jiang, B. Adams, and D. M. German, "Will My Patch Make It? And How Fast? Case Study on the Linux Kernel," in *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR)*, 2013, pp. 101–110.

[27] A. W. F. Edwards, "The Measure of Association in a 2 x 2 Table," *Journal of the Royal Statistical Society. Series A (General)*, vol. 126, no. 1, pp. 109–114, 1963.

[28] M. Beller, A. Bacchelli, A. Zaidman, and E. Juergens, "Modern Code Reviews in Open-Source Projects: Which Problems Do They Fix?" in *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR)*, 2014, pp. 202–211.

[29] S. K. Lwanga and S. Lemeshow, *Sample Size Determination in Health Studies: A Practical Manual*. Geneva: World Health Organization, 1991.

[30] L. Kish, *Survey sampling*. John Wiley and Sons, 1965.

[31] M. V. Mäntylä and C. Lassenius, "What Types of Defects Are Really Discovered in Code Reviews?" *Transactions on Software Engineering (TSE)*, vol. 35, no. 3, pp. 430–448, 2009.

[32] A. Hindle, D. M. German, M. W. Godfrey, and R. C. Holt, "Automatic Classification of Large Changes into Maintenance Categories," in *Proceedings of the 17th International Conference on Program Comprehension (ICPC)*, 2009, pp. 30–39.

[33] A. E. Hassan, "Automated Classification of Change Messages in Open Source Projects," in *Proceedings of the 23rd Symposium on Applied Computing (SAC)*, 2008, pp. 837–841.

[34] O. Baysal, O. Kononenko, R. Holmes, and M. W. Godfrey, "The Influence of Non-Technical Factors on Code Review," in *Proceedings of 20th Working Conference on Reverse Engineering (WCRE)*, 2013, pp. 122–131.

[35] Y. Tao, Y. Dang, T. Xie, D. Zhang, and S. Kim, "How Do Software Engineers Understand Code Changes?: An Exploratory Study in Industry," in *Proceedings of the 20th International Symposium on the Foundations of Software Engineering (FSE)*, 2012, pp. 51:1–51:11.

[36] A. A. Porter, H. P. Siy, C. A. Toman, and L. G. Votta, "An Experiment to Assess the Cost-Benefits of Code Inspections in Large Scale Software Development," *Transactions on Software Engineering (TSE)*, vol. 23, no. 6, pp. 329–346, 1997.

[37] A. Meneely, B. Spates, S. Trudeau, D. Neuberger, K. Whitlock, C. Ketant, and K. Davis, "An Empirical Investigation of Socio-technical Code Review Metrics and Security Vulnerabilities," in *Proceedings of the 6th International Workshop on Social Software Engineering (SSE)*, 2014, pp. 37–44.

[38] M. Nagappan, T. Zimmermann, and C. Bird, "Diversity in Software Engineering Research," in *Proceedings of the 9th joint meeting of the European Software Engineering Conference and the International Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2013, pp. 466–476.

[39] P. Thongtanunam, C. Tantithamthavorn, R. G. Kula, N. Yoshida, H. Iida, and K.-i. Matsumoto, "Who Should Review My Code? A File Location-Based Code-Reviewer Recommendation Approach for Modern Code Review," in *Proceedings of the 22nd International Conference on Software ANalysis, Evolution, and Reengineering (SANER)*, 2015, pp. 141–150.

[40] Y. Tao, D. Han, and S. Kim, "Writing Acceptable Patches: An Empirical Study of Open Source Project Patches," in *Proceedings of the 30th International Conference on Software Maintenance and Evolution (ICSME)*, 2014, pp. 271–280.