A Study of the Quality-Impacting Practices of Modern Code Review at Sony Mobile

Junji Shimagaki
Sony Mobile, Japan
Junji.Shimagaki@sonymobile.com

Yasutaka Kamei
Kyushu University, Japan
kamei@ait.kyushu-u.ac.jp

Shane McIntosh
McGill University, Canada
shane.mcintosh@mcgill.ca

Ahmed E. Hassan
Queen's University, Canada
ahmed@cs.queensu.ca

Naoyasu Ubayashi
Kyushu University, Japan
ubayashi@ait.kyushu-u.ac.jp

ABSTRACT

Nowadays, a flexible, lightweight variant of the code review process (i.e., the practice of having other team members critique software changes) is adopted by open source and proprietary software projects. While this flexibility is a blessing (e.g., enabling code reviews to span the globe), it does not mandate minimum review quality criteria like the formal code inspections of the past. Recent work shows that lax reviewing can impact the quality of open source systems. In this paper, we investigate the impact that code reviewing practices have on the quality of a proprietary system that is developed by Sony Mobile. We begin by replicating open source analyses of the relationship between software quality (as approximated by post-release defect-proneness) and: (1) code review coverage, i.e., the proportion of code changes that have been reviewed and (2) code review participation, i.e., the degree of reviewer involvement in the code review process. We also perform a qualitative analysis, with a survey of 93 stakeholders, semi-structured interviews with 15 stakeholders, and a follow-up survey of 25 senior engineers. Our results indicate that while past measures of review coverage and participation do not share a relationship with defect-proneness at Sony Mobile, reviewing measures that are aware of the Sony Mobile development context are associated with defect-proneness. Our results have lead to improvements of the Sony Mobile code review process.

CCS Concepts

•**Software and its engineering** → **Software testing and debugging;** *Software defect analysis;* •**General and reference** → *Empirical studies; Metrics;*

Keywords

Code review, software quality

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

DOI: http://dx.doi.org/10.1145/2889160.2889243

1. INTRODUCTION

Code review is recognized as an effective strategy to discover and fix software defects before a set of proposed code changes are integrated into the codebase. In 1976, Fagan [9] formalized the code inspection process, which mandates that reviewers follow checklists and participate in group meetings with the author and other stakeholders. Although code inspections have been shown to be effective at detecting errors during requirements analysis, design, and implementation [32], its rigid nature makes it difficult to adopt in today's globally-distributed, rapidly releasing software projects [31].

Unlike the formal code inspections of the past, modern code review is lightweight and flexible. Broadcast-based peer review proceeds asynchronously and is broadly adopted by Open Source Software (OSS) projects, e.g., Apache [26], that welcome contributions from developers that span the globe. Recent advances in tool support for code review (e.g., Gerrit[1]) have enabled tighter integration of code review with version control and issue tracking systems [25].

This flexibility of modern code review is both a blessing and a curse. On the one hand, modern code reviewing processes can easily scale out to support globally distributed software teams. On the other hand, modern code reviews do not mandate review checklists or in-person meetings, which guaranteed a base level of reviewer participation in the code inspection process of the past. Indeed, recent work shows that lax code review practices can impact software quality in large OSS projects. Reviewer involvement is known to share a relationship with software code quality [17, 18] and software design quality [21] in four large OSS projects. Moreover, Thongtanunam *et al.* [30] find that reviewers tend to be less careful in the files that will eventually have defects.

Yet, little is known about the impact that lax reviewing practices may have on systems that are developed in a proprietary setting. There are several differences in software project characteristics between OSS and proprietary projects that may affect the prior findings. For example, unlike the global and asynchronous development of OSS teams, proprietary software teams are often colocated. Herbleb and Grinter [13] found that these colocated development teams of proprietary software projects often use face-to-face (offline) communication to make project decisions. Aranda and Venolia [2] have shown that the software repositories of large proprietary software teams often omit (or contain erroneous)

[1] https://www.gerritcodereview.com/

Table 1: An overview of the Sony Mobile project in comparison to the previously studied projects [18].

	Our Project	Qt v5.1	VTK 5.10
# Commits	$\simeq$ 20,000	7,106	1,431
# Authors	$\simeq$ 1,000	422	55
# Components	500 $\sim$ 1,500	1,337	170
# Reviewers	$\simeq$ 1,000	348	45
Review coverage	81%	96%	39%

collaboration information because of the face-to-face nature of colocated collaboration.

In this paper, we re-examine McIntosh *et al.*'s prior study [18] in a proprietary setting at Sony Mobile. In addition to replicating the quantitative analysis of the prior work, we perform an extensive qualitative analysis, which includes: (a) a survey of 93 stakeholders at Sony Mobile, (b) semi-structured interviews with 15 of these stakeholders, and (c) a follow-up survey of 25 senior software engineers. The central question of our qualitative analysis is: *"Why are certain reviewing practices associated with better software quality?"* Triangulation of statistical analysis with stakeholder intuition shows that the degree of reviewer involvement does indeed have an impact on software quality.

This paper makes the following contributions:

- Identifying quality-impacting review practices in a proprietary development setting.
- An empirically grounded improvement plan for the code reviewing practices at Sony Mobile.

Paper organization. In Section 2, we introduce the code integration process at Sony Mobile. In Sections 3 and 4, we revisit the relationship between the degree of code reviewer involvement and software quality. In Section 5, we present our survey of 93 stakeholders. Section 6 triangulates our findings with those of related work to generate recommendations about quality-impacting code review practices. Section 7 discloses the threats to the validity of our study. Finally, Section 8 draws conclusions.

2. CODE INTEGRATION AT SONY MOBILE

In this section, we explain the characteristics of the studied project, the Gerrit code review tools, and how code changes are integrated into the software product.

2.1 Studied Project

Table 1 provides an overview of the studied project. The project is an embedded system under development that is derived from the Android codebase with built-in original apps. The system runs on a smartphone device that is commercially released. The system is developed by a network of colocated teams in Asia, Europe, and North America.

The system consists of two types of components: those that are developed in-house (e.g., Sony Mobile apps or Android extensions) and those that originate from external codebases (e.g., Android[2] and Qualcomm[3]). Roughly 30% of the components of the studied project are developed externally. Hence, a portion of the development effort at Sony Mobile is dedicated to the integration of changes in those external components while active development of Sony Mobile components is underway.

[2]https://source.android.com/

[3]http://codeaurora.org/

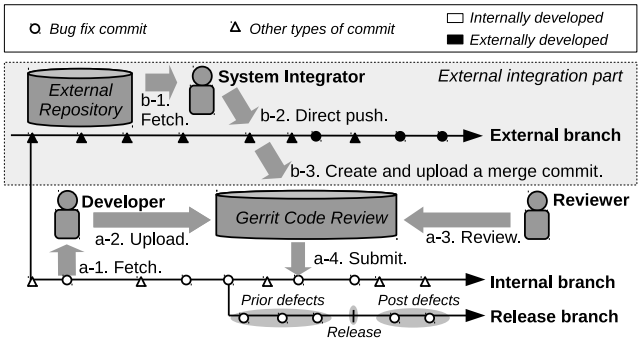

Figure 1: The Sony Mobile code integration process.

Furthermore, unlike typical OSS projects where contributors are globally distributed, the studied project is developed by a network of colocated teams in Asia, Europe, and North America. The colocated nature of much of the development allows team members to interact with one another in-person to a much larger extent than many OSS teams.

2.2 Code Integration Processes

At Sony Mobile, internally developed code must be critiqued by other team members using the Gerrit code review tool. Gerrit is a web-based code review tool that is broadly adopted by OSS and proprietary projects.[4]

Figure 1 provides an overview of the Gerrit-enabled code integration process at Sony Mobile. At Sony Mobile, there are integration processes for: (a) internal branches, (b) external branches, and (c) official releases. We describe each of the integration processes below.

a. Internal integration process. Developers fetch the latest code from an internal branch (a-1), modify the code, and make necessary local commits. Local commits must be uploaded for review using Gerrit (a-2), where other developers review the changes. To obtain submission privileges (a-3), reviewers must provide positive review and verification scores. If a negative score is given, the author must address the feedback of the reviewers by creating a new commit and requesting a re-review. After a commit has been granted submission privileges, the Gerrit system allows the author to submit the commit to the main project repository (a-4).

b. External integration process. Unlike internal integration, external integration is handled by system integrators. The system integrators have the permission to execute the *direct push* operation,[5] which is designed to bypass the code review process. At Sony Mobile, this direct push operation is only available to senior team members.

To perform a direct push, a system integrator first downloads the latest code from an external repository to their personal workspace (b-1). Then, the integrator pushes the latest code to the external branch in the internal code repository (b-2). Next, system integrators upload merge commits to the Gerrit code review process (b-3). These merge commits integrate the externally developed code into the internal Sony Mobile branch. Note that while merge commits are carefully reviewed to check for conflicts between internal and external branches, cleanly merging external code churn

[4]http://blogs.collab.net/git/why-gerrit-is-important-for-enterprise-git

[5]https://gerrit-review.googlesource.com/Documentation/access-control.html#category_push_direct

is often overlooked. Finally, merge commits can be submitted after submission privileges are granted (a-4).

c. Release integration process. As development proceeds, a so-called *release branch* is created (see Figure 1). In addition to the code review process that is applied to internal code, these release branches are more strictly monitored than development and only urgent fixes are granted submission privileges on these branches.

3. REPLICATION STUDY SETUP

We quantitatively analyze the historical code changes and code review data of a large scale commercial project. We do so by re-examining two research questions originally posed by McIntosh *et al.* [18] regarding code review practices:

RQ1 Is there a relationship between code review coverage and post-release defects?

RQ2 Is there a relationship between code review participation and post-release defects?

Similar to the prior study [18], we address these RQs by analyzing components of the studied system. To identify components in the Sony Mobile environment, we adopt the modular programming concept suggested by Parnas [24]: *"...it should be possible to make drastic changes to one module without a need to change others. ..."*. We use the commit activity of personnel who belong to a team, and the directory structure of the Sony Mobile system to identify components. We briefly describe each classification type below.

Role-based: Components are classified by connecting the team personnel data that is recorded in human resources databases with the commit activity data that is recorded in the main project repository. This connected data maps every Git commit to a team. Those commits that are recorded by a team are considered to be impacting one component.

Directory-based: Those commits that cannot be connected to the activity of a team are split into components using the directory structure of the system. These components are defined using the top-level directory name. A similar method was applied in the prior work [18].

Next, we define the software metrics that we use to quantify code review practices. Then, we describe our statistical approach to construct and analyze regression models that explain the incidence of post-release defects.

3.1 Software Metrics

In this paper, we study the relationship between several potential quality-impacting metrics and the incidence of post-release defects. Table 2 provides an overview of the studied metrics. We define each metric below.

Defects. Similar to several studies [6, 18, 23], we discover incidences of defects by scanning corresponding fixes for defects. We focus on defect fixing activity that occurs (or is merged into) the release branch. *Prior defects* are defects that were fixed before the product was released, while *post-release defects* are defects that were discovered in the field and fixed as part of a software update (see Figure 1).

Baseline metrics. In the literature, several metrics have been shown to share a relationship with software quality. To control for those confounding factors, we include them as baseline metrics. Size and Complexity are measured by reading the source code at the time of software release. Churn, Relative_Churn, and Entropy are measures of the code change process. Total, Major, Minor, and Ownership capture the degree of module responsibility that the authors of a change have.

Review coverage metrics. We measure the proportion of code change in a component that underwent code review. Reviewed_Commit is calculated by treating each commit as a discrete unit of equal value. Reviewed_Churn is calculated by treating each changed line as a discrete unit of equal value. As mentioned in Section 2.2, there is a substantial amount of external code in flux (See Figure 1). To account for this external code, we introduce In-House, i.e., the proportion of internally developed commits.

Review participation metrics. We study review participation along three dimensions: (1) the existence of a review that was written by other team members, i.e., the number of commits that were approved (Self_Approval) or verified (Self_Verify) by the authors themselves; (2) the time spent reviewing code, i.e., the Review_Window and the proportion of Hastily reviewed changes; and (3) the effort that was invested in improving the change, i.e., the number of comments in the review discussion (Discuss_Length), the proportion of reviews without discussions (No_Discuss), and the churn of a change during its review (Patch_Sd).

3.2 Model Construction

Similar to the prior study [18], we adopt the statistical approach of Harrell Jr. [10]. While previous work uses multiple linear regression models [18], we use logistic regression models because the proportion of components with multiple post-release defects is too low for counting models or linear fits. For our logistic fits, we label the components that include at least one post-release defect as defective. Conversely, those components that are free of post-release defects are labelled as clean.

Normality adjustment. For highly skewed metrics, we apply a logarithmic transformation to lessen the impact of outliers. We apply this logarithmic transformation to Size, Churn, and Relative_Churn, which have high variance values ($\simeq 10^5$).

Variable reduction. We perform a two-step correlation analysis to identify variables that are too highly correlated to include in the same model. The first step is to calculate the Spearman rank correlation between each pair of explanatory variables. We use hierarchical clustering to visualize the Spearman correlation values. Similar to prior work [18, 30], we consider a cluster of variables that has a Spearman correlation value of at least 0.7 to be too highly correlated to include together in the same model. We select one variable from each such cluster to include in the model.

In the second step, we examine how well each variable can be explained using a combination of the other variables. A variable that can be well-explained using other variables is redundant. We use the redundancy check implemented in the `redun` function of the `rms` R package [11], which builds models to predict the value of each explanatory variable using the others. If the fit of a model for an explanatory variable has an R^2 value of at least 0.9 (the default threshold of the `redun` function), the variable is considered redundant and is excluded from our defect models.

Model simplification. While the surviving metrics are not correlated or redundant, they may not contribute to the

Table 2: An overview of the studied software metrics.

	Metric	Log.	Description
Base	Prior_Defects		Number of prior defects [32].
	Size	✓	Lines of code.
	Churn	✓	Total amount of changed lines of code [22].
	Relative_Churn	✓	Normalized Churn by Size [22].
	Total		Number of unique committers of a component [6].
	Major		Number of unique committers whose commits represent more than 5% of all commits [6].
	Minor		Number of unique committers whose commits represent less than 5% of all commits [6].
	Ownership		Highest commit occupation ratio of the Major committers [6].
	Entropy		A measure of churn diversity among files in a component [12]. Calculated by $\sum_i^{L_c} \frac{p_i \log_2 p_i}{\log_2 L_c}$ where L_c is the churn of a component, p_i is a churn fraction of a file i.
	Complexity		Summation of McCabe's cyclomatic complexity number over files.
RQ1	Reviewed_Commit		Ratio of reviewed commits. Calculated by dividing the number of commits in the Gerrit code review system by that in `git log` [17, 18].
	Reviewed_Churn		Ratio of reviewed churn. Calculated as same as Reviewed_Commit [17, 18].
	∗In-House		Ratio of internal contribution in the entire history of the current branch. Calculated in the same manner as Reviewed_Commit.
RQ2	Self_Approval		Number of commits that are self-approved (i.e., when the engineer who authored a code change is only the engineer who provided a positive review score that grants submission privileges) [17, 18].
	∗Self_Verify		Number of commits that are self-verified (i.e., when the engineer who authored a code change is only the engineer who actually tested it and provided a positive verification score).
	Review_Window		Time interval between the upload of a commit until it is submitted [17, 18]. The median value across the changes to a component is used.
	Hastily		Number of commits which are hastily reviewed [17, 18].
	Discuss_Length		Average summed length of review comments on a commit until its submission [17, 18].
	No_Discuss		Number of commits that are submitted without any review comments [17, 18].
	∗Patch_Sd		Summation of normalized standard deviations of a patch churn until it is submitted. For example, if a change is revised 5 times with churn of (45,45,43,49,49) respectively, then its standard deviation is 2.9 and the mean is 46.6. In this case, Patch_Sd is 2.9/46.6=.06.

Abbreviations (Definition): Log. (Logarithmic transformation), ∗ (Newly added metrics in this study)

explanatory power of our defect models. To evaluate the contribution of our metrics, we examine the reduction in explanatory power between a preliminary model that contains all explanatory variables and another preliminary model that has all but one explanatory variable under test. Explanatory power of each preliminary model is estimated using the *AIC (Akaike Information Criterion)* [1]. If the AIC worsens after removing a variable, the variable is said to have an impact on the fit of the preliminary model and is retained for our final model fit. Otherwise, if the AIC does not worsen after removing a variable, it is excluded from our final model fit. This process is repeated until the model formula reaches a saturated form. We use `MASS::stepAIC` (with `dir="backward"` option) to evaluate the AIC.

3.3 Model Analysis

Similar to the prior work [18], we analyze our models from two perspectives: (1) model performance and (2) the impact of each explanatory variable on the model performance.

Model performance. We analyze the performance of constructed models using the discrimination index $D_{xy} = 2(c - 0.5)$ [7, 10], where c is the Area Under the receiver operating characteristic Curve (AUC). The AUC is a threshold-independent performance metric that measures a classifier's ability to discriminate between defective and clean components (i.e., do the defective components tend to have higher predicted probabilities than clean ones?). AUC is computed by measuring the area under the curve that plots the true positive rate against the false positive rate, while varying the threshold that is used to determine whether a component is classified as defective or clean. Values of AUC range between 0 (worst performance), 0.5 (random guessing performance), and 1 (best performance). Therefore, D_{xy} values range between -1 (worst performance), 0 (random guessing performance), and 1 (best performance). Furthermore, Hosmer *et al.* [7] state that $D_{xy} \geq 0.4$ can be considered as acceptable discrimination and $D_{xy} \geq 0.6$ as excellent discrimination.

The D_{xy} is inherently overestimated because the model is fitted and tested using the same data. To take model stability into account, we subtract the bootstrap-derived optimism [8] from the D_{xy}. The bootstrap technique draws n samples from the original dataset of size n with replacement. This procedure is repeated B times to create B new datasets. In each of the B datasets, the same model formula is applied but coefficients as well as confidence intervals are re-calculated. Using the bootstrap models, D_{xy} is calculated using both the bootstrap dataset and the original dataset. Then, the difference ΔD_{xy} between the bootstrap D_{xy} and the original D_{xy} is computed. Finally, the optimism is computed by taking the average of the ΔD_{xy} values across the B iterations. In our study, we use $B = 1,000$ iterations.

The explanatory power of our metrics. While the D_{xy} evaluates the model fits, we would like to estimate the impact that each explanatory variable has on our model performance. To that end, we show the coefficients, standard error, and χ^2 values of the explanatory variables in our fits. In the logistic regression model, the antilog of a coefficient is equivalent to the variable's *odds ratio*. Thus, a large coefficient indicates that an exploratory variable makes a large contribution to the performance of a model.

4. REPLICATION STUDY RESULTS

In this section, we present the results of our baseline model fits and our two research questions.

4.1 Baseline Model

Before we discuss the impact of review metrics on software quality, we first analyze the performance of a baseline model,

Table 3: The performance of our defect models.

	D_{xy} original (Diff. with Base)	Optimism	D_{xy} corrected (Diff. with Base)
Base	.637	.023	.614
RQ1	.695 (+ .058)	.026	.669 (+ .055)
RQ2	.718 (+ .081)	.045	.674 (+ .060)

Table 4: The relationship between explanatory variables and defect-proneness.

	Metric	Coef.	S.E.	χ^2	$\Pr(>\chi^2)$
Base	Size	+0.32	.09	4.01	.0453
	Churn	−0.32	.09	4.16	.0413
	Total	+0.07	.02	8.61	.0033
	Prior_Defects	+0.41	.12	14.02	.0002
RQ1	In-House	−1.44	.36	15.61	.0001
RQ2	Patch_Sd	−0.12	.04	8.77	.0031
	Self_Verify	+0.02	.01	6.26	.0124

which is trained using metrics that are known to share a relationship with defect-proneness (e.g., Size and Churn).
Baseline model construction. We first perform our correlation analysis, where we select Total instead of Ownership and Major because Total is easier to interpret. We then perform our redundancy analysis, where we find that Minor and Relative_Churn are redundant. Finally, we perform our model simplification, where we find that Entropy and Complexity provide insignificant amounts of explanatory power. Four metrics survive our model construction steps: Size, Churn, Total and Prior_Defects.
Baseline model analysis. Table 3 shows that our baseline model achieves a D_{xy} of 0.637, i.e., excellent discrimination [7]. The optimism value that we derive from 1,000 samples is small (0.023), which indicates that our model fit is robust. The optimism-corrected D_{xy} is 0.614, which still falls within the range of excellent discrimination [7].

Table 4 shows the contribution of each explanatory metric in our model fit. We observe that Prior_Defects, Size, and Total have a positive impact on defect-proneness, while surprisingly, Churn has a negative impact. We elaborate on the counterintuitive nature of the relationship between defect-proneness and Churn in Section 5.4.

Our baseline model achieves excellent discrimination, with a D_{xy} of 0.637. Prior_Defects, Size, *and* Total *share strong, increasing relationship with defect-proneness, while* Churn *shares strong, decreasing relationship with defect-proneness.*

4.2 Review Coverage (RQ1)

To address RQ1, we add our review coverage metrics to the baseline model and check whether the fit is improved.
Model construction (RQ1). In correlation analysis, we select Reviewed_Churn instead of Reviewed_Commit because it is a more exact measure of the amount of code that was reviewed. While redundancy analysis does not identify any problematic metrics, model simplification shows that Reviewed_Churn does not provide a significant amount of explanatory power. In summary, only the In-House review coverage metric survives our model construction steps.
Model analysis (RQ1). Table 3 shows that our new model achieves a D_{xy} of 0.695, outperforming the baseline model by 0.058. The optimism value is also small (0.026), which indicates that our model fit is robust. The optimism-corrected D_{xy} is 0.669, which still provides excellent discrimination [7].

Table 4 shows the contribution of the In-House metric. We observe that it has a large negative impact on defect-proneness. In Section 5.2, we study why only In-House contributes to our model fits.

Although our review coverage model outperforms our baseline model, of the three studied review coverage metrics, only the proportion of In-House *contributions contributes significantly to our model fits.*
Comparison with previous work. *Similar to the prior work [18], we find that* Reviewed_Commit *and* Reviewed_Churn *provide little explanatory power, suggesting that other reviewing factors are at play.*

4.3 Review Participation (RQ2)

To address RQ2, we add our review participation metrics to the RQ1 model and check if the model fit is improved. We use the RQ1 model as the baseline to control for coverage.
Model construction (RQ2). Correlation analysis does not identify any problematic pairs of explanatory variables. Redundancy analysis reveals that No_Discuss should be excluded. Model simplification reveals that Discuss_Length, Hastily, Self_Approval, and Review_Window provide insignificant amounts of explanatory power.

In short, we find that two review participation metrics survive our preliminary analyses: Patch_Sd and Self_Verify. Interestingly, we find that metrics that measure code reviewing time do not contribute distinct or significant amounts of explanatory power. We investigate this more in Section 5.3.
Model analysis (RQ2). Table 3 shows that our new model achieves a D_{xy} of 0.718, outperforming the baseline model by 0.081. On the other hand, the optimism value is slightly higher at 0.045, which suggests that our new model fit is less robust. Still, the optimism-corrected D_{xy} is 0.674, which is regarded as excellent discrimination [7].

Table 4 shows the impact of the Patch_Sd and Self_Verify metrics. We observe that Patch_Sd has a negative impact on defect-proneness, while Self_Verify has a positive impact. In Section 5.3, we qualitatively study what kinds of code reviewing practices are driving these results.

*Our review participation model also outperforms our baseline model. Of the studied review participation metrics, only the measure of accumulated effort to improve code changes (*Patch_Sd*) and the rate of author self-verification (*Self_Verify*) contribute significantly to our model fits.*
Comparison with previous work. *Unlike the prior work [18], code reviewing time and discussion length did not provide exploratory power to the Sony Mobile model.*

5. QUALITATIVE ANALYSIS

The purpose of qualitative analysis is to elicit expert opinion about our empirical results in Section 4. This analysis consists of three parts (see Figure 2); (***1***) a preliminary questionnaire (93 stakeholders), (***2***) follow-up interviews (6 sessions, 15 participants), and (***3***) a second questionnaire to validate our findings (25 senior developers).

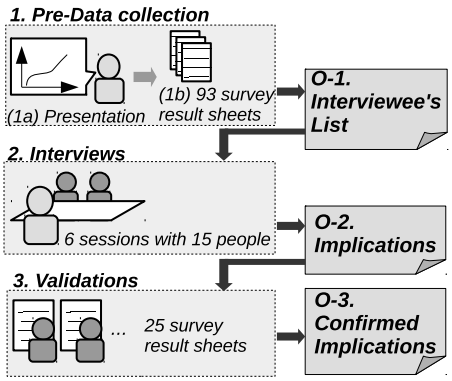

Figure 2: Approach for our qualitative analysis.

5.1 Methodology

We use the data collection and analysis approaches that are proposed by Seaman [27] to derive implications. We finally validate these implications with stakeholders. Figure 2 provides an overview of our qualitative analysis approach, which is composed of three parts:

1. Pre-data collection. The main objective of the pre-data collection is to identify stakeholders who have a breadth of experience in software development and/or management. Our aim is to triangulate our statistical results with stakeholder experience. We do so by explaining our statistical analysis and its results, and collecting and analyzing the stakeholder feedback.

First, we held a result sharing meeting to explain our statistical analysis and its results (***1a*** in Figure 2). We invited 300 members of the software development team at Sony Mobile to our meeting, of which 93 invitees attended (31%). Table 5 shows the summary of attendee profiles. The presentation took 20 minutes to elaborate on the research background, analysis methods, results, and findings.

Next, we issued a questionnaire to all of the attendees (***1b***). The questionnaire covered the stakeholder's technical background, impressions about the results, and their willingness to participate in an individual interview.

When selecting interviewees, we focused on stakeholders who made at least one pertinent comment on review metrics. The first author's background in software development at Sony Mobile also helped to identify key stakeholders. We selected 5 software engineers and 1 project manager as interviewees (***O-1***). The interviewees were asked to bring one or more colleagues from their team to provide a more objective perspective. In total, we interviewed 15 stakeholders.

2. Interviews. We conducted semi-structured interviews to uncover the code review practices at Sony Mobile. Semi-structured interviews begin with a set of prepared questions, but the structure of the interview is flexible, allowing the interviewer to dig into unexpected answers from the interviewees by developing new questions during the session [27]. Many of the questions are directly connected with our research interests, such as: *"Why is self-verification a common practice in your team?"*, or *"Why do you think that in-house components developed by your team tend to have fewer defects than external components?"*. All of the conversations were recorded and coded by the interviewer and were later checked by the interviewees for correctness.

Table 5: Profiles of respondent.

	Software Engineer	Testing / Quality Assurance	Project Manager	Total
Staff	69	7	5	81
Manager	6	4	2	12
Total	75	11	7	93

When analyzing the six interview transcripts, we grouped team responses according to the prepared questions. We selected the recurrent responses that were provided by at least three teams (i.e., 50% of the interviewed teams) for further validation in our secondary questionnaire. Using these recurring responses, we formulated six implications (***O-2***).

3. Validation. We validate the implications that we derive from our semi-structured interviews by performing a follow-up questionnaire, which was filled out by 25 senior software engineers. In analyzing the follow-up questionnaire, we check how many of the respondents agree with the derived implications (***O-3***).

The validation questionnaire presents the respondents with a list of our derived implications (presented in the rectangular boxes of Section 5.2 to 5.4) asking whether the implication agrees with their expertise or not. We calculate the ratio of respondents who agree with our implications out of all respondents. We exclude blank answers when calculating the ratio because these blank answers indicate that the respondents do not have the necessary expertise to answer the question.

5.2 Implications on Review Coverage

We qualitatively analyze review coverage from software quality and developer perspectives.

A) Why is In-House *associated with software quality, while* Reviewed_Commit *and* Reviewed_Churn *are not?*

Motivation. In-House represents the same concept as Reviewed_Commit and Reviewed_Churn about review coverage. The only difference is the measurement period: In-House sees the entire history of the branch. Hence, we are interested in understanding why In-House shares a stronger relationship with software quality than the other review coverage metrics.

Discussion. One project manager warns that we need special attention for external commits:

> *"Things go wrong when different systems are combined and integrated together. No matter how external commits ever got positive review scores in their code review system, we have to be aware that each system has its own test environment, approach, review criteria, which are different from ours."* - Project manager, 5 years

As mentioned in Section 2.2, an internal commit in the Gerrit system needs positive review scores before it is granted submission privileges. Although the integration of external code follows the same review process, the individual external commits are not reviewed by Sony Mobile engineers.

In-House measures the proportion of unreviewed commits during entire history of the component, whereas Reviewed_Churn and Reviewed_Commit include the external code changes that were merged during the current project period. We suspect this difference allows In-House to capture software quality more accurately than the other two metrics in the Sony Mobile context.

IN-HOUSE *captures the amount of unreviewed code in a more appropriate way for the Sony Mobile context than* REVIEWED_COMMIT/CHURN *metrics do, likely because of the difference in their measurement periods.*
Validation. *15 senior developers (75%) agreed and 5 senior developers (25%) disagreed with this implication.*

B) For developers, is it more difficult to improve software quality of components with a low IN-HOUSE *rate?*

Motivation. In the previous discussion, we studied the impact of IN-HOUSE on overall software quality. In this question, we focus on developers to study whether or not they are also affected by IN-HOUSE.
Discussion. A software engineer working on components of an external codebase explains his experiences:

> *"An internal codebase is much easier to work with, since I can discuss with the people who wrote the code. An external codebase takes more time from me to understand the code and to develop patches."* - Software engineer, 6 years

Code reviewing provides a mechanism for knowledge sharing [3]. The more that a codebase is developed and reviewed by internal developers, the more knowledge that they accumulate. An external codebase naturally has a smaller amount of information for developers. We suspect that developers generally have greater difficulty when they work on components that have originated from an external codebase. Since understanding plays a major role in defect repair [29], we suspect that, if left unchecked, a high dependence on external codebases may threaten software quality.

Developers require more time and effort to understand, extend, or repair components with low IN-HOUSE *rates.*
Validation. *23 senior developers (92%) agreed and 1 senior developer (4%) disagreed with this implication.*

5.3 Implications on Review Participation

In addition to IN-HOUSE, two of our review participation metrics have a significant impact on post-release defect-proneness. In the interviews, we focused on review practices of different teams to investigate why certain review practices became common and are associated with software quality. We formulate a question corresponding to each of the three studied dimensions of review participation (i.e., involvement, time, and discussion).

C) Why are higher rates of self-verification associated with lower software quality?

Motivation. Our models show that SELF_VERIFY is associated with software quality (see Section 4). We are interested in checking whether this self-verification trend agrees with expert opinion.
Discussion. While on the whole, self-verification is a bad practice, it has some advantages in the Sony Mobile context. For example, an author can verify that the intended functionality works precisely. On the other hand, an author is not a good candidate to verify unintended workflows of the program. One interviewee argues for the value of self-verification:

> *"Our components span across a low-level hardware layer to user-level application. I understand the architecture, and I am the one who can test my commit properly. Automated testing is not an option."* - Software engineer, 4 years

Yet, 4 out of 5 teams admitted that the self-verification practice is coloured by the author's subjective perspective, which may bias the test result. However, even if developers want to dedicate the testing work to other engineers, there are still many barriers to overcome, such as a detailed description of the testing environment, and identifying personnel with the appropriate expertise to carry out the test. Thus, self-verification is an alluring shortcut for developers to take.

Self-verified commits may yield biased testing criteria. However, practicalities of the complex development environment and time pressure of releases make self-verification an alluring shortcut for developers to take.
Validation. *20 senior developers (87%) agreed and 3 senior developers (13%) disagreed with this implication.*

D) Why do the metrics related to reviewing time have little impact on our software quality models?

Motivation. The prior study [18] showed that reviewing time shared a significant relationship with software quality in 2 of 4 studied releases of OSS systems. In Section 4, we find that the reviewing time metrics do not contribute to our model fits. Hence, we are interested in investigating why this might be the case for the Sony Mobile project.
Discussion. One potential reason for the discrepancy is provided by a senior engineer:

> *"I have many reviews in my to-do' list, but the order in which I do them is dictated by team priorities, e.g., usually hot-fixes are reviewed first. Any other factors, e.g., size or complexity of commits do not matter. Does that give rise to longer reviewing time for a small and low-priority commits?"* - Senior application software architect, 10 years

The architect points out that issue priority may influence reviewing time. HASTILY may have had a significant impact on the OSS systems of the prior work [18] because issue priority is often not set properly or is ignored in many OSS projects [20]. Commercial projects tend to put more emphasis on issue priority [16].

Issue priority may cause reviewing time measures to lose meaning in the Sony Mobile context.
Validation. *25 senior developers (100%) agreed with this implication.*

E) Why is increased PATCH_SD *associated with lower software quality?*

Motivation. Our models show that PATCH_SD is significantly associated with post-release defect-proneness (see Section 4). Hence, we are interested in investigating why.
Discussion. An interview with a senior software architect highlights the importance of offline code improvement:

> *"We have a conventional code review meeting regularly. When code change spans across different files, it is much easier to work with direct communication rather than with the Gerrit tools. We can talk in our language (Japanese) too unlike in Gerrit."* - Senior platform software architect, 10 years

At Sony Mobile, many developers rely on in-person communication (see Section 2.1) and claim that this is where code improvement activities take place. We suspect that the

volume of discussion in Gerrit, i.e., DISCUSS_LENGTH, only captures a limited amount of the code improvement activity. Hence, our discussion length metrics do not contribute as much value to our models as PATCH_SD does.

Self-improvement is another type of review activity that PATCH_SD captures, yet discussion metrics do not. There is a large proportion of commits that are revised several times without reviewer's feedback. Two engineers explained the motivation for updating commits before review feedback:

> *"I review my code on the browser. The GUI difference with my local text editor or IDE can make me more attentive to catch easy mistakes."* - 2 platform software engineers with 3 and 4 years of experience

Self-motivated code improvement is a common phenomena to improve software quality in other software projects, too [5, 30]. Self-reviewing practices also increase the chance of detecting underlying software defects. Indeed, PATCH_SD seems to capture the efforts of a developer to improve software quality in a more appropriate way for the Sony Mobile context than other review participation metrics do.

> PATCH_SD *captures developer effort in a way that is not diminished by offline review discussion at Sony Mobile.*
> **Validation.** *17 senior developers (81%) agreed and 4 senior developers (19%) disagreed with this implication.*

5.4 Miscellaneous Findings

Our final finding is associated with baseline metrics which are indirectly associated with our research interest in code review practices.

F) Why is more churn associated with higher software quality?

Motivation. Nagappan and Ball [22] found that the more lines of code that churned, the higher the likelihood of defects. In the studied Sony Mobile system, we observe the opposite, i.e., increases in code churn are associated with improvement in software quality. We are interested in investigating this counterintuitive indication of our models.
Discussion. Nagappan and Ball [22] studied a Windows release (W2k3-SP1), which mainly contains security vulnerability fixes and enhancements to pre-installed programs. Our studied Sony Mobile project has several different characteristics. For example, the Sony Mobile code changes also implement new features and apps beyond defect fixes (see Section 2.1). New features are often contributed to in in-house components. In Section 5.2, we suggested that in-house components tend to have more of a positive impact on software quality and developers' knowledge than external components. A senior application engineer also comments:

> *"New feature implementations of in-house components have much larger lines of code than defect-fixes in external components because we can write the code quicker."* - Senior application engineer, 10 years

We suspect that these characteristic differences are at the heart of the counterintuitive results that we arrive at with respect to CHURN. To investigate our suspicion, we compute a subset of the total churn that measures only the churn of defect fixes (PRIOR_DEFECTS_CHURN). We then used PRIOR_DEFECTS_-CHURN instead of CHURN and refit our models. Our new model fits show that PRIOR_DEFECTS_CHURN has a significant positive impact on the post-release defect-proneness, i.e., increases in PRIOR_DEFECTS_CHURN are associated with increases in post-release defect-proneness.

> *Characteristics of the types of changes and the target system type may be leading to this counterintuitive relationship between code churn and software quality.*
> **Validation.** *13 senior developers (68%) agreed and 6 senior developers (32%) disagreed with this implication.*

6. ACTION PLANS AND RELATED WORK

Table 6 provides an overview of the findings and implications. We find that all of the implications related to code review practices are supported by more than 75% of the senior engineers who responded to our follow-up questionnaire.

In this section, we discuss how the findings of our study: (a) have formed an action plan to improve code reviewing practices at Sony Mobile and (b) fit with the code review literature. We structure the discussion along the review coverage and participation dimensions of our study.

6.1 Review Coverage

Prior work suggests that coverage of the review process is important. Fagan [9] and Kemerer and Paulk [14] found that the introduction of an inspection process that covered all of the design and code changes lead to improvements in software quality. Tanaka et al. [28] suggest that a software team should meticulously review each change to the source code. Bavota and Russo [4] find that unreviewed code is two times more likely to introduce defects than reviewed code.

On the other hand, recent empirical studies suggest that review coverage is not the most important characteristic of a review process. For example, McIntosh et al. [17, 18] found that review coverage only shares a significant link with the incidence of post-release defects in two of four studied OSS releases. Morales et al. [21] found that review coverage only shares a significant link with software design quality in two of four studied OSS releases. Furthermore, Meneely et al. [19] find that the Chromium project enforces a 100% review coverage policy, rendering review coverage moot.

Implications for Sony Mobile. We find that the rate of churn in external code shares a significant link with software quality. Moving forward, the quality assurance teams of Sony Mobile have been made aware of this trend, and steps are underway to improve test coverage of external code.

6.2 Review Participation

A recent line of work has highlighted the importance of the investment that review participants make in the code review process. Bacchelli and Bird [3] find that the modern code review process has become a mechanism to support collaborative problem solving. McIntosh et al. [17, 18] and Morales et al. [21] find that review participation shares a consistent link with the incidence of post-release defects and design anti-patterns, respectively. Kononenko et al. [15] also find that review participation metrics are inversely associated with the commits that introduce defects.

Indeed, additional perspectives are invaluable in the code review process. Thongtanunam et al. [30] find that software modules that involve multiple reviewers in the code review process tend to be less susceptible to defects than software modules that involve few reviewers.

Table 6: Summary of findings and implications.

Dimension	Section	Findings and Implications	Ratio
Review Coverage	4.2	Although our review coverage model outperforms our baseline model, of the three studied review coverage metrics, only the proportion of IN-HOUSE contributions contributes significantly to our model fits.	-
	5.2.A	IN-HOUSE captures the amount of unreviewed code in a more appropriate way for the Sony Mobile context than REVIEWED_COMMIT/CHURN metrics do, likely because of the difference in their measurement periods.	75%
	5.2.B	Developers require more time and effort to understand, extend, or repair components with low IN-HOUSE rates.	92%
Review Participation	4.3	Our review participation model also outperforms our baseline model. Of the studied review participation metrics, only the measure of accumulated effort to improve code changes (PATCH_SD) and the rate of author self-verification (SELF_VERIFY) contribute significantly to our model fits.	-
	5.3.C	Self-verified commits may yield biased testing criteria. However, practicalities of the complex development environment and time pressure of releases make self-verification an alluring shortcut for developers to take.	87%
	5.3.D	Issue priority may cause reviewing time measures to lose meaning in the Sony Mobile context.	100%
	5.3.E	PATCH_SD captures developer effort in a way that is not diminished by offline review discussion at Sony Mobile.	
Baseline	4.1	Our baseline model achieves excellent discrimination, with a D_{xy} of 0.637. PRIOR_DEFECTS, SIZE, and TOTAL share strong, increasing relationship with defect-proneness, while CHURN shares strong, decreasing relationship with defect-proneness.	-
	5.4.F	Characteristics of the types of changes and the target system type may be leading to this counterintuitive relationship between code churn and software quality.	68%

Implications for Sony Mobile. We find that active participation in the code review process indeed has a positive impact on software quality, whereas improvement activities can take various forms (e.g., offline code review meetings and self-improvement). This result has lead the Sony Mobile development team to more confidently invest effort in the code review process. Furthermore, the Sony Mobile team has set out to improve its code review process by encouraging (a) passive developers to more actively participate in the code review process and (b) all developers to verify the code changes of other colleagues instead of their own.

7. THREATS TO VALIDITY

In this section, we discuss the extent to which our results are threatened by our experimental design choices.

Construct validity. We assume prior/post-release defects are defects that were fixed before/after the product was released. At the time right before the release of commercial systems, the defects that affect a wide range of components are not likely to be fixed to mitigate the risk of regression. Indeed, the number of prior/post-release defects that we compute may not exactly match the number of prior/post-release defects that the software system actually contains.

Internal validity. In our qualitative analysis, we selected 5 software engineers and 1 project manager as interviewees. This selection may bias our results. To mitigate this bias, we asked interviewees to bring colleagues from their teams and also performed an external validation with 25 senior engineers. Nonetheless, the implications we derived may be biased by the interviewer's background and the small number of people who commented during the interviews.

External validity. Our study focuses on one proprietary software system developed by Sony Mobile. This system has several unique characteristics, such as the integration of in-house and external codebases and being developed by a network of colocated teams. This project might not be representative of all proprietary software systems. However, generalizability is not the main goal of the study. Our findings have been useful to help Sony Mobile to put into action a plan to improve their code review process. Our findings may be useful for other teams with similar characteristics.

8. CONCLUSIONS

We quantitatively investigated the impact of code reviewing practices, and complemented our findings with a qualitative analysis involving 93 stakeholders at Sony Mobile. While prior metrics of review coverage do not share a relationship with defect-proneness in the Sony Mobile system, the rate at which externally developed code (which is not code reviewed by Sony Mobile engineers) is integrated with Sony Mobile components shares a strong association with defect-proneness. Furthermore, code review participation also shares a significant link with software quality. For example, author self-verification (i.e., when the engineer who authored a code change is also the engineer responsible for testing it) and the amount that code changes are improved during the code review process are associated with software quality. Our findings, which are confirmed by stakeholders, have informed our process improvement plan for code reviews at Sony Mobile.

9. ACKNOWLEDGMENTS

This research was partially supported by JSPS KAKENHI Grant Numbers 15H05306. The authors would like to thank D. Pursehouse, T. Nakagawa, S. Fujita, M. Tashiro, M. Matsuo and anonymous stakeholders at Sony Mobile to contribute to the study. The findings and opinions expressed in this paper are those of the authors and do not necessarily represent or reflect those of Sony Mobile and/or its subsidiaries and affiliates. Moreover, our results do not in any way reflect the quality of Sony Mobile's products.

10. REFERENCES

[1] H. Akaike. Information theory and an extension of the maximum likelihood principle. In *Proc. of the Int'l Symp. on Information Theory*, pages 267–281, 1973.

[2] J. Aranda and G. Venolia. The secret life of bugs: Going past the errors and omissions in software repositories. In *Proc. of the Int'l Conf. on Software Eng.*, pages 298–308, 2009.

[3] A. Bacchelli and C. Bird. Expectations, outcomes, and challenges of modern code review. In *Proc. of the Int'l Conf. on Software Eng.*, pages 712–721, 2013.

[4] G. Bavota and R. Barbara. Four eyes are better than two: On the impact of code reviews on software quality. In *Proc. of the Int'l Conf. on Software Maint. and Evolution*, pages 81–90, 2015.

[5] M. Beller, A. Bacchelli, A. Zaidman, and E. Juergens. Modern code reviews in open-source projects: Which problems do they fix? In *Proc. of Working Conf. on Mining Software Repositories*, pages 202–211, 2014.

[6] 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 *Proc. of the ACM SIGSOFT Symp. and the European Conf. on Foundations of Software Eng.*, pages 4–14, 2011.

[7] J. David W. Hosmer, S. Lemeshow, and R. X. Sturdivant. *Applied Logistic Regression*. Wiley, 2013.

[8] B. Efron and R. J. Tibshirani. *An Introduction to the Bootstrap*. Chapman & Hall, 1993.

[9] M. E. Fagan. Design and code inspections to reduce errors in program development. *IBM Syst. J.*, 38(2-3):258–287, 1999.

[10] F. E. Harrell Jr. *Regression modeling strategies*. Springer, 2001.

[11] F. E. Harrell Jr. *rms: Regression Modeling Strategies*, 2015. R package version 4.3-0.

[12] A. Hassan. Predicting faults using the complexity of code changes. In *Proc. of the Int'l Conf. on Software Eng.*, pages 78–88, 2009.

[13] J. D. Herbsleb and R. E. Grinter. Architectures, coordination, and distance: Conway's law and beyond. *IEEE Software*, 16(5):63–70, 1999.

[14] C. Kemerer and M. Paulk. The impact of design and code reviews on software quality: An empirical study based on PSP data. *IEEE Trans. Software Eng.*, 35(4):534–550, 2009.

[15] O. Kononenko, O. Baysal, L. Guerrouj, Y. Cao, and M. Godfrey, W. Investigating code review quality: Do people and participation matter? In *Proc. of the Int'l Conf. on Software Maint. and Evolution*, pages 111–120, 2015.

[16] M. Lavallée and P. N. Robillard. Why good developers write bad code: An observational case study of the impacts of organizational factors on software quality. In *Proc. of the Int'l Conf. on Software Eng.*, pages 677–687, 2015.

[17] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan. The impact of code review coverage and code review participation on software quality: A case study of the Qt, VTK, and ITK projects. In *Proc. of the Working Conf. on Mining Software Repositories*, pages 192–201, 2013.

[18] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan. An empirical study of the impact of modern code review practices on software quality. *Empirical Software Eng.*, page To appear, 2015.

[19] A. Meneely, A. C. R. Tejeda, 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 *Proc. of the Int'l Workshop on Social Software Eng.*, pages 37–44, 2014.

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

[21] R. Morales, S. McIntosh, and F. Khomh. Do code review practices impact design quality? a case study of the Qt, VTK, and ITK projects. In *Proc. of the Int'l Conf. on Software Analysis, Evolution and Reengineering*, pages 171–180, 2015.

[22] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *Proc. of the Int'l Conf. on Software Eng.*, pages 284–292, 2005.

[23] N. Nagappan, B. Murphy, and V. Basili. The influence of organizational structure on software quality: An empirical case study. In *Proc. of the Int'l Conf. on Software Eng.*, pages 521–530, 2008.

[24] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.

[25] P. C. Rigby and C. Bird. Convergent contemporary software peer review practices. In *Proc. of the ACM SIGSOFT Symp. and the European Conf. on Foundations of Software Eng.*, pages 202–212, 2013.

[26] P. C. Rigby and M. Storey. Understanding broadcast based peer review on open source software projects. In *Proc. of the Int'l Conf. on Software Eng.*, pages 541–550, 2011.

[27] C. B. Seaman. Qualitative methods. *Guide to Advanced Empirical Software Engineering, Springer*, pages 35–62, 2008.

[28] T. Tanaka, K. Sakamoto, S. Kusumoto, K. Matsumoto, and T. Kikuno. Improvement of software process by process description and benefit estimation. In *Proc. of the Int'l Conf. on Software Eng.*, pages 123–132, 1995.

[29] Y. Tao, Y. Dang, T. Xie, D. Zhang, and S. Kim. How do software engineers understand code changes?: An exploratory study in industry. In *Proc. of the ACM SIGSOFT Symp. on the Foundations of Software Eng.*, pages 51:1–51:11, 2012.

[30] P. Thongtanunam, S. McIntosh, A. E. Hassan, and H. Iida. Investigating code review practices in defective files: An empirical study of the qt system. In *Proc. of the Working Conf. on Mining Software Repositories*, pages 168–179, 2015.

[31] L. G. Votta, Jr. Does every inspection need a meeting? In *Proc. of the ACM SIGSOFT Symp. on Foundations of Software Eng.*, pages 107–114, 1993.

[32] T. Yu, V. Y. Shen, and H. E. Dunsmore. An analysis of several software defect models. *IEEE Trans. Software Eng.*, 14(9):1261–1270, 1988.