The Impact of Code Review Coverage and Code Review Participation on Software Quality

A Case Study of the Qt, VTK, and ITK Projects

Shane McIntosh[1], Yasutaka Kamei[2], Bram Adams[3], and Ahmed E. Hassan[1]
[1]Queen's University, Canada [2]Kyushu University, Japan [3]Polytechnique Montréal, Canada
[1]{mcintosh, ahmed}@cs.queensu.ca [2]kamei@ait.kyushu-u.ac.jp [3]bram.adams@polymtl.ca

ABSTRACT

Software code review, i.e., the practice of having third-party team members critique changes to a software system, is a well-established best practice in both open source and proprietary software domains. Prior work has shown that the formal code inspections of the past tend to improve the quality of software delivered by students and small teams. However, the formal code inspection process mandates strict review criteria (e.g., in-person meetings and reviewer checklists) to ensure a base level of review quality, while the modern, lightweight code reviewing process does not. Although recent work explores the modern code review process qualitatively, little research quantitatively explores the relationship between properties of the modern code review process and software quality. Hence, in this paper, we study the relationship between software quality and: (1) code review coverage, i.e., the proportion of changes that have been code reviewed, and (2) code review participation, i.e., the degree of reviewer involvement in the code review process. Through a case study of the Qt, VTK, and ITK projects, we find that both code review coverage and participation share a significant link with software quality. Low code review coverage and participation are estimated to produce components with up to two and five additional post-release defects respectively. Our results empirically confirm the intuition that poorly reviewed code has a negative impact on software quality in large systems using modern reviewing tools.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Code inspections and walk-throughs*

General Terms

Management, Measurement

Keywords

Code reviews, software quality

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

MSR'14, May 31 – June 1, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2863-0/14/05...$15.00
http://dx.doi.org/10.1145/2597073.2597076

1. INTRODUCTION

Software code reviews are a well-documented best practice for software projects. In Fagan's seminal work, formal design and code inspections with in-person meetings were found to reduce the number of errors detected during the testing phase in small development teams [8]. Rigby and Bird find that the modern code review processes that are adopted in a variety of reviewing environments (e.g., mailing lists or the Gerrit web application[1]) tend to converge on a lightweight variant of the formal code inspections of the past, where the focus has shifted from defect-hunting to group problem-solving [34]. Nonetheless, Bacchelli and Bird find that one of the main motivations of modern code review is to improve the quality of a change to the software prior to or after integration with the software system [2].

Prior work indicates that formal design and code inspections can be an effective means of identifying defects so that they can be fixed early in the development cycle [8]. Tanaka *et al.* suggest that code inspections should be applied meticulously to each code change [39]. Kemerer and Faulk indicate that student submissions tend to improve in quality when design and code inspections are introduced [19]. However, there is little quantitative evidence of the impact that modern, lightweight code review processes have on software quality in large systems.

In particular, to truly improve the quality of a set of proposed changes, reviewers must carefully consider the potential implications of the changes and engage in a discussion with the author. Under the formal code inspection model, time is allocated for preparation and execution of in-person meetings, where reviewers and author discuss the proposed code changes [8]. Furthermore, reviewers are encouraged to follow a checklist to ensure that a base level of review quality is achieved. However, in the modern reviewing process, such strict reviewing criteria are not mandated [36], and hence, reviews may not foster a sufficient amount of discussion between author and reviewers. Indeed, Microsoft developers complain that reviews often focus on minor logic errors rather than discussing deeper design issues [2].

We hypothesize that a modern code review process that neglects to review a large proportion of code changes, or suffers from low reviewer participation will likely have a negative impact on software quality. In other words:

[1] `https://code.google.com/p/gerrit/`

If a large proportion of the code changes that are integrated during development are either: (1) omitted from the code review process (low review coverage), or (2) have lax code review involvement (low review participation), then defect-prone code will permeate through to the released software product.

Tools that support the modern code reviewing process, such as Gerrit, explicitly link changes to a software system recorded in a Version Control System (VCS) to their respective code review. In this paper, we leverage these links to calculate code review coverage and participation metrics and add them to Multiple Linear Regression (MLR) models that are built to explain the incidence of *post-release defects* (i.e., defects in official releases of a software product), which is a popular proxy for software quality [5, 13, 18, 27, 30]. Rather than using these models for defect prediction, we analyze the impact that code review coverage and participation metrics have on them while controlling for a variety of metrics that are known to be good explainers of code quality. Through a case study of the large Qt, VTK, and ITK open source systems, we address the following two research questions:

(RQ1) Is there a relationship between code review coverage and post-release defects?
Review coverage is negatively associated with the incidence of post-release defects in all of our models. However, it only provides significant explanatory power to two of the four studied releases, suggesting that review coverage alone does not guarantee a low incidence rate of post-release defects.

(RQ2) Is there a relationship between code review participation and post-release defects?
Developer participation in code review is also associated with the incidence of post-release defects. In fact, when controlling for other significant explanatory variables, our models estimate that components with lax code review participation will contain up to five additional post-release defects.

Paper organization. The remainder of the paper is organized as follows. Section 2 describes the Gerrit-driven code review process that is used by the studied systems. Section 3 describes the design of our case study, while Section 4 presents the results of our two research questions. Section 5 discloses the threats to the validity of our study. Section 6 surveys related work. Finally, Section 7 draws conclusions.

2. GERRIT CODE REVIEW

Gerrit is a modern code review tool that facilitates a traceable code review process for `git`-based software projects [4]. Gerrit tightly integrates with test automation and code integration tools. Authors upload *patches*, i.e., collections of proposed changes to a software system, to a Gerrit server. The set of reviewers are either: (1) invited by the author, (2) appointed automatically based on their expertise with the modified system components, or (3) self-selected by broadcasting a review request to a mailing list. Figure 1 shows an example code review in Gerrit that was uploaded on December 1st, 2012. We use this figure to illustrate the role that reviewers and verifiers play in a code review below.

Reviewers. The reviewers are responsible for critiquing the changes proposed within the patch by leaving comments

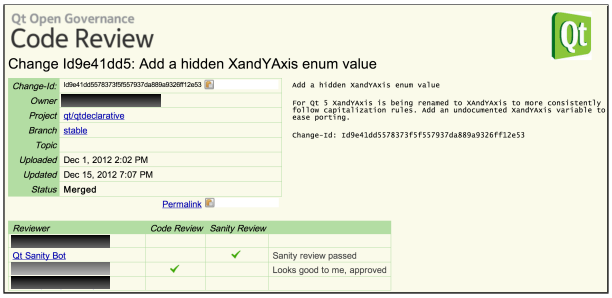

Figure 1: An example Gerrit code review.

for the author to address or discuss. The author can reply to comments or address them by producing a new revision of the patch for the reviewers to consider.

Reviewers can also give the changes proposed by a patch revision a *score*, which indicates: (1) agreement or disagreement with the proposed changes (positive or negative value), and (2) their level of confidence (1 or 2). The second column of the bottom-most table in Figure 1 shows that the change has been reviewed and the reviewer is in agreement with it (+). The text in the fourth column ("Looks good to me, approved") is displayed when the reviewer has a confidence level of two.

Verifiers. In addition to reviewers, verifiers are also invited to evaluate patches in the Gerrit system. Verifiers execute tests to ensure that: (1) patches truly fix the defect or add the feature that the authors claim to, and (2) do not cause regression of system behaviour. Similar to reviewers, verifiers can provide comments to describe verification issues that they have encountered during testing. Furthermore, verifiers can also provide a score of 1 to indicate successful verification, and -1 to indicate failure.

While team personnel can act as verifiers, so too can Continuous Integration (CI) tools that automatically build and test patches. For example, CI build and testing jobs can be automatically generated each time a new review request or patch revision is uploaded to Gerrit. The reports generated by these CI jobs can be automatically appended as a verification report to the code review discussion. The third column of the bottom-most table in Figure 1 shows that the "Qt Sanity Bot" has successfully verified the change.

Automated integration. Gerrit allows teams to codify code review and verification criteria that must be satisfied before changes are integrated into upstream VCS repositories. For example, a team policy may specify that at least one reviewer and one verifier provide positive scores prior to integration. Once the criteria are satisfied, patches are automatically integrated into upstream repositories. The "Merged" status shown in the upper-most table of Figure 1 indicates that the proposed changes have been integrated.

3. CASE STUDY DESIGN

In this section, we present our rationale for selecting our research questions, describe the studied systems, and present our data extraction and analysis approaches.

(RQ1) Is there a relationship between code review coverage and post-release defects?
Tanaka *et al.* suggest that a software team should meticulously review each change to the source code

to ensure that quality standards are met [39]. In more recent work, Kemerer and Faulk find that design and code inspections have a measurable impact on the defect density of student submissions at the Software Engineering Institute (SEI) [19]. While these findings suggest that there is a relationship between code review coverage and software quality, it has remained largely unexplored in large software systems using modern code review tools.

(RQ2) Is there a relationship between code review participation and post-release defects?
To truly have an impact on software quality, developers must invest in the code reviewing process. In other words, if developers are simply approving code changes without discussing them, the code review process likely provides little value. Hence, we set out to study the relationship between developer participation in code reviews and software quality.

3.1 Studied Systems

In order to address our research questions, we perform a case study on large, successful, and rapidly-evolving open source systems with globally distributed development teams. In selecting the subject systems, we identified two important criteria that needed to be satisfied:

Criterion 1: Reviewing Policy – We want to study systems that have made a serious investment in code reviewing. Hence, we only study systems where a large number of the integrated patches have been reviewed.

Criterion 2: Traceability – The code review process for a subject system must be *traceable*, i.e., it should be reasonably straightforward to connect a large proportion of the integrated patches to the associated code reviews. Without a traceable code review process, review coverage and participation metrics cannot be calculated, and hence, we cannot perform our analysis.

To satisfy the traceability criterion, we focus on software systems using the Gerrit code review tool. We began our study with five subject systems, however after preprocessing the data, we found that only 2% of *Android* and 14% of *LibreOffice* changes could be linked to reviews, so both systems had to be removed from our analysis (Criterion 1).

Table 1 shows that the *Qt*, *VTK*, and *ITK* systems satisfied our criteria for analysis. *Qt* is a cross-platform application framework whose development is supported by the Digia corporation, however welcomes contributions from the community-at-large.[2] The *Visualization ToolKit* (VTK) is used to generate 3D computer graphics and process images.[3] The *Insight segmentation and registration ToolKit* (ITK) provides a suite of tools for in-depth image analysis.[4]

3.2 Data Extraction

In order to evaluate the impact that code review coverage and participation have on software quality, we extract code review data from the Gerrit review databases of the studied systems, and link the review data to the integrated patches recorded in the corresponding VCSs.

[2] `http://qt.digia.com/`
[3] `http://vtk.org/`
[4] `http://itk.org/`

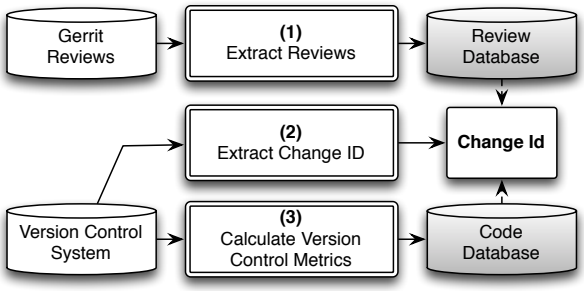

Figure 2: Overview of our data extraction approach.

Figure 2 shows that our data extraction approach is broken down into three steps: (1) extract review data from the Gerrit review database, (2) extract Gerrit change IDs from the VCS commits, and (3) calculate version control metrics. We briefly describe each step of our approach below.

Extract reviews. Our analysis is based on the Qt code reviews dataset collected by Hamasaki *et al.* [12]. The dataset describes each review, the personnel involved, and the details of the review discussions. We expand the dataset to include the reviews from the VTK and ITK systems, as well as those reviews that occurred during more recent development of Qt 5.1.0. To do so, we use a modified version of the GerritMiner scripts provided by Mukadam *et al.* [28].

Extract change ID. Each review in a Gerrit database is uniquely identified by an alpha-numeric hash code called a *change ID*. When a review has satisfied project-specific criteria, it is automatically integrated into the upstream VCS (*cf.* Section 2). For traceability purposes, the commit message of the automatically integrated patch contains the change ID. We extract the change ID from commit messages in order to automatically connect patches in the VCS with the associated code review process data. To facilitate future work, we have made the code and review databases available online.[5]

Calculate version control metrics. Prior work has found that several types of metrics have a relationship with defect-proneness. Since we aim to investigate the impact that code reviewing has on defect-proneness, we control for the three most common families of metrics that are known to have a relationship with defect-proneness [5, 13, 38]. Table 2 provides a brief description and the motivating rationale for each of the studied metrics.

We focus our analysis on the development activity that occurs on or has been merged into the `release` branch of each studied system. Prior to a release, the integration of changes on a `release` branch is more strictly controlled than a typical development branch to ensure that only the appropriately triaged changes will appear in the upcoming release. Moreover, changes that land on a release branch after a release are also strictly controlled to ensure that only high priority fixes land in maintenance releases. In other words, the changes that we study correspond to the development and maintenance of official software releases.

To determine whether a change fixes a defect, we search VCS commit messages for co-occurrences of defect identifiers with keywords like "bug", "fix", "defect", or "patch". A similar approach was used to determine defect-fixing and defect-inducing changes in other work [18, 20]. Similar to

[5] `http://sailhome.cs.queensu.ca/replication/reviewing_quality/`

Table 1: Overview of the studied systems. Those above the double line satisfy our criteria for analysis.

Overview				Components		Commits		Personnel	
Product	Version	Tag name	Lines of code	With defects	Total	With reviews	Total	Authors	Reviewers
Qt	5.0.0	v5.0.0	5,560,317	254	1,339	10,003	10,163	435	358
	5.1.0	v5.1.0	5,187,788	187	1,337	6,795	7,106	422	348
VTK	5.10.0	v5.10.0	1,921,850	15	170	554	1,431	55	45
ITK	4.3.0	v4.3.0	1,123,614	24	218	344	352	41	37
Android	4.0.4	4.0.4_r2.1	18,247,796	-	-	1,727	80,398	-	-
LibreOffice	4.0.0	4.0.0	4,789,039	-	-	1,679	11,988	-	-

prior work [18], we define post-release defects as those with fixes recorded in the six-month period after the release date.

Product metrics. Product metrics measure the source code of a system at the time of a release. It is common practice to preserve the released versions of the source code of a software system in the VCS using tags. In order to calculate product metrics for the studied releases, we first extract the released versions of the source code by "checking out" those tags from the VCS.

We measure the size and complexity of each component (i.e., directory) as described below. We measure the size of a component by aggregating the number of lines of code in each of its files. We use McCabe's cyclomatic complexity [23] (calculated using Scitools Understand[6]) to measure the complexity of a file. To measure the complexity of a component, we aggregate the complexity of each file within it. Finally, since complexity measures are often highly correlated with size, we divide the complexity of each component by its size to reduce the influence of size on complexity measures. A similar approach was used in prior work [17].

Process metrics. Process metrics measure the change activity that occurred during the development of a new release. Process metrics must be calculated with respect to a time period and a development branch. Again, similar to prior work [18], we measure process metrics using the six-month period prior to each release date on the `release` branch.

We use prior defects, churn, and change entropy to measure the change process. We count the number of defects fixed in a component prior to a release by using the same pattern-based approach we use to identify post-release defects. Churn measures the total number of lines added and removed to a component prior to release. Change entropy measures how the complexity of a change process is distributed across files [13]. To measure the change entropy in a component, we adopt the time decay variant of the History Complexity Metric (HCM^{1d}), which reduces the impact of older changes, since prior work identified HCM^{1d} as the most powerful HCM variant for defect prediction [13].

Human factors. Human factor metrics measure developer expertise and code ownership. Similar to process metrics, human factor metrics must also be calculated with respect to a time period. We again adopt a six-month period prior to each release date as the window for metric calculation.

We adopt the suite of ownership metrics proposed by Bird *et al.* [5]. Total authors is the number of authors that contribute to a component. Minor authors is the number of authors that contribute fewer than 5% of the commits to a component. Major authors is the number of authors that contribute at least 5% of the commits to a component. Author ownership is the proportion of commits that the most active contributor to a component has made.

[6] `http://www.scitools.com/documents/metricsList.php?#Cyclomatic`

3.3 Model Construction

We build Multiple Linear Regression (MLR) models to explain the incidence of post-release defects detected in the components of the studied systems. An MLR model fits a line of the form $y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_n x_n$ to the data, where y is the dependent variable and each x_i is an explanatory variable. In our models, the dependent variable is post-release defect count and the explanatory variables are the set of metrics outlined in Table 2.

Similar to Mockus [25] and others [6, 37], our goal is to understand the relationship between the explanatory variables (code review coverage and participation) and the dependent variable (post-release defect counts). Hence, we adopt a similar model construction technique.

To lessen the impact of outliers on our models, we apply a log transformation [$log(x+1)$] to those metrics whose values are natural numbers. To handle metrics whose values are proportions ranging between 0 and 1, we apply a logit transformation [$log(\frac{x}{1-x})$]. Since the logit transformations of 0 and 1 yield undefined values, the data is proportionally remapped to a range between 0.025 and 0.975 by the `logit` function provided by the `car` package [10] in `R`.

Minimizing multicollinearity. Prior to building our models, we check for explanatory variables that are highly correlated with one another using Spearman rank correlation tests (ρ). We choose a rank correlation instead of other types of correlation (e.g., Pearson) because rank correlation is resilient to data that is not normally distributed. We consider a pair of variables highly correlated when $|\rho| > 0.7$, and only include one of the pair in the model.

In addition to correlation analysis, after constructing preliminary models, we check them for multicollinearity using the Variance Inflation Factor (VIF) score. A VIF score is calculated for each explanatory variable used by the model. A VIF score of 1 indicates that there is no correlation between the variable and others, while values greater than 1 indicate the ratio of inflation in the variance explained due to collinearity. We select a VIF score threshold of five as suggested by Fox [9]. When our models contain variables with VIF scores greater than five, we remove the variable with the highest VIF score from the model. We then recalculate the VIF scores for the new model and repeat the removal process until all variables have VIF scores below five.

3.4 Model Analysis

After building MLR models, we evaluate the goodness of fit using the *Akaike Information Criterion* (AIC) [1] and the *Adjusted* R^2 [14]. Unlike the unadjusted R^2, the AIC and the adjusted R^2 account for the bias of introducing additional explanatory variables by penalizing models for each additional metric.

To decide whether an explanatory variable is a signifi-

Table 2: A taxonomy of the considered control (top) and reviewing metrics (bottom).

	Metric	Description	Rationale
Prod	Size	Number of lines of code.	Large components are more likely to be defect-prone [21].
	Complexity	The McCabe cyclomatic complexity.	More complex components are likely more defect-prone [24].
Process	Prior defects	Number of defects fixed prior to release.	Defects may linger in components that were recently defective [11].
	Churn	Sum of added and removed lines of code.	Components that have undergone a lot of change are likely defect-prone [29, 30].
	Change entropy	A measure of the volatility of the change process.	Components with a volatile change process, where changes are spread amongst several files are likely defect-prone [13].
Human Factors	Total authors	Number of unique authors.	Components with many unique authors likely lack strong ownership, which in turn may lead to more defects [5, 11].
	Minor authors	Number of unique authors who have contributed less than 5% of the changes.	Developers who make few changes to a component may lack the expertise required to perform the change in a defect-free manner [5]. Hence, components with many minor contributors are likely defect-prone.
	Major authors	Number of unique authors who have contributed at least 5% of the changes.	Similarly, components with a large number of major contributors, i.e., those with component-specific expertise are less likely to be defect-prone [5].
	Author ownership	The proportion of changes contributed by the author who made the most changes.	Components with a highly active component owner are less likely to be defect-prone [5].
Coverage (RQ1)	Proportion of reviewed changes	The proportion of changes that have been reviewed in the past.	Since code review will likely catch defects, components where changes are most often reviewed are less likely to contain defects.
	Proportion of reviewed churn	The proportion of churn that has been reviewed in the past.	Despite the defect-inducing nature of code churn, code review should have a preventative impact on defect-proneness. Hence, we expect that the larger the proportion of code churn that has been reviewed, the less defect prone a module will be.
Participation (RQ2)	Proportion of self-approved changes	The proportion of changes to a component that are only approved for integration by the original author.	By submitting a review request, the original author already believes that the code is ready for integration. Hence, changes that are only approved by the original author have essentially not been reviewed.
	Proportion of hastily reviewed changes	The proportion of changes that are approved for integration at a rate that is faster than 200 lines per hour.	Prior work has shown that when developers review more than 200 lines of code per hour, they are more likely to produce lower quality software [19]. Hence, components with many changes that are approved at a rate faster than 200 lines per hour are more likely to be defect-prone.
	Proportion of changes without discussion	The proportion of changes to a component that are not discussed.	Components with many changes that are approved for integration without critical discussion are likely to be defect-prone.

cant contributor to the fit of our models, we perform drop one tests [7] using the implementation provided by the core `stats` package of R [31]. The test measures the impact of an explanatory variable on the model by measuring the AIC of models consisting of: (1) all explanatory variables (the full model), and (2) all explanatory variables except for the one under test (the dropped model). A χ^2 test is applied to the resulting values to detect whether each explanatory variable improves the AIC of the model to a statistically significant degree. We discard the explanatory variables that do not improve the AIC by a significant amount ($\alpha = 0.05$).

Explanatory variable impact analysis. To study the impact that explanatory variables have on the incidence of post-release defects, we calculate the expected number of defects in a typical component using our models. First, an artificial component is simulated by setting all of the explanatory variables to their median values. The variable under test is then set to a specific value. The model is then applied to the artificial component and the *Predicted Defect Count* (PDC) is calculated, i.e., the number of defects that the model estimates to be within the artificial component.

Note that the MLR model may predict that a component has a negative or fractional number of defects. Since negative or fractional numbers of defects cannot exist in reality, we calculate the *Concrete Predicted Defect Count* (CPDC) as follows:

$$CPDC(x_i) = \begin{cases} 0, & \text{if } PDC(x_i) \leq 0 \\ \lceil PDC(x_i) \rceil, & \text{otherwise} \end{cases} \quad (1)$$

We take the ceiling of positive fractional PDC values rather than rounding so as to accurately reflect the worst-case concrete values. Finally, we use plots of CPDC values as we change the variable under test to evaluate its impact on post-release defect counts.

4. CASE STUDY RESULTS

In this section, we present the results of our case study with respect to our two research questions. For each question, we present the metrics that we use to measure the reviewing property, then discuss the results of adding those metrics to our MLR models.

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

Intuitively, one would expect that higher rates of code review coverage will lead to fewer incidences of post-release defects. To investigate this, we add the code review coverage metrics described in Table 2 to our MLR models.

Coverage metrics. The *proportion of reviewed changes* is the proportion of changes committed to a component that

Table 3: Review coverage model statistics. ΔAIC indicates the change in AIC when the given metric is removed from the model (larger ΔAIC values indicate more explanatory power). *Coef.* provides the coefficient of the given metric in our models.

	Qt				VTK		ITK	
	5.0.0		5.1.0		5.10.0		4.3.0	
Adjusted R^2	0.40		0.19		0.38		0.24	
Total AIC	4,853		6,611		219		15	
	Coef.	ΔAIC	Coef.	ΔAIC	Coef.	ΔAIC	Coef.	ΔAIC
Size		$\diamond$	0.46	6^{**}	0.19	223.4^{*}		$\diamond$
Complexity		$\diamond$		$\diamond$		$\diamond$		$\diamond$
Prior defects	5.08	106^{***}		$\diamond$	3.47	71^{***}	0.08	13^{***}
Churn		$\diamond$		$\diamond$		$\dagger$		$\dagger$
Change entropy		$\diamond$		$\diamond$		$\diamond$		$\diamond$
Total authors		$\ddagger$		$\dagger$		$\ddagger$		$\ddagger$
Minor authors	2.57	49^{***}	10.77	210^{***}	2.79	50^{***}	1.58	23^{***}
Major authors		$\dagger$		$\dagger$		$\dagger$		$\dagger$
Author ownership		$\diamond$		$\diamond$		$\diamond$		$\diamond$
Reviewed changes	-0.25	9^{***}		$\diamond$	-0.30	15^{***}		$\diamond$
Reviewed churn		$\dagger$		$\dagger$		$\dagger$		$\dagger$

$\dagger$ Discarded during correlation analysis ($|\rho| > 0.7$)
$\ddagger$ Discarded during VIF analysis (VIF coefficient > 5)
Statistical significance of explanatory power according to Drop One analysis:
$\diamond$ $p \geq 0.05$; * $p < 0.05$; ** $p < 0.01$; *** $p < 0.001$

are associated with code reviews. Similarly, *proportion of reviewed churn* is the proportion of the churn of a component that is associated with code reviews.

Table 3 shows that the proportion of reviewed churn is too highly correlated with the proportion of reviewed changes to include both metrics in the same model. We selected the proportion of reviewed changes for our models because it is a simpler metric. For the sake of completeness, we analyzed models that use the proportion of reviewed churn instead of the proportion of reviewed changes and found that it had no discernible impact on model performance.

Components with higher review coverage tend to have fewer post-release defects. Table 3 shows that the proportion of reviewed changes has a statistically significant impact in the defect models of Qt 5.0.0 and VTK 5.10.0. Even in the Qt 5.1.0 and ITK models (where the proportion of reviewed changes is removed due to a lack of explanatory power), its estimated coefficient is negative, indicating that an increase in review coverage tends to lower the incidence rate of post-release defects in a component.

Components with review coverage below 0.29 (VTK) or 0.6 (Qt) are expected to contain at least one post-release defect. Figure 3 shows the CPDC (*cf.* Equation 1) of a component with a varying proportion of reviewed changes. In other words, each point on the line indicates the expected number of post-release defects in a typical component due to a corresponding proportion of reviewed changes.

As shown in Figure 3, our models indicate that a typical Qt 5.0.0 component with a proportion of reviewed changes of less than 0.6 is expected to contain at least one post-release defect. Moreover, Qt 5.0.0 components with a proportion of reviewed changes of less than 0.06 are expected to have at least two post-release defects. To put this in perspective, a post-release defect count of two corresponds to the 89th percentile of the observed post-release defect counts in Qt 5.0.0, and the 40th percentile of Qt 5.0.0 components with at least one post-release defect.

Typical VTK 5.10.0 components are expected to contain one post-release defect if the proportion of reviewed changes drops below 0.29. Since VTK components with post-release

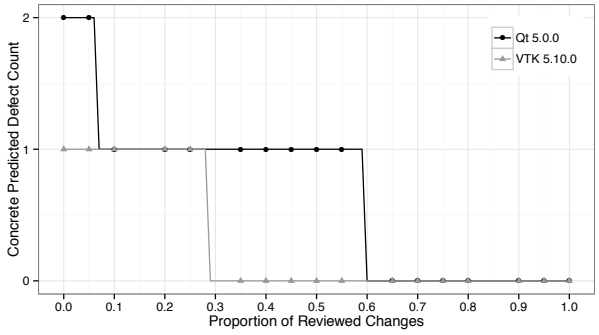

Figure 3: The predicted count of post-release defects in a typical component for various proportions of reviewed changes.

defects are relatively rare, a post-release defect count of one corresponds to the 92nd percentile of the observed post-release defect counts in VTK 5.10.0.

Other code review properties may provide additional explanatory power. While the proportion of reviewed changes is associated with components of higher software quality in two of the four studied releases, it does not have a significant impact on our Qt 5.1.0 and ITK models. To gain a richer perspective about the relationship between code review coverage and software quality, we manually inspect the Qt 5.0.0 components with the most post-release defects.

As our Qt 5.0.0 model suggests, the components with many post-release defects indeed tend to have lower proportions of reviewed changes. This is especially true for the collection of nine components that make up the `QtSerialPort` subsystem, where the proportion of reviewed changes does not exceed 0.1. Initial development of the `QtSerialPort` subsystem began during Qt 4.x, prior to the introduction of Gerrit to the Qt development process. Many foundational features of the subsystem were introduced in an incubation area of the Qt development tree, where reviewing policies are lax. Hence, much of the `QtSerialPort` code was likely

not code reviewed, which may have lead to the inflation in post-release defect counts.

On the other hand, there are components with a proportion of reviewed changes of 1 that still have post-release defects. Although only 7% of the VTK components with post-release defects (1/15) have a proportion of reviewed changes of 1, 87% (222/254), 70% (131/187), and 83% (20/24) of Qt 5.0.0, Qt 5.1.0, and ITK respectively have a proportion of reviewed changes of 1. We further investigate with one-tailed Mann-Whitney U tests ($\alpha = 0.05$) comparing the incidence of post-release defects in components with a proportion of reviewed changes of 1 to those components with proportions of reviewed change below 1. Test results indicate that only in Qt 5.1.0 is the incidence of post-release defects in components with proportions of reviewed changes of 1 significantly less than the incidence of post-release defects in components with proportions lower than 1 ($p < 2.2 \times 10^{-16}$). In the other systems, the difference is not significant ($p > 0.05$).

Although review coverage is negatively associated with software quality in our models, several defect-prone components have high coverage rates, suggesting that other properties of the code review process are at play.

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

As discussed in RQ1, even components with a proportion of reviewed changes of 1 (i.e., 100% code review coverage) can have high post-release defect rates. We suggest that a lack of participation in the code review process could be contributing to this. In fact, in thriving open source projects, such as the Linux kernel, insufficient discussion is one of the most frequently cited reasons for the rejection of a patch.[7] In recent work, Jiang *et al.* found that the amount of reviewing discussion is an important indicator of whether a patch will be accepted for integration into the Linux kernel [16]. To investigate whether code review participation has a measurable impact on software quality, we add the participation metrics described in Table 2 to our defect models.

Since we have observed that review coverage has an impact on post-release defect rates (RQ1), we need to control for the proportion of reviewed changes when addressing RQ2. We do so by selecting only those components with a proportion of reviewed changes of 1 for analysis. Although 90% (1,201/1,339) of the Qt 5.0.0, 88% (1,175/1,337) of the Qt 5.1.0, and 125/218 (57%) of the ITK components survive the filtering process, only 5% (8/170) of the VTK components survive. Since the VTK dataset is no longer large enough for statistical analysis, we omit it from this analysis.

Participation metrics. We describe the three metrics that we have devised to measure code review participation below. The *proportion of self-approved changes* is the proportion of changes that have only been approved for integration by the original author of the change.

An appropriate amount of time should be allocated in order to sufficiently critique a proposed change. Best practices suggest that code should be not be reviewed at a rate faster than 200 lines per hour [19]. Therefore, if the time window between the creation of a review request and its approval for integration is shorter than this, the review is likely suboptimal. The *proportion of hastily reviewed changes* is the proportion of changes that have been reviewed at a rate faster than 200 lines per hour. Since our definition of hastily reviewed changes assumes that reviewers begin reviewing a change as soon as it is assigned to them, our metric represents a lower bound of the actual proportion. We discuss the further implications of this definition in Section 5.

[7] https://www.kernel.org/doc/Documentation/SubmittingPatches

Table 4: Review participation model statistics. ΔAIC indicates the change in AIC when the given metric is removed from the model (larger ΔAIC values indicate more explanatory power). *Coef.* indicates whether the coefficient of the given metric is positive or negative.

	Qt				ITK	
	5.0.0		5.1.0		4.3.0	
Adjusted R^2	0.44		0.26		0.25	
Total AIC	4,328		1,639		71	
	Coef.	ΔAIC	Coef.	ΔAIC	Coef.	ΔAIC
Size		◇	0.08	4*		◇
Complexity		◇		◇		◇
Prior defects	4.20	68***	0.95	28***	0.70	5*
Churn		†		◇		†
Change entropy		◇		◇		◇
Total authors		‡		†		‡
Minor authors	2.06	24***	3.22	85***	1.01	8**
Major authors		†		†		†
Author ownership		†		†		◇
Self-approval	1.34	11***		◇		◇
Hastily reviewed		◇	0.55	8**		◇
No discussion	0.83	4*	0.74	15***	0.70	4*

† Discarded during correlation analysis ($|\rho| > 0.7$)
‡ Discarded during VIF analysis (VIF coefficient > 5)
Statistical significance of explanatory power according to Drop One analysis:
◇ $p \geq 0.05$; * $p < 0.05$; ** $p < 0.01$; *** $p < 0.001$

Reviews without accompanying discussion have not received critical analysis from other members of the development team, and hence may be prone to defects that a more thorough critique could have prevented. The operational definition that we use for a review without discussion is a patch that has been approved for integration, yet does not have any attached comments from other team members. Since our intent is to measure team discussion, we ignore comments generated by automated verifiers (e.g., CI systems), since they do not create a team dialogue. Finally, the *proportion of changes without discussion* is calculated as the proportion of changes that have been approved for integration without discussion.

Table 4 describes the results of our model construction experiment. Although our code review participation models achieve better adjusted R^2 and AIC scores than the code review coverage models do, a comparison between the two should not be drawn, since the participation models are built using a subset of the system components.

Components with high rates of participation in code review tend to have fewer post-release defects. Table 4 shows that the proportion of changes without discussion has a statistically significant impact on the models of all three of the studied releases. Furthermore, the proportion of self-approved changes has a significant impact on the Qt 5.0.0 model and the proportion of hastily reviewed changes has a significant impact on the Qt 5.1.0 model. The estimated coefficients are positive in all cases, indicating that the components that integrate more insufficiently discussed, hastily reviewed, and/or self-approved patches tend to be more defect-prone.

Conversely, components with low participation rates in code review tend to have high post-release defect counts. Figure 4 shows that Qt 5.0.0 components with a proportion of self-approved changes of 0.84 or higher are es-

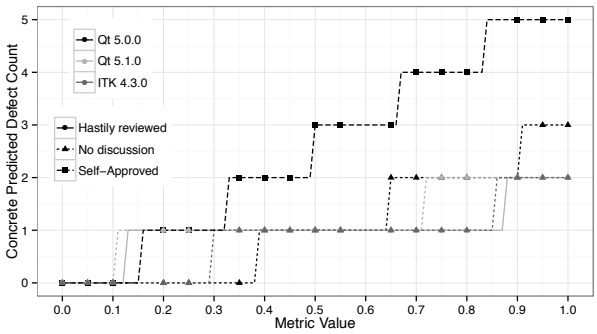

Figure 4: The predicted count of post-release defects in a component for varying participation rates.

timated to contain five additional post-release defects. To put this in perspective, a post-release defect count of five corresponds to the 95th percentile of the observed post-release defect counts in Qt 5.0.0, and the 70th percentile of Qt 5.0.0 components with at least one post-release defect. Components where the proportion of changes without discussion is above 0.71 are estimated to have at least two post-release defects in both of the studied Qt releases, while those Qt 5.0.0 components with a proportion above 0.9 are estimated to have at least three post-release defects.

Manual analysis of the data reveals that the several Qt components that provide backwards compatibility for Qt 4 APIs (e.g., `qt4support`) have a proportion of changes without discussion above 0.9. Perhaps this is due to a shift in team focus towards newer functionality. However, our results suggest that changes to these components should also be reviewed actively.

Our models also indicate that Qt components quickly become defect-prone when review participation decreases. Either the proportion of hastily reviewed changes or the proportion of changes without discussion need only reach 0.1 and 0.13 respectively before our Qt 5.1.0 model expects that a component will contain a post-release defect. Similarly, the proportion of self-approved changes need only reach 0.16 in Qt 5.0.0 before our model anticipates a post-release defect.

Lack of participation in code review has a negative impact on software quality. Reviews without discussion are associated with higher post-release defect counts, suggesting that the amount of discussion generated during review should be considered when making integration decisions.

5. THREATS TO VALIDITY

External validity. We focus our study on three open source systems, due to the low number of systems that satisfied our eligibility criteria for analysis. The proportion of commits that underwent code review through Gerrit presented a major challenge. Nonetheless, additional replication studies are needed.

Construct validity. Our models assume that each post-release defect is of the same weight, while in reality it may be that some post-release defects are more severe than others. Although modern Issue Tracking Systems (ITS) provide a field for practitioners to denote the priority and severity of a defect, recent work suggests that these fields are rarely accurate. For example, Herraiz *et al.* argue that the severity levels offered by the Eclipse bug reporting tool do not agree with clusters of defects that form based on the time taken to deliver a fix [15]. Indeed, Mockus *et al.* find that the recorded priority in Apache and Mozilla projects was not related to the time taken to resolve an issue, largely because the reporters who file the defects had far less experience than the core developers who fix them [26]. Nonetheless, each defect that we consider as a quality-impacting post-release defect was at least severe enough to warrant a fix that was integrated into the strictly controlled `release` branches of the studied systems.

Internal validity. We assume that a code review has been rushed if the elapsed time between the time that a patch has been uploaded and the time that it has been approved is shorter than the amount of time that should have been spent if the reviewer was digesting 200 lines of code per hour. However, there are likely cases where reviewers do not start reviewing the change immediately, but rush their review on a later date. Unfortunately, since reviewers do not record the time that they actually spent reviewing a patch, we must rely on heuristics to recover this information. On the other hand, our heuristic is highly conservative, i.e., reviews that are flagged as rushed are certainly rushed. Furthermore, setting the reviewing speed threshold to 100 lines per hour had little impact on our models.

Since there is an inherent delay between the code review (and integration) of a change and its appearance in a release, confounding factors could influence our results. However, our conclusions are intuitive, i.e., lax reviewing practices could allow defects to permeate through to the release.

6. RELATED WORK

In this section, we discuss the related work with respect to code review and software quality dimensions.

Code reviews. Prior work has qualitatively analyzed the modern code review process used by large software systems. Rigby *et al.* find that the Apache project adopted a broadcast-based style of code review, where frequent reviews of small and independent changes were in juxtaposition to the formal code inspection style prescribed by prior research, yet were still able to achieve a high level of software quality [35]. In more recent work, Rigby and Storey find that open source developers that adopt the broadcast-based code review style actively avoid discussions in reviews about opinionated and trivial patch characteristics [36]. In our work, we find that active participation in the code review process tends to reduce post-release counts and improve software quality.

The identification of defects is not the sole motivation for modern code review. For example, Rigby and Storey show that non-technical issues are a frequent motivation for the patch rejection in several open source systems [36]. Indeed, Baysal *et al.* find that *review positivity*, i.e., the proportion of accepted patches, is also influenced by non-technical factors [3]. Furthermore, a recent qualitative study at Microsoft indicates that sharing knowledge among team members is also considered a very important motivation of modern code review [2]. Inspired by these studies, we empirically analyze the relationship between developer investment in the code review process and software quality.

Kemerer and Faulk show that the introduction of design and code review to student projects at the SEI leads to code that is of higher quality [19]. By studying student projects, Kemerer and Faulk are able to control for several confound-

ing factors like team dynamics. Rather than control for team dynamics, our study aims to complement prior work by examining the impact of participation in the code review process of three large open source systems.

Software quality. There are many empirical studies that propose software metrics to predict software quality. For example, Hassan proposes complexity metrics (e.g., change entropy used in our paper) that are based on the code change process instead of on the code [13]. He shows that the entropy of the code change process is a good indicator of defect-prone source code files. Rahman and Devanbu built defect prediction models to compare the impact of product and process metrics [33]. They show that product metrics are generally less useful than process metrics for defect prediction. Through a case study of Eclipse, Kamei *et al.* also find that process metrics tend to outperform product metrics when software quality assurance effort is considered [17]. In this paper, our focus is on explaining the impact that code review coverage and participation have on software quality, rather than predicting it. Hence, we build models to study whether metrics that measure code review coverage and participation add unique information that helps to explain incidence rates of post-release defects.

Recent work studies the relationship between source code ownership and software quality. Bird *et al.* find that ownership measures have a strong relationship with both pre- and post-release defect-proneness. Matsumoto *et al.* show that their proposed ownership measures (e.g., the number of developers and the code churn generated by each developer) are also good indicators of defect-prone source code files [22]. Rahman and Devanbu find that lines of code that are implicated in a fix for a defect are more strongly associated with single developer contributions, suggesting that code review is a crucial part of the software quality assurance [32]. We find that the code ownership metrics that we adopt in the baseline analysis of the studied systems are very powerful, contributing a statistically significant amount of explanatory power to each of the defect models that we built.

7. CONCLUSIONS

Although code reviewing is a broadly endorsed best practice for software development, little work has empirically evaluated the impact that properties of the modern code review process have on software quality in large software systems. With the recent emergence of modern code reviewing tools like Gerrit, high quality data is now becoming available to enable such empirical studies.

The lightweight nature of modern code review processes relaxes the strict criteria of the formal code inspections that were mandated to ensure that a basic level of review participation was achieved (e.g., in-person meetings and reviewer checklists). In this paper, we quantitatively investigate three large software systems using modern code review tools (i.e., Gerrit). We build and analyze MLR models that explain the incidence of post-release defects in the components of these systems. Specifically, we evaluate the conjecture that:

> *If a large proportion of the code changes that are integrated during development are either: (1) omitted from the code review process (low review coverage), or (2) have lax code review involvement (low review participation), then defect-prone code will permeate through to the released software product.*

The results of our case study indicate that:

- Code review coverage metrics only contribute a significant amount of explanatory power to two of the four defect models when we control for several metrics that are known to be good explainers of software quality.
- Two of the three code review participation metrics contribute significant amounts of explanatory power to the defect models of each of the studied Qt releases.
- Components with low review participation are estimated to contain up to five additional post-release defects.

We believe that our findings provide strong empirical evidence to support the design of modern code integration policies that take code review coverage and participation into consideration. Our models suggest that such policies will lead to higher quality, less defect-prone software.

Future work. Although code review coverage tends to improve software quality in general, there are still many components with high review coverage rates that suffer from poor quality. This suggests that there are other properties of the code review process at play. In this paper, we study participation, but there are several other code review properties that are ripe for exploration. For example, code ownership metrics are strong indicators of defect-prone code [5, 32]. However, these metrics are calculated based only on version control repositories. We are actively exploring the impact of an expansion of the scope of the code ownership concept to include data from code review processes.

8. ACKNOWLEDGMENTS

This research was partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) and JSPS KAKENHI Grant Numbers 24680003 and 25540026.

9. REFERENCES

[1] H. Akaike. A New Look at the Statistical Model Identification. *Transactions on Automatic Control (TAC)*, 19(6):716–723, 1974.

[2] A. Bacchelli and C. Bird. Expectations, Outcomes, and Challenges of Modern Code Review. In *Proc. of the 35th Int'l Conf. on Software Engineering (ICSE)*, pages 712–721, 2013.

[3] O. Baysal, O. Kononenko, R. Holmes, and M. W. Godfrey. The Influence of Non-technical Factors on Code Review. In *Proc. of the 20th Working Conf. on Reverse Engineering (WCRE)*, pages 122–131, 2013.

[4] N. Bettenburg, A. E. Hassan, B. Adams, and D. M. German. Management of community contributions: A case study on the Android and Linux software ecosystems. *Empirical Software Engineering*, To appear, 2014.

[5] 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 8th joint meeting of the European Software Engineering Conf. and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 4–14, 2011.

[6] M. Cataldo, A. Mockus, J. A. Roberts, and J. D. Herbsleb. Software Dependencies, Work Dependencies, and Their Impact on Failures. *Transactions on Software Engineering (TSE)*, 35(6):864–878, 2009.

[7] J. M. Chambers and T. J. Hastie, editors. *Statistical Models in S*, chapter 4. Wadsworth and Brooks/Cole, 1992.

[8] M. E. Fagan. Design and Code Inspections to Reduce Errors in Program Development. *IBM Systems Journal*, 15(3):182–211, 1976.

[9] J. Fox. *Applied Regression Analysis and Generalized Linear Models*. Sage Publications, 2nd edition, 2008.

[10] J. Fox and S. Weisberg. *An R Companion to Applied Regression*. Sage, 2nd edition, 2011.

[11] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy. Predicting Fault Incidence using Software Change History. *Transactions on Software Engineering (TSE)*, 26(7):653–661, 2000.

[12] K. Hamasaki, R. G. Kula, N. Yoshida, A. E. C. Cruz, K. Fujiwara, and H. Iida. Who Does What during a Code Review? Datasets of OSS Peer Review Repositories. In *Proc. of the 10th Working Conf. on Mining Software Repositories (MSR)*, pages 49–52, 2013.

[13] A. E. Hassan. Predicting Faults Using the Complexity of Code Changes. In *Proc. of the 31st Int'l Conf. on Software Engineering (ICSE)*, pages 78–88, 2009.

[14] T. Hastie, R. Tibshirani, and J. Friedman. *Elements of Statistical Learning*. Springer, 2nd edition, 2009.

[15] I. Herraiz, D. M. German, J. M. Gonzalez-Barahona, and G. Robles. Towards a Simplification of the Bug Report form in Eclipse. In *Proc. of the 5th Working Conf. on Mining Software Repositories (MSR)*, pages 145–148, 2008.

[16] Y. Jiang, B. Adams, and D. M. German. Will My Patch Make It? And How Fast?: Case Study on the Linux Kernel. In *Proc. of the 10th Working Conf. on Mining Software Repositories (MSR)*, pages 101–110, 2013.

[17] Y. Kamei, S. Matsumoto, A. Monden, K. ichi Matsumoto, B. Adams, and A. E. Hassan. Revisiting Common Bug Prediction Findings Using Effort-Aware Models. In *Proc. of the 26th Int'l Conf. on Software Maintenance (ICSM)*, pages 1–10, 2010.

[18] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi. A Large-Scale Empirical Study of Just-in-Time Quality Assurance. *Transactions on Software Engineering (TSE)*, 39(6):757–773, 2013.

[19] 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)*, 35(4):534–550, 2009.

[20] S. Kim, E. J. Whitehead, Jr., and Y. Zhang. Classifying software changes: Clean or buggy? *Transactions on Software Engineering (TSE)*, 34(2):181–196, 2008.

[21] A. G. Koru, D. Zhang, K. E. Emam, and H. Liu. An Investigation into the Functional Form of the Size-Defect Relationship for Software Modules. *Transactions on Software Engineering (TSE)*, 35(2):293–304, 2009.

[22] S. Matsumoto, Y. Kamei, A. Monden, K. ichi Matsumoto, and M. Nakamura. An analysis of developer metrics for fault prediction. In *Proc. of the 6th Int'l Conf. on Predictive Models in Software Engineering (PROMISE)*, pages 18:1–18:9, 2010.

[23] T. J. McCabe. A complexity measure. In *Proc. of the 2nd Int'l Conf. on Software Engineering (ICSE)*, page 407, 1976.

[24] T. Menzies, J. S. D. Stefano, M. Chapman, and K. McGill. Metrics That Matter. In *Proc of the 27th Annual NASA Goddard/IEEE Software Engineering Workshop*, pages 51–57, 2002.

[25] A. Mockus. Organizational Volatility and its Effects on Software Defects. In *Proc. of the 18th Symposium on the Foundations of Software Engineering (FSE)*, pages 117–126, 2010.

[26] A. Mockus, R. T. Fielding, and J. D. Herbsleb. Two Case Studies of Open Source Software Development: Apache and Mozilla. *Transactions On Software Engineering and Methodology (TOSEM)*, 11(3):309–346, 2002.

[27] A. Mockus and D. M. Weiss. Predicting Risk of Software Changes. *Bell Labs Technical Journal*, 5(2):169–180, 2000.

[28] M. Mukadam, C. Bird, and P. C. Rigby. Gerrit Software Code Review Data from Android. In *Proc. of the 10th Working Conf. on Mining Software Repositories (MSR)*, pages 45–48, 2013.

[29] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *Proc. of the 27th Int'l Conf. on Software Engineering (ICSE)*, pages 284–292, 2005.

[30] N. Nagappan and T. Ball. Using Software Dependencies and Churn Metrics to Predict Field Failures: An Empirical Case Study. In *Proc. of the 1st Int'l Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 364–373, 2007.

[31] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2013.

[32] F. Rahman and P. Devanbu. Ownership, Experience and Defects: A Fine-Grained Study of Authorship. In *Proc. of the 33rd Int'l Conf. on Software Engineering (ICSE)*, pages 491–500, 2011.

[33] F. Rahman and P. Devanbu. How, and why, process metrics are better. In *Proc. of the 35th Int'l Conf. on Software Engineering (ICSE)*, pages 432–441, 2013.

[34] P. C. Rigby and C. Bird. Convergent Contemporary Software Peer Review Practices. In *Proc. of the 9th joint meeting of the European Software Engineering Conf. and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 202–212, 2013.

[35] P. C. Rigby, D. M. German, and M.-A. Storey. Open Source Software Peer Review Practices: A Case Study of the Apache Server. In *Proc. of the 30th Int'l Conf. on Software Engineering (ICSE)*, pages 541–550, 2008.

[36] P. C. Rigby and M.-A. Storey. Understanding Broadcast Based Peer Review on Open Source Software Projects. In *Proc. of the 33rd Int'l Conf. on Software Engineering (ICSE)*, pages 541–550, 2011.

[37] E. Shihab, Z. M. Jiang, W. M. Ibrahim, B. Adams, and A. E. Hassan. Understanding the Impact of Code and Process Metrics on Post-Release Defects: A Case Study on the Eclipse Project. In *Proc. of the 4th Int'l Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–10, 2010.

[38] E. Shihab, A. Mockus, Y. Kamei, B. Adams, and A. E. Hassan. High-Impact Defects: A Study of Breakage and Surprise Defects. In *Proc. of the 8th joint meeting of the European Software Engineering Conf. and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 300–310, 2011.

[39] T. Tanaka, K. Sakamoto, S. Kusumoto, K. ichi Matsumoto, and T. Kikuno. Improvement of Software Process by Process Description and Benefit Estimation. In *Proc. of the 17th Int'l Conf. on Software Engineering (ICSE)*, pages 123–132, 1995.