Revisiting Code Ownership and its Relationship with Software Quality in the Scope of Modern Code Review

Patanamon Thongtanunam[1], Shane McIntosh[2], Ahmed E. Hassan[3], Hajimu Iida[1]

[1]Nara Institute of Science and Technology, Japan
patanamon-t@is.naist.jp, iida@itc.naist.jp

[2]McGill University, Canada
shane.mcintosh@mcgill.ca

[3]Queen's University, Canada
ahmed@cs.queensu.ca

ABSTRACT

Code ownership establishes a chain of responsibility for modules in large software systems. Although prior work uncovers a link between code ownership heuristics and software quality, these heuristics rely solely on the authorship of code changes. In addition to authoring code changes, developers also make important contributions to a module by reviewing code changes. Indeed, recent work shows that reviewers are highly active in modern code review processes, often suggesting alternative solutions or providing updates to the code changes. In this paper, we complement traditional code ownership heuristics using code review activity. Through a case study of six releases of the large Qt and OpenStack systems, we find that: (1) 67%-86% of developers did not author any code changes for a module, but still actively contributed by reviewing 21%-39% of the code changes, (2) code ownership heuristics that are aware of reviewing activity share a relationship with software quality, and (3) the proportion of reviewers without expertise shares a strong, increasing relationship with the likelihood of having post-release defects. Our results suggest that reviewing activity captures an important aspect of code ownership, and should be included in approximations of it in future studies.

CCS Concepts

•**Software and its engineering** → *Maintaining software;* Programming teams;

Keywords

Ownership, Expertise, Software Quality

1. INTRODUCTION

Code ownership is an important concept for large software teams. In large software systems, with hundreds or even thousands of modules, code ownership is used to establish a chain of responsibility. When changes need to be made to a module with strong code ownership, there is a module owner who is responsible for it. On the other hand, it is more difficult to identify the developer to whom tasks should be delegated in modules with weak code ownership.

In the literature, code ownership is typically estimated using heuristics that are derived from code authorship data. For example, Mockus and Herbsleb use the number of tasks that a developer has completed within a time window that modify a given module to identify the developer that is responsible for that module [33]. Furthermore, Bird *et al.* estimate a code ownership value for a developer within a module by computing the proportion of code changes that the developer has authored within that module [7]. Rahman and Devanbu estimate code ownership values at a finer granularity by computing the proportion of lines of changed code that each developer has authored [38]. Indeed, the intuition behind traditional code ownership heuristics is that developers who author the majority of changes to a module are likely to be the owners of those modules.

However, in addition to authorship contributions, developers can also contribute to the evolution of a module by critiquing code changes that other developers have authored in that module. Indeed, many contemporary software development teams use Modern Code Review (MCR), a tool-based code review process, which tightly integrates with the software development process [39]. In the MCR process, code changes are critiqued by reviewers (typically other developers) to ensure that such code changes are of sufficient quality prior to their integration with the codebase.

While several studies have shown that the modern code review activities share a link to software quality [25, 29, 50], the MCR process is more than just a defect-hunting exercise. Indeed, Morales *et al.* show that developer involvement in the MCR process shares a relationship with software design quality [34]. Bacchelli and Bird report that the focus of code review at Microsoft has shifted from being defect-driven to collaboration-driven [1]. Others report that reviewers in MCR processes often suggest alternative solutions [4, 49, 52] or even provide updates to the code changes themselves [50].

Despite the active role that reviewers play in the MCR process, review contributions are disregarded by traditional code ownership heuristics. For example, a senior developer who reviews many of the code changes to a module, while authoring relatively few will not be identified as a module owner by traditional code ownership heuristics. Therefore, in this paper, we set out to complement traditional code ownership heuristics using data that is derived from code review repositories. More specifically, we adapt the popular code ownership heuristics of Bird *et al.* [7] to be: (1) *review-*

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

DOI: http://dx.doi.org/10.1145/2884781.2884852

specific, i.e., solely derived from code review data and (2) *review-aware*, i.e., combine code authorship and review activity. We then use review-specific and review-aware code ownership heuristics to build regression models that classify modules as defect-prone or not in order to revisit the relationship between the code ownership and software quality. Through a case study of six releases of the large Qt and OpenStack open source systems, we address the following three research questions:

(RQ1) How do code authoring and reviewing contributions differ?
Motivation: While prior work examines the contributions of developers in terms of code authorship [7, 12, 13, 32, 37, 38], the review contributions that these developers make to a module still remains largely unexplored. Hence, we first study how developers contribute to the evolution of a module in terms of code authorship and review.
Results: 67%-86% of the developers who contribute to a module did not author any code changes, yet they participated in the reviews of 21%-39% of the code changes made to that module. Moreover, 18%-50% of these review-only contributors are documented core team members.

(RQ2) Should code review activity be used to refine traditional code ownership heuristics?
Motivation: Prior work finds that modules with many minor authors (i.e., developers who seldom wrote code changes in that module) are likely to be defective [7]. However, there are likely cases where minor authors review several of the code changes in that module. Hence, we investigate whether refining traditional code ownership heuristics by using code review activity will provide a more comprehensive picture of the association between code ownership and software quality.
Results: 13%-58% of developers who are flagged as minor contributors of a module by traditional code ownership heuristics are actually major contributors when their code review activity is taken into consideration. Moreover, modules without post-release defects tend to have a large proportion of developers who have low traditional code ownership values but high review-specific ownership values. Conversely, the modules with post-release defects tend to have a large proportion of developers who have both low traditional and review-specific ownership values.

(RQ3) Is there a relationship between review-specific and review-aware code ownership heuristics and defect-proneness?
Motivation: Prior work uses defect models to understand the relationship between traditional code ownership heuristics and software quality [7, 9, 14, 35]. Such an understanding of defect-proneness is essential to chart quality improvement plans, e.g., developing code ownership policies. Yet, since traditional code ownership heuristics do not include code review activity, little is known about the role that reviewing plays in ownership and its impact on defect-proneness. Hence, we revisit the relationship between code ownership and software quality using review-specific and review-aware heuristics.

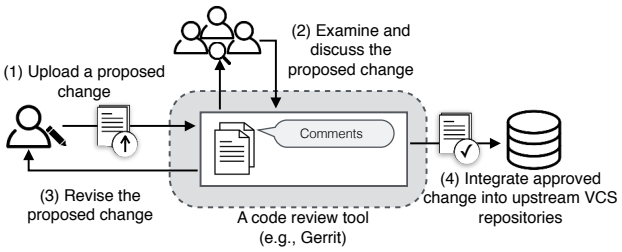


Figure 1: An overview of the MCR process.

Results: When controlling for several factors that are known to share a relationship with defect-proneness, our models show that the proportion of developers who have both low traditional and review ownership values shares a strong, increasing relationship with the likelihood of having post-release defects.

Our results lead us to conclude that code reviewing activity provides an important perspective that contributes to the code ownership concept. Future approximations of code ownership should take code reviewing activity into account in order to more accurately model the contributions that developers make to evolve modern software systems.

Paper organization. Section 2 describes the MCR process and the studied code ownership heuristics in more detail. Section 3 describes the design of our case study, while Section 4 presents the results with respect to our three research questions. Section 5 discusses the broader implications of our observations. Section 6 discloses the threats to the validity of our study. Section 7 surveys related work. Finally, Section 8 draws conclusions.

2. BACKGROUND & DEFINITION

In this section, we provide a brief description of the MCR process and the studied code ownership heuristics.

2.1 Modern Code Review

Modern Code Review (MCR) is a tool-based code review process [1], which tightly integrates with the software development process. MCR is now widely adopted by proprietary and open source software organizations [39]. Recent studies have shown that MCR actively shares a relationship with post-release defects [29], the quality of code changes [1, 49, 52], and software design quality [34]. Furthermore, MCR provides a hub-like mechanism for sharing knowledge among developers [39].

Figure 1 provides an overview of MCR processes in the studied Qt and OpenStack systems. The studied processes are composed of four main steps:

(1) Upload a proposed change. Once a proposed change has been made, the author uploads it to a code review tool and invites reviewers to examine it.

(2) Examine and discuss the proposed change. Reviewers examine the technical content of the proposed change and provide feedback to the author. Reviewers also provide a review score from -2 to $+2$ to indicate agreement (positive value) or disagreement (negative value).

(3) Revise the proposed change. The author or reviewers update the proposed change to address the feedback, and upload a new revision to the code review tool.

(4) Integrate the approved change. Steps 2 and 3 are repeated until reviewers are satisfied that the code change is of sufficient quality to be integrated. Then, the proposed change is automatically integrated into upstream Version Control System (VCS, e.g., Git).

2.2 Code Ownership

Code ownership heuristics that operate at different granularities have been proposed in the literature [7, 13, 14, 33, 38]. For example, Bird *et al.* estimate code ownership values using the code change granularity [7]. Fritz *et al.* propose code ownership heuristics at element (i.e., class) granularity [13]. Rahman and Devanbu estimate code ownership values at the finer granularity of a code fragment. Since the studied MCR processes are connected with code changes, i.e., proposed changes should be reviewed prior to integration into the VCS, we opt to extend the code ownership heuristics of Bird *et al.* [7] using code review activity.

Below, we describe the traditional, review-specific, and review-aware code ownership heuristics.

2.2.1 Traditional Code Ownership Heuristics

The traditional code ownership heuristics of Bird *et al.* [7] are computed using the authorship of code changes to estimate the code ownership of a developer for a module. For a developer D, **Traditional Code Ownership (TCO)** of a module M is computed as follows:

$$\text{TCO}(D, M) = \frac{a(D, M)}{C(M)} \quad (1)$$

where $a(D, M)$ is the number of code changes that D has authored in M and $C(M)$ is the total number of code changes made to M.

In addition, Bird *et al.* also define two levels of developer expertise within a module. Developers with low TCO values (i.e., below 5%) are considered to be *minor authors*, while developers with high TCO values (i.e., above 5%) are considered to be *major authors*.

2.2.2 Review-Specific Ownership Heuristics

We define review-specific ownership heuristics that use code review data to estimate the code ownership of developers according to the review contributions that they have made to a module. For a developer D, **Review-Specific Ownership (RSO)** of a module M is computed as follows:

$$\text{RSO}(D, M) = \frac{\sum_{k=1}^{r(D,M)} p(D, k)}{C(M)} \quad (2)$$

where $r(D, M)$ is the number of reviews of code changes made to M in which D has participated and $p(D, k)$ is a proportion of review contributions that D made to code change k. Since one review can have many reviewers, we normalize the value of review-specific ownership in each code change using $p(D, k)$ in order to avoid having RSO values that do not sum up to 100%. We explore two RSO heuristics by varying the definition of $p(D, k)$:

1. RSO_{Even}, where $p(D, k) = \frac{1}{R(k)}$: This normalization evenly divides the share of review contributions to every reviewer of k. Hence, $R(k)$ is the total number of developers who participated in the review of k. This normalization assumes that every reviewer contributes equally to k.

Table 1: A contingency table of the review-aware ownership expertise category.

		Traditional code ownership	
		$\leq 5\%$	$> 5\%$
Review-specific ownership	$\leq 5\%$	Minor author & minor reviewer	Major author & minor reviewer
	$> 5\%$	Minor author & major reviewer	Major author & major reviewer

2. $\text{RSO}_{\text{Proportional}}$, where $p(D, k) = F(D, k)$: This normalization assigns a share of the review contribution to D that is proportional to the amount of feedback (i.e., number of review comments) that D has provided to the review of k. The intuition behind this normalization is that the more the feedback that D provides during the review of k, the larger the share of review contributions that D made to k.

Similar to traditional code ownership, we consider developers with low RSO values (i.e., below 5%) to be *minor reviewers*, and developers with high RSO values (i.e., above 5%) to be *major reviewers*.

2.2.3 Review-Aware Ownership Heuristics

In practice, developers act as both authors and reviewers. For example, a developer can be the author of a code change to a module, while also being a reviewer of code changes that other developers have authored to that module. However, the traditional and review-specific ownership heuristics independently estimate code ownership using either authorship or review contributions, respectively.

To address this, we propose review-aware code ownership heuristics. We adapt the traditional code ownership heuristics to be review-aware using the pair of TCO and RSO values that a developer can have within a module. Then, we refine the levels of the traditional code ownership using the levels of review-specific code ownership as shown in Table 1. For example, for a module M, if a developer D has a TCO value of 3% and an RSO value of 25%, then D would have a review-aware ownership value of $(3\%, 25\%)$, which falls into the minor author & major reviewer category.

3. CASE STUDY DESIGN

In this section, we outline our criteria for selecting the studied systems and our data preparation approach.

3.1 Studied Systems

In order to address our research questions, we perform an empirical study on large, rapidly-evolving open source systems. In selecting the subject system, we identified two important criteria that needed to be satisfied:

Criterion 1: Traceability — We focus our study on systems where the MCR process records explicit links between code changes and the associated reviews.

Criterion 2: Full Review Coverage — Since we will investigate the effect of review-specific and review-aware ownership heuristics on software quality, we need to ensure that unreviewed changes are not a confounding factor [29]. Hence, we focus our study on systems that have a large number of modules with 100% review coverage, i.e., modules where every code change made to them has undergone a code review.

Table 2: Overview of the studied systems. Those above the double line satisfy our criteria for analysis.

System			Commits		Modules			Personnel	
Name	Version	Tag name	Total	Linkage rate	Total	With 100% review coverage	With defects	Authors	Reviewers
Qt	5.0	v5.0.0	2,955	95%	389	328 (84%)	70 (21%)	156	156
	5.1	v5.1.0	2,509	96%	450	438 (97%)	77 (18%)	186	170
OpenStack	Folsom	2012.2	2,315	99%	258	241 (93%)	70 (29%)	235	152
	Grizzly	2013.1	2,881	99%	336	326 (97%)	123 (37%)	330	205
	Havana	2013.2	3,583	99%	527	515 (97%)	128 (25%)	451	359
	Icehouse	2014.1	3,021	100%	499	499 (100%)	198 (40%)	499	480
VTK	5.10	v5.10.0	1,431	39%	170	8 (5%)	-	-	-
ITK	4.0	v4.3.0	352	97%	218	125 (57%)	-	-	-

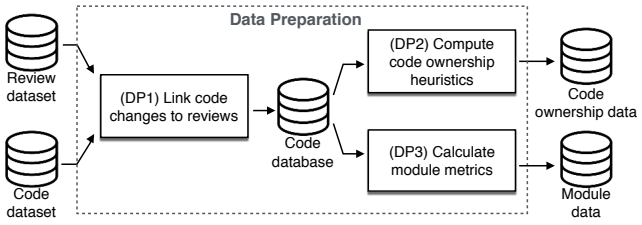


Figure 2: An overview of data preparation approach.

To satisfy criterion 1, we began our study with four software systems that use the Gerrit code review tool. To mitigate bias that may be introduced into our datasets through noisy manual linkage [5], we select systems that use the Gerrit code reviewing tool for our analysis. Gerrit automatically records a unique ID that can link code changes to the reviews that they have undergone. We then discard VTK, since its linkage rate is too low. We also remove ITK from our analysis, since it does not satisfy criterion 2.

Table 2 shows that the Qt and OpenStack systems satisfy our criteria for analysis. Qt is a cross-platform application and UI framework that is developed by the Digia corporation, while welcoming contributions from the community-at-large.[1] OpenStack is an open-source software platform for cloud computing that is developed by many well-known companies, e.g., IBM, VMware, and NEC.[2]

3.2 Data Preparation

We use the review datasets that are provided by Hamasaki *et al.* [16], which describes patch information, reviewer scoring, the involved personnel, and review discussion. We use the code dataset for the Qt system provided by our prior work [29], which describes the recorded commits on the `release` branch of the Qt VCSs during the development and maintenance of each studied Qt release. We also expand the code dataset for the OpenStack system using the same approach as our prior work [29].

In order to produce the datasets that are necessary for our study, we link the review and code datasets, and compute the code ownership heuristics. Figure 2 provides an overview of our data preparation process, which is broken down into the following three steps.

(DP1) Link code changes to reviews. Similar to our prior work [29], we link the code and review datasets using the change ID, i.e., a review reference that is automatically generated by Gerrit. For each code change, we extract the change ID recorded in the commit message. To link the associated review, Gerrit uses "<*subsystem name*>_<*VCS branch*>_<*change ID*>" as a unique reference. Hence, we extract the commit data from the VCSs to generate a reference, then link the code change to the associated review.

Once the code and review datasets are linked, we measure review coverage for each module (i.e., directory). Since this study revisits the traditional code ownership heuristics [7] which were previously studied at the module level, we conduct our study at the same module level to aid the comparability of our results with the prior work. We then remove modules that do not have 100% review coverage from our datasets in order to control for the confounding effect that a lack review coverage may have [29].

(DP2) Compute code ownership heuristics. To estimate the code ownership of a developer for a module, we first identify code changes that the developer has authored using the owner field recorded in Gerrit. We then identify the code changes that the developer has reviewed using the review comments that the developer posted. A code change that spans multiple modules is treated as contributing to all of the changed modules.

Next, we estimate code ownership using the traditional, review-specific, and review-aware code ownership heuristics (see Section 2.2) for every developer who was involved with the development of each module. Similar to prior work [22, 30], we use a six-month period prior to each release date to capture the authorship and review contributions that developers made during the development cycle.

(DP3) Calculate module metrics. Prior work has found that several types of metrics have an impact on software quality. Hence, we also measure popular product and process metrics that are known to have a relationship with defect-proneness in order to control for their impact [20, 47].

For the product metrics, we measure the *size* of the source code in a module at the time of a release by aggregating the number of lines of code in each of its files. For the process metrics, we use churn and entropy to measure the change activity that occurred during the development cycle of a studied release. We again use use the six-month period prior to each release date to capture the change activity. *Churn* counts the total number of lines added to and removed from a module prior to release. *Entropy* measures how the complexity of a change process is distributed across source code files [20]. We measure the entropy of a module using a calculation of $H(M) = -\frac{1}{log_2 n}\sum_{k=1}^{n}(p_k \times log_2 p_k)$, where n is the number of source code files in module M, and p_k is the proportion of the changes to M that occur in file k.

We also detect whether there are post-release defects in each module. To detect post-release defects in a module, we identify defect-fixing changes that occurred after a studied release date. By studying the release practice of the

[1] http://qt-project.org/
[2] http://www.openstack.org/

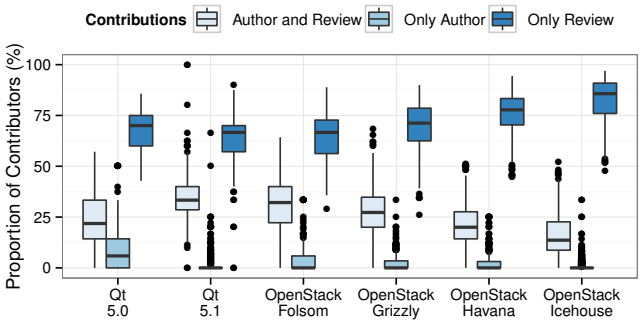


Figure 3: Number of developers who contribute to a module (i.e., authoring vs reviewing code changes).

studied systems, we found that the studied systems release sub-versions every two month after the main version is released. Hence, we use a two-month window to capture the defect-fixing changes. Similar to prior work [29], we search the VCS commit messages for co-occurrences of defect identifiers with keywords like "bug", "fix", or "defect". A similar approach is commonly used to determine defect-fixing and defect-inducing changes in prior work [22, 24, 30].

4. CASE STUDY RESULTS

In this section, we present the results of our case study with respect to our three research questions. For each research question, we present our empirical observations, followed by a general conclusion.

(RQ1) How do code authoring and reviewing contributions differ?

Approach. To address our RQ1, we examine the contributions of developers in each release of the studied systems. We analyze descriptive statistics of the number of developers who contribute by authoring or reviewing code changes to modules. Since the number of involved developers can vary among modules, we analyze the proportion of developers in a module instead of the actual number of developers.

Results. Observation 1 — 67%-86% of developers who contribute to a module did not previously author any code changes, yet they had previously reviewed 21%-39% of the code changes in that module. We find that there are on average 6-8 (Qt) and 17-32 (OpenStack) developers who contribute to a module. Figure 3 shows that the developers who previously only reviewed code changes are the largest set of developers who contribute to a module. 67%-70% (Qt) and 67%-86% (OpenStack) of developers are review-only contributors. On average, these review-only contributors have reviewed 29%-39% (Qt) and 21%-31% (OpenStack) of the code changes made to a module. This suggests that many developers contribute by only reviewing code changes to a module, yet their expertise is not captured by traditional code ownership heuristics.

Observation 2 — 18%-50% of the developers who only review code changes made to a module are documented core developers. 44%-51% (Qt) and 18%-21% (OpenStack) of the review-only contributors to modules are documented core developers.[3] On the other hand, 18%-20% (Qt) and 12%-26% (OpenStack) of developers who have authored a code change to a module are documented as core developers. Moreover, we observe that core developers tend to often contribute to a module as reviewers rather than authors. Indeed, 51%-54% (Qt) and 42%-60% (OpenStack) of modules do not have any code changes authored by core developers. On the other hand, 0%-8% (Qt) and 23%-36% (OpenStack) of modules that do not have any code changes reviewed by core developers.

The developers who only contribute to a module by reviewing code changes account for the largest set of contributors to that module. Moreover, 18%-50% of these review-only developers are documented core developers of the studied systems, suggesting that code ownership heuristics that only consider authorship activity are missing the activity of these major contributors.

(RQ2) Should code review activity be used to refine traditional code ownership heuristics?

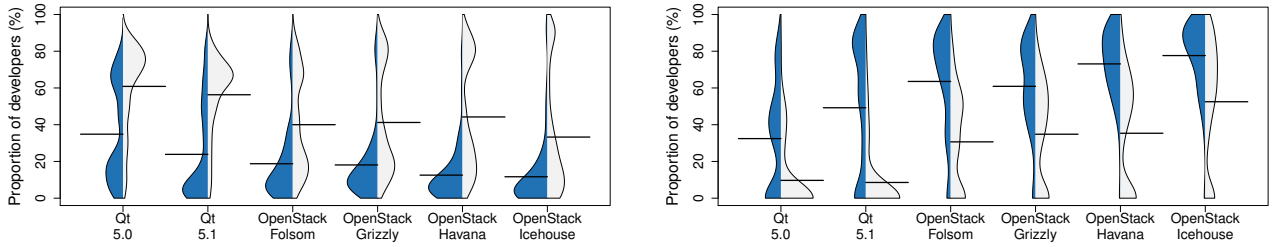
Approach. To address RQ2, we first examine the TCO values of each developer in each module against their RSO values. Next, we analyze the relationship between ownership and defect-proneness. To do so, for each module, we compute the proportion of developers in each expertise category of the review-aware ownership heuristic (see Table 1), i.e., (1) minor author & minor reviewer, (2) minor author & major reviewer, (3) major author & minor reviewer, and (4) major author & major reviewer. Then, we compare the proportion of developers in each of the expertise categories of defective and clean modules using beanplots [23]. Beanplots are boxplots in which the vertical curves summarize the distributions of different datasets. Defective modules are those that have at least one post-release defect, while clean modules are those that are free from post-release defects.

We use one-tailed Mann-Whitney U tests ($\alpha = 0.05$) to detect whether there is a statistically significant difference among the defective and clean modules in terms of the proportion of developers in the different expertise categories. We use Mann-Whitney U tests instead of T-tests because we observe that the distributions of the proportions of developers do not follow a normal distribution (Shapiro-Wilk test p-values are less than 2.2×10^{16} for all distributions). We also measure the effect size, i.e., the magnitude of the difference using Cliff's δ [27]. Cliff's δ is considered as trivial for $\delta < 0.147$, small for $0.147 \leq \delta < 0.33$, medium for $0.33 \leq \delta < 0.474$ and large for $\delta \geq 0.474$ [43].

Results. Observation 3 — 13%-58% of minor authors are major reviewers. When we assume that reviewers contribute equally (i.e., using RSO_{Even}), 41%-58% (Qt) and 13%-22% (OpenStack) of minor authors fall in the minor author & major reviewer category. Similarly, when we assume that reviewers who post more comments make more of a contribution (i.e., using $\text{RSO}_{\text{Proportional}}$), 32%-48% (Qt) and 11%-18% (OpenStack) of minor authors fall in the minor author & major reviewer category. This indicates that many minor authors make large review contributions.

Observation 4 — Clean modules tend to have more developers in the minor author & major reviewer category than defective modules do. Figure 4(a) shows that when we use RSO_{Even}, the proportion of developers in the minor author & major reviewer category of clean modules is larger than that of defective modules. Table 3 shows

[3]Based on the list in https://wiki.openstack.org/wiki/Project_Resources and http://wiki.qt.io/Maintainers



(a) Minor author & major reviewer category

(b) Minor author & minor reviewer category

Figure 4: The distribution of developers of defective (blue) and clean (gray) modules when using RSO_{Even}. The horizontal lines indicates the median of distributions.

Table 3: Results of one-tailed Mann-Whitney U tests for the developers in defective and clean models.

Release	RSO_{Even}				$RSO_{Proportional}$			
	Minor author & minor reviewer	Minor author & major reviewer	Major author & minor reviewer	Major author & major reviewer	Minor author & minor reviewer	Minor author & major reviewer	Major author & minor reviewer	Major author & major reviewer
5.0	D>C*** (M)	D<C*** (L)	D>C* (S)	∘	D>C*** (M)	D<C*** (L)	D>C* (S)	∘
5.1	D>C*** (L)	D<C*** (L)	D>C*** (M)	D<C*** (M)	D>C*** (L)	D<C*** (L)	D>C*** (S)	D<C*** (M)
Folsom	D>C*** (L)	D<C*** (L)	D<C* (S)	D<C*** (M)	D>C*** (L)	D<C*** (L)	∘	D<C*** (L)
Grizzly	D>C*** (L)	D<C*** (L)	∘	D<C*** (M)	D>C*** (L)	D<C*** (L)	D>C* (N)	D<C*** (M)
Havana	D>C*** (L)	D<C*** (L)	∘	D<C*** (M)	D>C*** (L)	D<C*** (L)	D>C** (N)	D<C*** (L)
Icehouse	D>C*** (L)	D<C*** (M)	∘	D<C*** (M)	D>C*** (L)	D<C*** (L)	D>C*** (S)	D<C*** (L)

Statistical significance: $\circ p \geq 0.05$, $^{*}p < 0.05$, $^{**}p < 0.01$, $^{***}p < 0.001$; **Effect size**: (L) Cliff's $\delta \geq 0.474$, (M) $0.33 \leq \delta < 0.474$, (S) $0.147 \leq \delta < 0.33$, (N) $\delta < 0.147$

that when using either RSO_{Even} or $RSO_{Proportional}$, the differences are statistically significant, with medium or large effect sizes. Our results indicate that post-release defects occur less frequently in the modules with a large proportion of developers in the minor author & major reviewer category than those with a smaller proportion of developers in minor author & major reviewer category.

Observation 5 — Conversely, defective modules tend to have more developers in the minor author & minor reviewer category than clean modules do. Figure 4(b) shows that when we use RSO_{Even}, the proportion of developers in the minor author & minor reviewer category of defective modules is larger than that of clean modules. Table 3 shows that the differences are statistically significant, with medium or large effect sizes when using either RSO_{Even} or $RSO_{Proportional}$. Our results indicate that post-release defects occur more frequently in the modules with a large proportion of developers in the minor author & minor reviewer category than those with a smaller proportion of developers in minor author & minor reviewer category.

We observe similar differences among the defective and clean modules for the major author & minor reviewer and the major author & major reviewer categories. When we use RSO_{Even}, Table 3 shows that the proportion of developers in the major author & minor reviewer category of defective modules is significantly larger than that of clean modules, with small or medium effect sizes for the Qt datasets. On the other hand, the proportion of developers in the major author & major reviewer category of defective modules is significantly smaller than that of clean modules, with medium effect sizes for the Qt 5.1 and all OpenStack datasets. We obtain similar results when using $RSO_{Proportional}$.

Many minor authors are major reviewers who actually make large contributions to the evolution of modules by reviewing code changes. Code review activity can be used to refine traditional code ownership heuristics to more accurately identify the defect-prone modules.

(RQ3) Is there a relationship between review-specific and review-aware code ownership heuristics and defect-proneness?

Although in RQ2 we find that there is a relationship between the expertise categories of review-aware heuristics and the tendency of having post-release defects in modules of the studied datasets, their effects could be correlated with other metrics that are known to share a relationship with defect proneness (e.g., module size) [55]. Hence, we examine the impact that review-specific and review-aware code ownership heuristics can have on defect-proneness using defect models that control for several confounding factors.

Approach. Similar to Bird *et al.* and other work [7, 10, 30, 55], our main goal of building defect models is not to predict defect-prone modules, but to understand the relationship between the explanatory variables and defect-proneness. To build defect models, we use logistic regression models to fit our studied datasets. Similar to our prior work [30], we adopt a nonlinear regression modeling approach, which enhances the fit of the data to be more accurate and robust, while carefully considering the potential for overfitting [18].

Our models operate at the module-level, where the response variable is assigned a value of TRUE if a module has at least one post-release defect, and FALSE otherwise. The explanatory variables are outlined in Table 4. Similar to prior work [7], we estimate the traditional and review-specific code ownership for a module by using the largest TCO and RSO values of the developers who contributed to that module. We also estimate the review-aware ownership for a module by computing the proportion of developers in each expertise category (*cf.* RQ2). Furthermore, in addition to the product and process metrics (i.e., size, churn, and entropy), we control for the number of contributors, authors, and reviewers in our models, since these metrics may have an impact on defect-proneness. Figure 5 provides an overview of the model construction and analysis approaches, which we describe below.

Table 4: A taxonomy of the considered control (top) and code ownership metrics (bottom).

Metrics	Description
Control Metrics	
Size	Number of lines of code.
Churn	Sum of added and removed lines of code.
Entropy	Distribution of changes among files.
#Contributor	The number of developers who contribute by authoring or reviewing code changes to the module.
#Author	The number of developers who have authored code changes to the module.
#Reviewer	The number of developers who have reviewed code changes to the module.
Code Ownership Metrics	
Top TCO	The traditional code ownership value of the developer who authored the most code changes to the module.
Top RSO	The review-specific ownership value of the developer who reviewed the most code changes to the module.
Review-Aware Ownership Metrics	
Proportion of minor author & major reviewer	A proportion of developers in the minor author & major reviewer category.
Proportion of major author & major reviewer	A proportion of developers in the major author & major reviewer category.
Proportion of minor author & minor reviewer	A proportion of developers in the minor author & minor reviewer category.
Proportion of major author & minor reviewer	A proportion of developers in the major author & minor reviewer category.

Minimize collinearity: We remove highly correlated explanatory variables before constructing our models to reduce the risk that those correlated variables interfere with our interpretation of the models. We measure the correlation between explanatory variables using Spearman rank correlation tests (ρ), which are resilient to data that is not normally distributed. We then use a variable clustering analysis approach [44] to construct a hierarchical overview of the inter-variable correlation and select one explanatory variable from each cluster of highly-correlated variables, i.e., $|\rho| > 0.7$ [26].

We also check for redundant variables (i.e., variables that do not offer a unique signal with respect to the other variables). We use the `redun` function in the `rms` R package [19] to detect redundant variables. However, we find that none of the explanatory variables that survive our correlation analysis are redundant.

Fit logistic regression models: In fitting our models, we carefully relax the linearity of the modeled relationship between explanatory and response variables, while being mindful of the risk of overfitting. To do so, we only allocate additional degrees of freedom (i.e., the number of regression parameters) to the explanatory variables that have more potential for sharing a nonlinear relationship with the response variable. We measure the potential for nonlinearity in the relationship between explanatory and response variables using a calculation of the Spearman multiple ρ^2. We then allocate five, three, and one degree of freedom to the explanatory variables that have strong ($\rho^2 > 0.3$), moderate ($0.15 < \rho^2 \leq 0.3$), and weak ($\rho^2 \leq 0.15$) potential for nonlinearity relationship, respectively.

Assess models and reliability: After fitting our defect models, we measure how well a model can discriminate between the potential response using the Area Under the receiver operating characteristic Curve (AUC) [17]. Furthermore, we evaluate the reliability of our models, since AUC can be too optimistic if the model is overfit to the dataset. Similar to our prior work [30], we estimate the optimism of AUC using a bootstrap-derived approach [11]. Small optimism values indicate that the model does not suffer from overfitting.

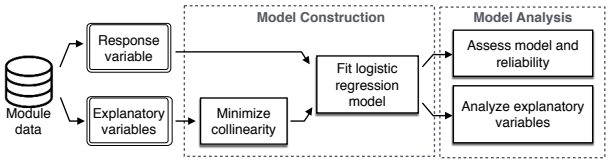


Figure 5: An overview of defect model construction and analysis approaches.

Analyze explanatory variables: We first measure the power of the explanatory variables that contribute to the fit of our models using Wald statistics. We use the `anova` function in the `rms` R package [19] to estimate the explanatory power (Wald χ^2) and the statistical significance (p-value) of each explanatory variable in our models. The larger the Wald χ^2 value, the larger the explanatory power of that variable.

We then examine the explanatory variables in relation to the predicted likelihood of module defect-proneness. We use the `Predict` function in the `rms` R package [19] to plot changes in the estimated likelihood of defect-proneness while varying one explanatory variable under test and holding the other explanatory variables at their median values.

Results. During correlation analysis, we find that the top TCO, the number of contributors, authors, and reviewers are often highly correlated. We select the number of contributors as the representative for these variables, since the number of contributors is simpler to calculate and can capture the number of both authors and reviewers. After removing the highly correlated variables, we repeat the variables clustering analysis and find that the number of contributors, the proportions of minor author & major reviewer and minor author & minor reviewer are highly correlated. We opt to remove the number of contributors, since the proportion of minor author & major reviewer and minor author & minor reviewer metrics are already controlled by the number of contributors. We also remove the proportion of minor author & major reviewer, since we want to revisit the relationship between the proportion of minor contributors (i.e., minor author & minor reviewer) and defect-proneness. For the sake of completeness, we analyze models that use the proportion of minor author & major reviewer instead of the proportion of minor author & minor reviewer, and found that the proportion of minor author & major reviewer had no discernible impact on model performance.

Table 5 shows that when using RSO_{Even} to estimate review-specific and review-aware ownership, our defect models achieve an AUC of between 0.81 (Qt 5.0 and OpenStack Icehouse) and 0.89 (OpenStack Havana). The AUC optimism is also relatively small ranging from 0.01 (OpenStack Havana and Icehouse) to 0.03 (OpenStack Folsom). We obtain similar model statistics when using $\text{RSO}_{\text{Proportional}}$. The AUC values of the $\text{RSO}_{\text{Proportional}}$ models range from 0.81 (Qt 5.0) to 0.87 (Qt 5.1 and OpenStack Havana), with an AUC optimism of 0.01-0.03. Since RSO_{Even} models use a simpler approximation of review-specific and review-aware ownership, we report our observations of the explanatory variables in the RSO_{Even} models below.

Observation 6 — The proportion of developers in the minor author & minor reviewer category shares a strong relationship with post-release defect prone-

Table 5: Statistics of defect models where review-specific and review-aware ownership are estimated using $\mathbf{RSO_{Even}}$. The explanatory power (χ^2) of each variable is shown in a proportion to Wald χ^2 of the model.

		Qt 5.0		Qt 5.1		OpenStack Folsom		OpenStack Grizzly		OpenStack Havana		OpenStack Icehouse	
AUC		0.81		0.86		0.89		0.83		0.88		0.81	
AUC optimism		0.02		0.02		0.03		0.02		0.01		0.01	
Wald χ^2		48^{***}		81^{***}		57^{***}		68^{***}		114^{***}		93^{***}	
		Overall	Nonlinear	Overall	Nonlinear	Overall	Nonlinear	Overall	Nonlinear	Overall	Nonlinear	Overall	Nonlinear
Size	D.F.	2	1	2	1	2	1	2	1	2	1	2	1
	χ^2	13%*	0%°	4%°	1%°	24%***	10%*	13%*	0%°	4%°	0%°	27%***	1%°
Churn	D.F.	1	–	1	–	1	–	1	–	1	–	1	–
	χ^2	4%°		0%°		0%°		1%°		15%***		12%**	
Entropy	D.F.	1	–	1	–	1	–	1	–	1	–	1	–
	χ^2	1%°		2%°		6%°		2%°		0%°		2%°	
Top TCO	D.F.	†		†		1		4	3	2	1	†	
	χ^2					5%°		13%°	2%°	1%°	0%°		
Top RSO_{Even}	D.F.	1	–	1	–	1	–	1	–	1	–	1	–
	χ^2	0%°		5%*		2%°		6%*		0%°		4%°	
Major author & major reviewer	D.F.	1	–	2	1	2	1	2	1	1	–	1	–
	χ^2	11%*		8%*	1%°	14%*	6%°	2%°	1%°	3%*		2%°	
Minor author & minor reviewer	D.F.	2	1	4	3	4	3	2	1	4	3	2	1
	χ^2	46%***	4%°	42%***	8%°	10%°	9%°	11%*	6%*	32%***	6%°	13%**	1%°
Major author & minor reviewer	D.F.	1	–	2	1	1	–	1	–	1	–	1	–
	χ^2	6%°		1%°	0%°	5%°		3%°		1%°		3%°	

†: Discarded during variable clustering analysis ($|\rho| \geq 0.7$)
The number of contributors, authors, reviewers, and the proportion of minor author & major reviewer are also discard during variable clustering analysis.
–: Nonlinear degrees of freedom not allocated.
Statistical significance of explanatory power according to Wald χ^2 likelihood ratio test: $\circ p \geq 0.05$; * $p < 0.05$; ** $p < 0.01$; *** $p < 0.001$

ness. Table 5 shows that the proportion of minor author & minor reviewer contributes a significant amount of explanatory power to the fit of our models (shown in the overall Wald χ^2 column). The OpenStack Folsom model is the only one where the proportion of minor author & minor reviewer did not contribute a significant amount of explanatory power. Furthermore, we observe that the proportion of minor author & minor reviewer accounts for most of the explanatory power in the Qt 5.0, Qt 5.1, and OpenStack Havana models. This result indicates that the rate of contributors who lack both authorship and reviewing expertise in a module shares a strong relationship between with the post-release quality.

Table 5 also shows that the additional degrees of freedom that we allocate to the proportion of minor author & minor reviewer did not contribute a significant amount of explanatory power to the fit of our models (shown in the nonlinear Wald χ^2 column). This result indicates that there was no significant benefit in spending additional degrees of freedom on this metric—the relationship is primarily log-linear.

Observation 7 — Modules with a higher rate of developers in the minor author & minor reviewer category are more likely to be defect-prone. Figure 6 shows that there is an increasing trend in the probability that a typical module will have post-release defects as the proportion of minor author & minor reviewer increases. The narrow breadth of the confidence interval (gray area) indicates that there is sufficient data to support the curve. Moreover, Figure 6 shows that the probability of having post-release defects rapidly increases when the proportion of minor author & minor reviewer increases beyond 0.5 in the Qt 5.0, Qt 5.1, OpenStack Grizzly and Havana models.

We also analyze the defect models that use the proportion of minor author & major reviewer instead of the proportion of minor author & minor reviewer. We find that the proportion of minor author & major reviewer contributes a significant amount of explanatory power to the Qt 5.0, Qt 5.1, OpenStack Havana and Icehouse models. Furthermore, we observe that there is an inverse relationship between the proportion of minor author & major reviewer and the probability that a typical module will have post-release defects in the Qt 5.0, Qt 5.1, OpenStack Havana and Icehouse models. This result indicates that the larger the proportion of developers in the minor author & major reviewer category, the lower the likelihood of a module having post-release defects.

Even when we control for several confounding factors, the proportion of developers in the minor author & minor reviewer category shares a strong relationship with defect-proneness. Indeed, modules with a larger proportion of developers without authorship or reviewing expertise are more likely to be defect-prone.

5. PRACTICAL SUGGESTIONS

In this section, we discuss the broader implications of our observations by offering the following suggestions:

(1) Code review activity should be included in future approximations of code ownership.
Observations 1 and 2 show that apart from the developers who author code changes to a module, there are major contributors who only contribute to that module by reviewing code changes. Furthermore, observation 3 shows that many developers who were identified as minor contributors by traditional code ownership heuristics are actually major contributors when their code review contributions are taken into consideration.

(2) Teams should apply additional scrutiny to module contributions from developers who have neither authored nor reviewed many code changes to that module in the past.
Observation 5 shows that modules with post-release defects tend to have a larger proportion of minor authors who are also minor reviewers than modules without post-release defects do. Furthermore, observations 6 and 7 show that the proportion of developers in the minor author & minor reviewer category shares a strong increasing relationship with the likelihood of having post-release defects in a module, even when we control for several confounding factors.

Figure 6: The estimated probability in a typical module for the proportion of developers in the minor author & minor reviewer category ranging. The gray area shows the 95% confidence interval.

(3) A module with many developers who have not authored many code changes should not be considered risky if those developers have reviewed many of the code changes to that module.
While observations 6 and 7 confirm the findings of prior work [7, 38], i.e., the number of minor contributors shares a relationship with defect-proneness, we find that the proportion of developers in another category of minor authors (i.e., minor author & major reviewer) shares an inverse relationship with the defect-proneness (see RQ3). Indeed, observation 4 shows that modules without post-release defects tend to have a larger proportion of developers in minor author & major reviewer than modules with post-release defects do.

6. THREATS TO VALIDITY

We now discuss the threats to the validity of our study.

External validity. We focus our study on two open source systems, due to the low number of systems that satisfied our eligibility criteria for analysis (see Section 3.1). Thus, our results may not generalize to all software systems. However, the goal of this paper is not to build a theory that applies to all systems, but rather to show that code review activity can have an impact on code ownership approximations. Our results suggest that code review activity should be considered in future studies of code ownership. Nonetheless, additional replication studies are needed to generalize our results.

Construct validity. Our analysis is focused on the code review activity that is recorded in the code review tools of the studied systems, i.e., Gerrit. However, there are likely cases where developers perform code reviews through other communication media, such as through in-person discussions [4], a group IRC [46], or a mailing list [15, 41]. Unfortunately, there are no explicit links of code changes to those communication threads, and recovering these links is a non-trivial research problem [2, 6]. Nevertheless, we perform our study on modules where every code change could be linked to the reviews in Gerrit, which should capture the majority of the review discussion in the studied systems.

Since we identify defect-fixing changes using keyword-based approach, there are likely cases that our defect data is inaccurate [5]. To evaluate this, we measure the accuracy of our approach by manually examining samples of defect-fixing changes. For each studied system, we randomly select 50 code changes that are flagged as bug-fixing changes by our keyword-based approach. We find that 88%-92% of the sampled code changes are correctly identified.

Internal validity. We identify developers who post at least one review comment as reviewers of a module, although some of the identified reviewers may only leave superficial or unrelated review comments [8, 36]. We attempt to mitigate this risk with our $\text{RSO}_{\text{Proportional}}$ heuristic, which allocates less ownership value for reviewers who provide less feedback. We find that the results of our study using $\text{RSO}_{\text{Proportional}}$ heuristic are similar to the results using the simpler, RSO_{Even} heuristic, which allocates an even share of the ownership value to every reviewer of a code change. This suggests that the noise of reviewer contributions is not heavily biasing our results. Nonetheless, we plan to explore more advanced review contribution heuristics in future work.

In this paper, we opt to measure RSO and TCO values separately. Even when they are combined in our the review-aware ownership heuristics, the values are plotted on orthogonal axes rather than summed. Summation of RSO and TCO values may have a different association with software quality. However, since reviewing and authoring are different activities, a naive summation may not be desired. Additional work is needed to investigate appropriate means of computing a generic expertise metric.

There may still be cases where developers who make many contributions by either authoring or reviewing code changes are not the actual owner of the modules. Unfortunately, ground truth data is not available for us to validate against. Nevertheless, we use a list of core developers that is available in the documentation of the studied systems to validate our heuristics. Our results also show that many of these core developers can only be module owners if their code review contributions are considered in code ownership heuristics.

7. RELATED WORK

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

Code review. Defect prevention is not the sole concern of modern code reviews [41, 49]. Beller *et al.* find that review comments are often focused on improving maintainability rather than fixing defects [4]. Tsay *et al.* report that reviewers are concerned with the appropriateness of a code solution and often provide alternative solutions during code reviews [52]. Bacchelli and Bird find that code reviews at Microsoft provide additional benefits to development teams, such as knowledge transfer among team members [1]. Our study shows that the review contributions of developers can be used to estimate their module-specific expertise.

Other recent work has analyzed the usefulness of code reviews in the MCR process. Rigby *et al.* investigate several factors that have an impact on the effectiveness of code

reviews [40]. Bosu *et al.* uncover characteristics of useful review comments and investigate the factors that influence the comment usefulness density in Microsoft projects [8]. Baysal *et al.* also study the influence of many technical and non-technical factors on the timeliness of code reviews in the WebKit and Google Blink open source projects [3]. Inspired by these studies, we propose $RSO_{Proportional}$ in order to take the number of review comments into an account when weighting review contributions.

Several studies propose an approach to optimize the code review processes. Shull *et al.* report that perspective-based reviews catch 35% more defects than non-directed reviews [48]. Rigby and Bird find that two reviewers find an optimal number of defects [39]. Recent work also proposes approaches to find reviewers who should be included in a review [51, 53, 54]. In this paper, our results suggest that reducing the proportion of developers who neither author nor review many code changes in a module tends to decrease the likelihood of having defects in the future.

Moreover, many recent studies have investigated the effect that MCR has on software quality. Meneely *et al.* analyzed the association between review activities and the risk of having security-related problems [31]. Our prior work investigates the relationship between developer involvement in the code review process and the incidence of post-release defects in several open source projects [29, 30]. Morales *et al.* also examine the impact that review investment has on software design quality [34]. Moreover, our recent work investigates code review activity in the files that have been defective in the past, and files that will be defective in the future [50]. In this paper, our results show that the review-aware ownership heuristics shares a strong relationship with the likelihood of having post-release defects in a module.

This related work strengthens our assertion that reviewers make a non-trivial contribution to the evolution of a software system. In this paper, we argue that this non-trivial contribution of reviewers should be taken into consideration when approximating code ownership. Our results indicate that reviewing activity can indeed be used to refine existing code ownership heuristics, narrowing their focus to the developers who truly lack module-specific expertise.

Code ownership. Prior work has shown that the code authorship can be used to estimate developer expertise. Mockus and Herbsleb use Experience Atoms (EA), i.e., the development and maintenance tasks that a developer has completed, as a measure of developer expertise [33]. Schuler and Zimmermann create an expertise profile from a set of methods that developers have implemented or referenced [45]. We find that code review activities also should be considered when estimating developer expertise.

Several studies also use code authorship to identify the defect-prone modules. Pinzger *et al.* show that contribution networks that are built from author contributions can accurately identify defect-prone modules in the Microsoft Vista system [37]. Meneely *et al.* examine the association between the number of commits that developers have made to a module and the incidence of security-related problems in the Red Hat Enterprise Linux 4 kernel [32]. Matsumoto *et al.* also find that developer metrics, e.g., the code churn or the number of commits that are generated by each developer are good indicators of defect-prone source code files in the Eclipse Platform project [28]. Furthermore, recent work studies the relationship between software quality and code ownership approximations that are derived from code authorship data. Nagappan *et al.* study the influence that organizational structure has on software quality [35]. Bird *et al.* find that code ownership of developers shares a strong relationship with defect-proneness [7]. Rahman and Devanbu also report that an area of source code that has been associated with defects in the past tends to be written by developers with low code ownership [38]. We find that refining code authorship by using code review activities provides a more comprehensive picture of the association between developer contributions and software quality.

On the other hand, recent work by Fritz *et al.* shows that authorship data was not strongly associated with knowledge about a module [12]. Indeed, several studies complement authorship data with the data recorded during developer IDE activities to compute developer expertise [13, 21, 42]. Similar to these studies, our study aims to incorporate another overlooked source of expertise data—we adapt traditional code ownership heuristics to include code review activity.

8. CONCLUSIONS

Code ownership heuristics have been used in many studies for identifying the developers who are responsible for maintaining modules. These heuristics are traditionally computed using code authorship contributions. However, developers can also make important contributions to modules by critiquing code changes during the code review process.

In this paper, we extend the traditional code ownership heuristics to be: (1) review-specific, i.e., a code ownership approximation that is derived solely using review contributions and (2) review-aware, i.e., a code ownership approximation that is derived using both authorship and review contributions. Through a case study of six releases of the large Qt and OpenStack open source systems, we make the following observations:

- 67%-86% of developers only contribute to a module by reviewing code changes. 18%-50% of these review-only contributors are documented core developers of the studied systems (Observations 1 and 2).
- 13%-58% of developers who are flagged as minor contributors by traditional code ownership heuristics are actually major contributors when their code review activity is considered (Observation 3).
- When traditional code ownership heuristics are refined by code review activity, we find that modules without post-release defects tend to have a higher rate of developers in the minor author & major reviewer category, but a lower rate of developers in the minor author & minor reviewer category than modules with post-release defects do (Observations 4 and 5).
- Even when we control for several factors that are known to have an impact on software quality, the proportion of developers in the minor author & minor reviewer category shares a strong, increasing relationship with the likelihood of having post-release defects in a module (Observations 6 and 7).

Our results suggest that future approximations of code ownership should take code review activity into consideration in order to more accurately model the contributions that developers have made to evolve software systems and to identify defect-prone modules.

9. REFERENCES

[1] A. Bacchelli and C. Bird. Expectations, Outcomes, and Challenges Of Modern Code Review. In *Proceedings of the 35th International Conference on Software Engineering*, pages 712–721, 2013.

[2] A. Bacchelli, M. Lanza, and R. Robbes. Linking E-Mails and Source Code Artifacts. In *Proceedings of the 32nd International Conference on Software Engineering*, pages 375–384, 2010.

[3] O. Baysal, O. Kononenko, R. Holmes, and M. W. Godfrey. Investigating Technical and Non-Technical Factors Influencing Modern Code Review. *Empirical Software Engineering*, page to appear, 2015.

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

[5] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu. Fair and Balanced? Bias in Bug-Fix Datasets. In *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the International Symposium on the Foundations of Software Engineering*, pages 121–130, 2009.

[6] C. Bird, A. Gourley, and P. Devanbu. Detecting Patch Submission and Acceptance in OSS Projects. In *Proceedings of the 4th International Working Conference on Mining Software Repositories*, pages 26–29, 2007.

[7] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu. Don't Touch My Code! Examining the Effects of Ownership on Software Quality. In *Proceedings of the 8th joint meeting of the European Software Engineering Conference and the Internaltional Symposium on the Foundations of Software Engineering*, pages 4–14, 2011.

[8] A. Bosu, M. Greiler, and C. Bird. Characteristics of Useful Code Reviews: An Empirical Study at Microsoft. In *Proceedings of the 12th International Working Conference on Mining Software Repositories*, pages 146–156, 2015.

[9] B. Caglayan, B. Turhan, A. Bener, M. Habayeb, A. Miransky, and E. Cialini. Merits of Organizational Metrics in Defect Prediction: An Industrial Replication. In *Proceedings of the 37th International Conference on Software Engineering*, pages 89–98, 2015.

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

[11] B. Efron. How Biased is the Apparent Error Rate of a Prediction Rule? *Journal of the American Statistical Association*, 81(394):461–470, 1986.

[12] T. Fritz, G. C. Murphy, and E. Hill. Does a Programmer's Activity Indicate Knowledge of Code? In *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the International Symposium on the Foundations of Software Engineering*, pages 341–350, 2007.

[13] T. Fritz, J. Ou, G. C. Murphy, and E. Murphy-Hill. A Degree-of-Knowledge Model to Capture Source Code Familiarity. In *Proceedings of the 32nd International Conference on Software Engineering*, pages 385–394, 2010.

[14] M. Greiler, K. Herzig, and J. Czerwonka. Code Ownership and Software Quality: A Replication Study. In *Proceedings of the 12th International Working Conference on Mining Software Repositories*, pages 2–12, 2015.

[15] A. Guzzi, A. Bacchelli, M. Lanza, M. Pinzger, and A. Van Deursen. Communication in Open Source Software Development Mailing Lists. In *Proceedings of the 10th International Working Conference on Mining Software Repositories*, pages 277–286, 2013.

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

[17] J. A. Hanley and B. J. McNeil. The Meaning and Use of the Area under a Receiver Operating Characteristic Curve. *Radiological Society of North America*, 143(1):29–36, 1982.

[18] F. E. Harrell Jr. *Regression Modeling Strategies: With Application to Liniear Models, Logistic Regression, and Survival Analysis*. Springer, 1st edition, 2002.

[19] F. E. Harrell Jr. rms: Regression Modeling Strategies, 2015.

[20] A. E. Hassan. Predicting Faults Using the Complexity of Code Changes. In *Proceedings of the 31st International Conference on Software Engineering*, pages 78–88, 2009.

[21] L. P. Hattori, M. Lanza, and R. Robbes. Refining code ownership with synchronous changes. *Empirical Software Engineering*, 17(4-5):467–499, 2012.

[22] 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*, 39(6):757–773, 2013.

[23] P. Kampstra. Beanplot: A Boxplot Alternative for Visual Comparison of Distributions. *Journal of Statistical Software*, 28(1):1–9, 2008.

[24] S. Kim, J. Whitehead Jr., and Y. Zhang. Classifying Software Changes: Clean or Buggy? *Transactions on Software Engineering*, 34(2):181–196, 2008.

[25] O. Kononenko, O. Baysal, L. Guerrouj, Y. Cao, and M. W. Godfrey. Investigating Code Review Quality: Do People and Participation Matter? In *Proceedings of the 31st International Conference on Software Maintenance and Evolution*, pages 111–120, 2015.

[26] H. C. Kraemer, G. A. Morgan, N. L. Leech, J. A. Gliner, J. J. Vaske, and R. J. Harmon. Measures of Clinical Significance. *Journal of the American Academy of Child & Adolescent Psychiatry*, 42(12):1534–1529, 2003.

[27] G. Macbeth, E. Razumiejczyk, and R. D. Ledesma. Cliff's Delta Calculator: A Non-parametric Effect Size Program for Two Groups of Observations. *Universitas Psychologica*, 10:545–555, 2011.

[28] S. Matsumoto, Y. Kamei, A. Monden, K.-i. Matsumoto, and M. Nakamura. An Analysis of Developer Metrics for Fault Prediction. In *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, pages 18:1–18:9, 2010.

[29] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan. The Impact of Code Review Coverage and Code Review Participation on Software Quality. In *Proceedings of the 11th International Working Conference on Mining Software Repositories*, pages 192–201, 2014.

[30] 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 Engineering*, page to appear, 2015.

[31] 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 *Proceedings of the 6th International Workshop on Social Software Engineering*, pages 37–44, 2014.

[32] A. Meneely and L. Williams. Secure Open Source Collaboration: An Empirical Study of Linus' Law. In *Proceedings of the 16th Conference on Computer and Communications Security*, pages 453–462, 2009.

[33] A. Mockus and J. D. Herbsleb. Expertise Browser: A Quantitative Approach to Identify Expertise. In *Proceedings of the 24th International Conference on Software Engineering*, pages 503–512, 2002.

[34] R. Morales, S. McIntosh, and F. Khomh. Do Code Review Practices Impact Design Quality? A Case Study of the Qt, VTK, and ITK Projects. In *Proceedings of the 22nd International Conference on Software Analysis, Evolution, and Reengineering*, pages 171–180, 2015.

[35] N. Nagappan, B. Murphy, and V. R. Basili. The Influence of Organizational Structure on Software Quality: An Empirical Case Study. In *Proceedings of th 30th International Conference on Software Engineering*, pages 521–530, 2008.

[36] T. Pangsakulyanont, P. Thongtanunam, D. Port, and H. Iida. Assessing MCR Discussion Usefulness using Semantic Similarity. In *Proceedings of the 6th International Workshop on Empirical Software Engineering in Practice*, pages 49–54, 2014.

[37] M. Pinzger, N. Nagappan, and B. Murphy. Can Developer-Module Networks Predict Failures? In *Proceedings of the 16th International Symposium on Foundations of Software Engineering*, pages 2–12, 2008.

[38] F. Rahman and P. Devanbu. Ownership, Experience and Defects: A Fine-Grained Study of Authorship. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 491–500, 2011.

[39] P. C. Rigby and C. Bird. Convergent Contemporary Software Peer Review Practices. In *Proceedings of the 9th joint meeting of the European Software Engineering Conference and the International Symposium on the Foundations of Software Engineering*, pages 202–212, 2013.

[40] P. C. Rigby, D. M. German, L. Cowen, and M.-a. Storey. Peer Review on Open-Source Software Projects: Parameters, Statistical Models, and Theory. *Transactions on Software Engineering and Methodology*, 23(4):35:1–35:33, 2014.

[41] P. C. Rigby and M.-A. Storey. Understanding Broadcast Based Peer Review on Open Source Software Projects. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 541–550, 2011.

[42] R. Robbes and D. Röthlisberger. Using developer interaction data to compare expertise metrics. In *Proceedings of the 10th International Working Conference on Mining Software Repositories*, pages 297–300, 2013.

[43] J. Romano, J. D. Kromrey, J. Coraggio, and J. Skowronek. Appropriate Statistics for Ordinal Level Data: Should We Really Be Using T-Test and Cohen's d for Evaluating Group Differences on the NSSE and Other Surveys? In *the annual meeting of the Florida Association of Institutional Research*, pages 1–33, 2006.

[44] W. S. Sarle. The VARCLUS Procedure. In *SAS/STAT User's Guide*. SAS Institute, Inc, 4th edition, 1990.

[45] D. Schuler and T. Zimmermann. Mining Usage Expertise from Version Archives. In *Proceedings of the 5th International Working Conference on Mining Software Repositories*, pages 121–124, 2008.

[46] E. Shihab, Z. M. Jiang, and A. E. Hassan. Studying the Use of Developer IRC Meetings in Open Source Projects. In *Proceedings of the 25th International Conference on Software Maintenance*, pages 147–156, 2009.

[47] E. Shihab, A. Mockus, Y. Kamei, B. Adams, and A. E. Hassan. High-Impact Defects: A Study of Breakage and Surprise Defects. In *Proceedings of the 8th joint meeting of the European Software Engineering Conference and the Internaltional Symposium on the Foundations of Software Engineering*, pages 300–310, 2011.

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

[49] Y. Tao, D. Han, and S. Kim. Writing Acceptable Patches: An Empirical Study of Open Source Project Patches. In *Proceedings of the 30th International Conference on Software Maintenance and Evolution*, pages 271–280, 2014.

[50] 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 *Proceedings of the 12th International Working Conference on Mining Software Repositories*, pages 168–179, 2015.

[51] P. Thongtanunam, C. Tantithamthavorn, R. G. Kula, N. Yoshida, H. Iida, and K.-i. Matsumoto. Who Should Review My Code? A File Location-Based Code-Reviewer Recommendation Approach for Modern Code Review. In *Proceedings of the the 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering*, pages 141–150, 2015.

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

[53] X. Xia, D. Lo, X. Wang, and X. Yang. Who Should Review This Change? Putting Text and File Location Analyses Together for More Accurate Recommendations. In *Proceedings of the 31st International Conference on Software Maintenance and Evolution*, pages 261–270, 2015.

[54] M. Zanjani, H. Kagdi, and C. Bird. Automatically Recommending Peer Reviewers in Modern Code Review. *Transactions on Software Engineering*, page to appear, 2015.

[55] T. Zimmermann, N. Nagappan, P. J. Guo, and B. Murphy. Characterizing and Predicting Which Bugs Get Reopened. In *Proceedings of the 34th International Conference on Software Engineering*, pages 1074–1083, 2012.