A Mixed Methods Approach to Mining Code Review Data: Examples and a study of multi-commit reviews and pull requests

Peter C Rigby,[1] Alberto Bacchelli,[2] Georgios Gousios,[2] Murtuza Mukadam[1]
[1]Concordia University, Montreal, Canada
[2]Delft University of Technology, The Netherlands

Abstract

Software code review has been considered an important quality assurance mechanism for the last 35 years. The techniques for conducting modern code reviews have evolved along with the software industry and have become progressively incremental and lightweight. We have studied code review in number of contemporary settings, including Apache, Linux, KDE, Microsoft, Android, and GitHub. Code review is an inherently social activity, so we have used both quantitative and qualitative methods to understand the underlying parameters (or measures) of the process, as well as the rich interactions and motivations for doing code review. In this chapter, we describe how we have used a mixed methods approach to triangulate our findings on code review. We also describe how we use quantitative data to help us sample the most interesting cases from our data to be analyzed qualitatively. To illustrate code review research, we provide new results that contrast single and multi-commit reviews. We find that while multi-commit reviews take longer and have more lines churned than single commit reviews, the same number of people are involved in both types of review. To enrich and triangulate our findings, we qualitatively analyze the characteristics of multi-commit reviews and find that there are two types: reviews of branches and revisions to single commits. We also examine the reasons why commits on GitHub pull requests are rejected.

1 Introduction

Fagan's study of software inspections (*i.e.*, formal code review) in 1976 was one of the first attempts to provide an empirical basis for a software engineering process [13]. He showed that inspection, in which an independent evaluator examines software artifacts for problems, effectively found defects early in the development cycle and reduced costs. He concluded that the increased upfront investment in inspection led to fewer costly customer reported defects.

The techniques for conducting reviews have evolved along with the software industry and have progressively moved from Fagan's rigid and formal inspection process to an incremental and more lightweight modern code review process. We have studied modern code review in a number of settings: Apache, Linux, KDE [42, 43, 39, 41], Microsoft [4, 40], Android [29], and GitHub [19]. Since code review is an inherently complex social activity, we have used both quantitative and qualitative methods to understand the underlying parameters (or measures) of the process (*e.g.*, [40]) as well as the rich interactions and motivations for doing code review (*e.g.*, [4]). The goal of this chapter is to introduce the reader to code review data and to demonstrate how a mixed quantitative and qualitative approach can be used to triangulate empirical software engineering findings.

This chapter is structured as follows. In Section 2, we compare qualitative and quantitative methods and describe how and when they can be combined. In Section 3, we describe the available code review data sources and provide a metamodel of the fields one can extract. In Section 4, we conduct an illustrative quantitative investigation to study how multiple related commits (*e.g.*, commits on a branch) are reviewed. This study replicates many of the measures that have been used in the past, such as number of reviewers and time to perform a review. In Section 5, we describe how to collect and sample non-numerical data, *e.g.*, with interviews and from review emails, and extract themes from the data. Data is analyzed using a research method based on *grounded theory* [18]. In Section 6, we triangulate our findings on multi-commit reviews by quantitatively examining review discussions on multiple commits. We also suggest how one might continue this study by using card sorting or interviews. Section 7 concludes by summarizing our findings and the techniques we use.

2 Motivation for a mixed methods approach

While it is useful to develop new processes, practices, and tools, researchers suggest that these should not be "invented" by a single theorizing individual, but should be derived from empirical findings that can lead to empirically supported theories and testable hypotheses. Grounded, empirical findings are necessary to advance software development as an engineering discipline. With empirical work, one tries not to tell developers how they should be working, but instead tries to understand, for example, how the most effective developers work describing the essential attributes of their work practices. This knowledge can inform practitioners and researchers and influence the design of tools.

There are two complementary methods for conducting empirical work [11]: quantitative and qualitative. *Quantitative* analysis involves measuring the case. For example, we can measure how many people make a comment during a review meeting. Since there is little or no interpretation involved in extracting these measurements, quantitative findings are objective. One of the common risks when extracting measures is construct validity. Do the measures assess a real phenomenon or is there a systematic bias that reduces the meaningfulness of the measurements?

In contrast, *qualitative* findings allow the researcher to extract complex rich patterns of interactions. For example, knowing the number of people at a review meeting does not give information about *how* they interacted. Using a qualitative approach, one must code data to find the qualitative themes, such as how reviewers ask questions. While qualitative approaches like grounded theory ensure that each theme is tied back to a particular piece of data, potential for researcher bias is present.

Triangulation involves combining one or more research methods and data sources. The goal is to limit the weaknesses and biases present in each research method and dataset by using complementary methods and datasets. For example, one can measure attributes of archival review discussion and then interview developers involved in the reviews to check the validity of these measurements.

Replication involves conducting the same study using the same methodology on new cases. Yin [56] identifies two types of case study replications: literal and contrasting. The purpose of a literal replication is to ensure that similar projects produce similar results. For example, do two similar projects, such as Apache and Subversion, yield similar findings? Contrasting replications should produce contrasting results, but for reasons *predicted* by one's understanding of the differences between projects. For example, one might compare how reviewers select contributions for review on the Linux project with Microsoft Office. We would expect

to see differences between these two projects based on their drastically different organizational structures.

In subsequent sections, we will use quantitative methods to measure aspects of six case study replications, we triangulate our findings by using qualitative coding of archival review discussion to understand how group commits for review.

3 Review Process and Data

To give the reader a sense of the different types of code review, we summarize how review is done traditionally, on Open Source Software (OSS) projects, at Microsoft, on Google-lead OSS projects, and on GitHub. Subsequently, we provide a table of the different attributes of code review that we can measure in each environment.

3.1 Software Inspection

Software inspections are the most formal type of review. They are conducted after a software artifact meets predefined exit criteria (*e.g.*, a particular requirement is implemented). The process, originally defined by Fagan [13], involves some variations of the following steps: planning, overview, preparation, inspection, reworking, and follow-up. In the first three steps, the author creates an inspection package (*i.e.*, determines what is to be inspected), roles are assigned (*e.g.*, moderator), meetings are scheduled, and the inspectors examine the inspection package. The inspection is conducted, and defects are recorded but not fixed. In the final steps, the author fixes the defects and the mediator ensures that the fixes are appropriate. Although there are many variations on formal inspections, "their similarities outweigh their differences" [55].

3.2 Open Source Software Code Review

Asynchronous, electronic code review is a natural way for OSS developers, who meet in person only occasionally and to discuss higher level concerns [20], to ensure that the community agrees on what constitutes a good code contribution. Most large, successful OSS projects see code review as one of their most important quality assurance practices [41, 31, 2]. In OSS projects, a review begins with a developer creating a patch. A patch is a development artifact, usually code, that the developer feels will add value to the project. Although the level of formality

of the review processes varies among OSS projects, the general steps are consistent across most projects: (1) The author submits a contribution by emailing it to the developer mailing list or posting to the bug/review tracking system, (2) one or more people review the contribution, (3) the contribution is modified until it reaches the standards of the community, and (4) the revised contribution is committed to the code base. Many contributions are ignored or rejected and never make it into the code base [7].

3.3 Code Review at Microsoft

Microsoft developed an internal tool, CodeFlow, to aid in the review process. In CodeFlow a review occurs when a developer has completed a change, but prior to checking it into the version control system. A developer will create a review by indicating which changed files should be included, providing a description of the change (similar to a commit message), and specifying who should be included in the review. Those included receive email notifications and then open the review tool that displays the changes to the files and allows the reviewers to annotate the changes with their own comments and questions. The author can respond to the comments within the review and can also submit a new set of changes that addresses issues that the reviewers have brought up. Once a reviewer is satisfied with the changes, he can 'sign off' on the review in CodeFlow. For more details on the code review process at Microsoft, we refer the reader to an earlier empirical study [4] in which we investigated the purposes for code review (*e.g.*, finding defects, sharing knowledge) along with the actual outcomes (*e.g.*, creating awareness and gaining code understanding) at Microsoft.

3.4 Google-based Gerrit Code Review

When the Android project was released as OSS, the Google engineers working on Android wanted to continue using the internal Mondrian code review tool and process used at Google [45]. Gerrit is an OSS, Git specific implementation of the code review tool used internally at Google, created by Google Engineers [14]. Gerrit centralizes Git acting as a barrier between a developer's private repository and the shared centralized repository. Developers make local changes in their private Git repositories and then submit these changes for review. Reviewers make comments via the Gerrit web interface. For a change to be merged into the centralized source tree, it must be approved and verified by another developer. The review process has the following stages:

1. *Verified* - Before a review begins, someone must verify that the change merges with the current master branch and does not break the build. In many cases, this step is done automatically.

2. *Approved* - While anyone can comment on the change, someone with appropriate privileges and expertise must approve the change.

3. *Submitted/Merged* - Once the change has been approved it is merged into Google's master branch so that other developers can get the latest version of the system.

3.5 GitHub pull requests

Pull requests is the mechanism that GitHub offers for doing code reviews on incoming source code changes.[1] A GitHub pull request contains a branch (local or in another repository) from which a core team member should pull commits. GitHub automatically discovers the commits to be merged and presents them in the pull request. By default, pull requests are submitted to the base ('upstream' in Git parlance) repository for review. There are two types of review comments:

1. *Discussion* - Comments on the overall contents of the pull request. Interested parties engage in technical discussion regarding the suitability of the pull request as a whole.

2. *Code Review* - Comments on specific sections of the code. The reviewer makes notes on the commit diff, usually of technical nature to pinpoint potential improvements.

Any GitHub user can participate in both types of review. As a result of the review, pull requests can be updated with new commits or the pull request can be rejected — as redundant, uninteresting, or duplicate. The exact reason a pull request is rejected is not recorded, but can be inferred from the pull request discussion. In case an update is required as a result of a code review, the contributor creates new commits in the forked repository and, after the changes are pushed to the branch to be merged, GitHub will automatically update the commit list in the pull request. The code review can then be repeated on the refreshed commits.

[1] GitHub pull requests `https://help.github.com/articles/using-pull-requests` Accessed July 2014

When the inspection process ends and the pull request is deemed satisfactory, it can be merged by a core team member. The versatility of Git enables pull requests to be merged in various ways, with varying levels of preservation of the original source code metadata (*e.g.*, authorship information, commit dates, *etc.*).

Code reviews in pull requests are in many cases implicit and therefore not observable. For example, many pull requests receive no code comments and no discussion, while they are still merged. Unless it is project policy to accept any pull request without reviewing, it is usually safe to assume that a developer reviewed the pull request before merging.

3.6 Data Measures and Attributes

A code review is effective if the proposed changes are eventually accepted or bugs are prevented, and it is efficient if the time this takes is as short as possible. To study patch acceptance and rejection and the speed of the code review process, a framework for extracting meta-information about code reviews is required. Code reviewing processes are common in both open source [41, 19] and commercial software development environments [4, 40]. Researchers have identified and studied code review in contexts such as patch submission and acceptance [30, 7, 54, 5] and bug triaging [1, 15]. Our meta-model of code review features is presented in Table 1. To develop this meta-model we have included the features used in existing work on code review. This meta-model is not exhaustive, but it forms a basis that other researchers can extend. Our meta-model features can be split in three broad categories:

1. *Proposed change features* - These characteristics attempt to quantify the impact of the proposed change on the affected code base. When examining external code contributions, the size of the patch affects both acceptance and acceptance time [54]. There are various metrics to determine the size of a patch that have been used by researchers: code churn [30, 38], changed files [30] and number of commits. In the particular case of GitHub pull requests, developers reported that the presence of tests in a pull request increases their confidence to merge it [34]. The number of participants has been shown to influence the amount of time take to conduct a code review [40].

2. *Project features* - These features quantify the receptiveness of a project to an incoming code change. If the project's process is open to external contributions, then we expect to see an increased ratio of external contributors

Feature	Description
Code Review Features	
num_commits	Number of commits in the proposed change
src_churn	Number of lines changed (added + deleted) by the proposed change.
test_churn	Number of test lines changed in the proposed change.
files_changed	Number of files touched by the proposed change.
num_comments	Discussion and code review comments.
num_participants	Number of participants in the code review discussion
Project Features	
sloc	Executable lines of code when the proposed change was created.
team_size	Number of active core team members during the last 3 months prior to the proposed change creation.
perc_ext_contribs	The ratio of commits from external members over core team members in the last *n* months.
commits_files_touched	Number of total commits on files touched by the proposed change *n* months before the proposed change creation time.
test_lines_per_kloc	A proxy for the project's test coverage.
Developer	
prev_changes	Number of changes submitted by a specific developer, prior to the examined proposed change.
requester_succ_rate	% of the developer's changes that have been integrated up to the creation of the examined proposed change.
reputation	Quantification of the developer's reputation in the project's community, *e.g.*, followers on GitHub

Table 1: Meta-model for code review analysis.

over team members. The project's size may be a detrimental factor to the speed of processing a proposed change, as its impact may be more difficult to assess. Also, incoming changes tend to cluster over time (the "yesterday's weather" change pattern [16]), so it is natural to assume that proposed changes affecting a part of the system that is under active development will be more likely to merge. Testing plays a role in speed of processing; according to [34], projects struggling with a constant flux of contributors use testing, manual or preferably automated, as a safety net to handle contributions from unknown developers.

3. *Developer* - Developer-based features attempt to quantify the influence that the person who created the proposed change has on the decision to merge it and the time to process it. In particular, the developer who created the patch has been shown to influence the patch acceptance decision [21] (recent work on different systems interestingly reported opposite results [6]). To abstract the results across projects with different developers, researchers devised features that quantify the developer's track record [12], namely the number of previous proposed changes and their acceptance rate; the former has been identified as a strong indicator of proposed change quality [34]. Finally, Bird *et al.* [8], presented evidence that social reputation has an impact on whether a patch will be merged; consequently, features that quantify the developer's social reputation (*e.g.*, follower's in GitHub's case) can be used to track this.

4 Quantitative Replication Study: Code review on branches

Many studies have quantified the attributes of code review that we discussed in Table 1 (*e.g.*, [42, 47, 5]). All review studies to date have ignored the number of commits (*i.e.*, changes or patches) that are under discussion during review. Multi-commit reviews usually involve a feature that is broken into multiple changes (*i.e.*, a branch) or a review that requires additional corrective changes submitted to the original patch (*i.e.*, revisions). In this section, we perform a replication study using some of the attributes described in the previous section to understand how multiple related commits affect the code review process. In Section 6, we triangulate our results by qualitatively examining how and why multiple commits are reviewed. We answer the following research questions:

Projects	Period	Years
Linux Kernel	2005–2008	3.5
Android	2008–2013	4.0
Chrome	2011–2013	2.1
Rails	2008–2011	3.2
Katello	2008–2011	3.2
Wildfly	2008–2011	3.2

Table 2: The time period we examined in years and the number of reviews

RQ1 How many commits are part of each review?

RQ2 How many files and lines of code are changed per review?

RQ3 How long does it take to perform a review?

RQ4 How many reviewers make comments during a review?

We answer these questions in the context of Android and Chromium OS, which use Gerrit for review; the Linux Kernel, which performs email-based review; and Rails, Katello, and Wildfly projects, which use GitHub pull requests for review. These projects were selected because they are all successful, medium to large in size, and represent different software domains.

Table 2 shows the dataset and the time period we examined in years. We use the data extracted from these projects and perform comparisons that help us draw conclusions.

More details on the review process used by each project are provided in the discussion in Section 3. We present our results as boxplots, which show the distribution quartiles, or, when the range is large enough, a distribution density plot [22]. In both plots, the median is represented by a bold line.

4.1 RQ 1 — Commits per review

How many commits are part of each review?

Table 3 shows the size of the data set and proportion of reviews that involve more than one commit. Chrome has the largest percentage of multi-commit reviews at 63%, and Rails has the smallest at 29%. Single commit reviews dominate on most projects. In Figure 1 we only consider multi-commit reviews and find that

Projects	All	ingle Commit	Multi-commit
Linux Kernel	20.2K	70%	30%
Android	16.4K	66%	34%
Chrome	38.7K	37%	63%
Rails	7.3K	71%	29%
Katello	2.6K	62%	38%
Wildfly	5K	60%	40%

Table 3: Number of reviews in our dataset and number of commits per review

Linux, Wildfly, Katello have a median of three commits. Android, Chrome, Rails have a median of two commits.

4.2 RQ2 — Size of commits

How many files and lines of code are changed per review?

Figure 2 shows that WildFly makes the larges commits with single commits having a median of 40 lines churned, while Katello makes the smallest changes (median 8 lines). WildFly also makes the largest multi-commit changes, with a median of 420 lines churned, and Rails makes the smallest with a median of 43.

In terms of lines churned, multi-commit reviews are 10, 5, 5, 5, 11, and 14 times larger than single commit reviews for Linux, Android, Chrome, Rails, Wildfly, and Katello, respectively. In comparison with Figure 1, we see that multi-commit reviews have one to two more commits than single commit reviews. Normalizing for the number of commits, we see that individual commits in multi-commit reviews contain more lines churned than single commits. We conjecture that multi-commit changes implement new features or involve complex changes.

4.3 RQ3 — Review Interval

How long does it take to perform a review?

The review interval is the calendar time since the commit was posted until the end of discussion of the change. Review interval is an important measure of review efficiency [25, 35]. The speed of feedback provided to the author depends on the length of the review interval. Researchers also found that the review interval is related to the overall timeliness of a project [52].

Current practice is to review small changes to the code before they are committed and to do this review quickly. Reviews on OSS systems, at Microsoft, and

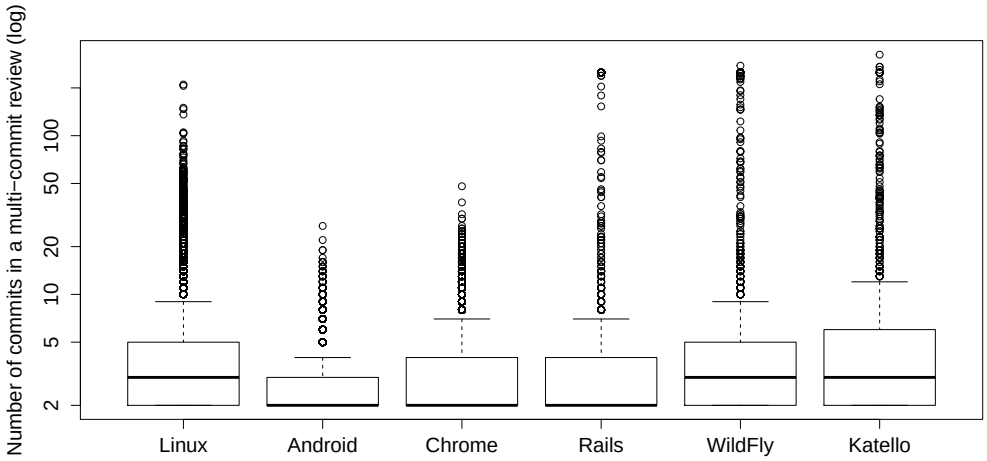

Figure 1: Number of Patches

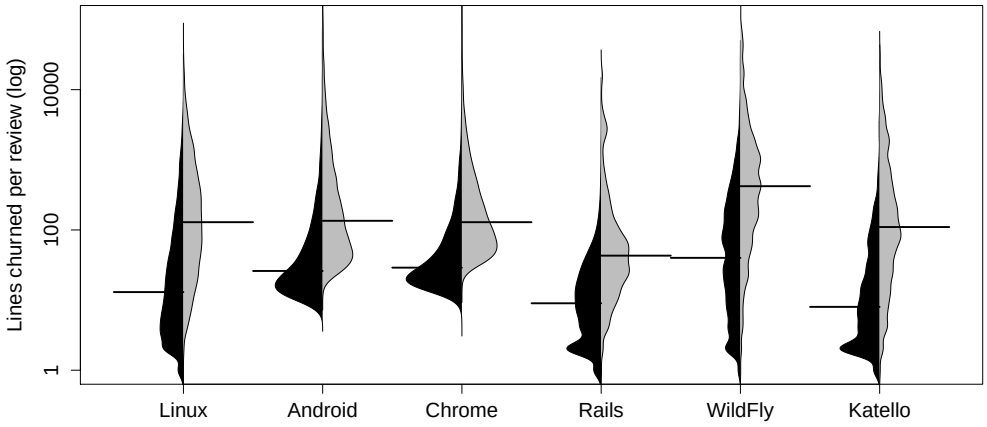

Figure 2: Number of lines churned – left: single commits, right: multi-commit reviews

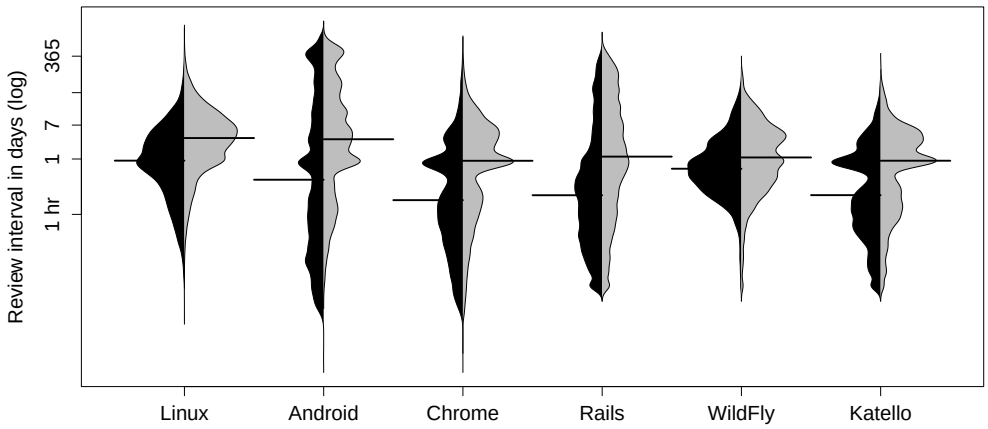

Figure 3: Review Interval – left: single commits, right: multi-commit reviews

on Google-lead projects last for around 24 hours [40]. This interval is very short compared with the months or weeks required for formal inspections [13, 35].

Figure 3 shows that the GitHub projects perform reviews at least as quickly as Linux and Android. Single commit reviews happen in a median of 2.3 hours (Chrome) and 22 hours (Linux). Multi-commit reviews are finished in a median of 22 hours (Chrome) and 3.3 days (Linux).

Multi-commit reviews take 4, 10, 10, 9, 2, and 7 times longer than single commits for Linux, Android, Chrome, Rails, Wildfly, and Katello, respectively.

4.4 RQ4 - Reviewer Participation

How many reviewers make comments during a review?

According to Sauer *et al.*, two reviewers tend to find an optimal number of defects [44]. Despite different processes for reviewer selection (*e.g.*, self-selection vs. assignment to review), the number of reviewers per review is two in the median case across a large diverse set of projects [40]. With the exception of Rails that has a median of three reviewers on multi-commits, reviews are conducted by two reviewers regardless of the number of commits (see Figure 4).

The number of comments made during a review varies with the number of

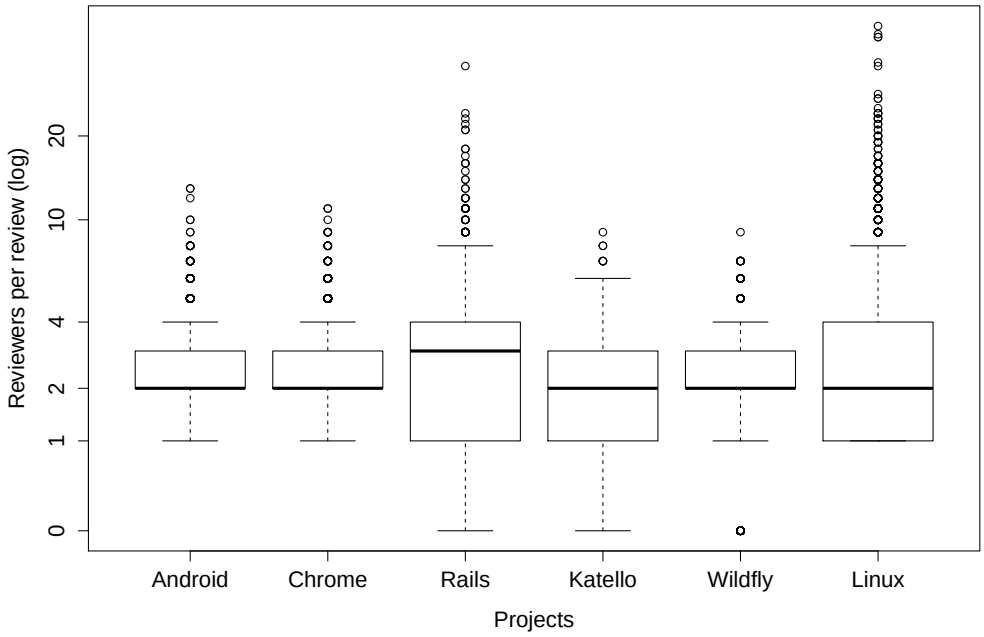

Figure 4: Number of reviewers in multi-commit reviews

commits. Android, Chrome, and Wildfly increase from 3 to 4 comments, while Rails and Katello go from 1 to 3 comments. Linux sees a much larger increase from 2 to 6 comments.

4.5 Conclusion

In this section, we contrasted multi-commit and single-commit reviews on the number of commits, churn, interval, and participation. We found that multi-commit reviews increase the number of commits by one to two in the median case. The churn increases more dramatically, between 5 and 14 times the number of changed lines in the median case. The amount of time to perform a review varies quite substantially from 2.3 hours to 3.3 days in the median case. Multi-commit reviews take 2 to 10 times longer than single commit reviews. The number of reviewers is largely unaffected by review, we see two reviewers per review, with the exception of Rails. The number of comments per review increases on multi-commit reviews. We purposefully did not perform statistical comparisons

among projects. Statistical comparisons are not useful because we have the entire population of reviews for each project and not a sample of reviews. Furthermore, given the large number of reviews we have for each project, even small differences that have no practical importance to software engineers would be highlighted by statistical tests.

No clear patterns emerged when comparing the multi-commit and single-commit review styles across projects and across review infrastructures (*e.g.*, Gerrit vs. GitHub). Our main concern is that multi-commit reviews will serve two quite different purposes. The first purpose would be to review branches that contain multiple related commits. The second purpose will be to review a single commit after it has been revised to incorporate reviewer feedback. In Section 5, we randomly sample multi-commit reviews and perform a qualitative analysis to determine the purpose of each review. These qualitative findings will enhance the measurements made in this section by uncovering the underlying practices used on each project.

5 Qualitative Approaches

Studies of formal software inspection [13, 26, 25] and code review [10, 42, 37, 31, 47, 40, 5] have largely been quantitative with the goal of showing that a new process or practice is effective, finds defects, and is efficient.

Unlike traditional inspection that has a prescriptive process, modern code review has gradually emerged from industrial and open-source settings. Since the process is less well defined, it is important to conduct exploratory analyses that acknowledge the context and interactions that occur during code review. These studies are required to gain an in-depth *understanding* of code review, which is a complex phenomenon that encompasses non-trivial social aspects. Such understanding can be gained only by answering many *how* and *why* questions, for which numbers and statistics would only give a partial picture. For this reason, the studies were conducted using *qualitative methods*.

Qualitative methods involve the "*systematic gathering and interpretation of nonnumerical data*" [24]. Software engineering researchers collect the non-numerical data by studying the people involved in a software project as they work, typically by conducting *field research* [9]. Field research consists of "a group of methods that can be used, individually or in combination, to understand different aspects of real world environments" [27]. Lethbridge *et al.* surveyed how field research had been performed in software engineering and accordingly proposed a taxonomy of

data collection techniques by grouping them in three main sets [27]: (1) direct, (2) indirect, and (3) independent. Direct data collection techniques (*e.g.*, focus groups, interviews, questionnaires, or think-aloud sessions) require researchers to have direct involvement with the participant population; indirect techniques (*e.g.*, instrumenting systems, or "fly on the wall") require the researcher to have only indirect access to the participants via direct access to their work environment; independent techniques (*e.g.*, analysis of tool use logs, documentation analysis) require researcher to access only work artifacts, such as issue reports or source code.

In this section, we analyze two exploitative studies on code reviews. We dive into the details of the qualitative research methods used in two of those studies, to understand them in more detail and to show how qualitative research could be employed to shine a light on aspects of modern code review practices.

The first study manually examines archives of review data to understand how OSS developers interact and manage to effectively conduct reviews on large mailing lists [43]. The second study investigates the motivations driving developers and managers to do and require code review among product teams at Microsoft [4]. The two considered studies use both direct and independent data collection techniques. The gathered data is then manually analyzed according to different qualitative methodologies.

5.1 Sampling approaches

Qualitative research requires such labor-intensive data analysis that not all the available data can be processed and it is necessary to extract samples. This regards not only indirect data (*e.g.*, documents), but also direct data and its collection (*e.g.*, people to interview), so that the data can be subsequently analyzed.

In quantitative research, where computing power can be put to good use, one of the most common approaches for sampling is random sampling: Picking a high number of cases, randomly, to perform analyses on. Such an approach requires a relatively high number of cases so that they will be *statistically* significant. For example, if we are interested in estimating a proportion (*e.g.*, number of developers who do code reviews) in a population with 1,000,000, we have to randomly sample 16,317 cases to achieve a confidence level of 99% with an error of 1% [50]. Unless very computationally expensive, such a number is simple to reach with quantitative analyses.

Given the difficulty of manually coding data points in qualitative research, random sampling can lead to non-optimal samples. In the following we explain

how sampling was conducted on the two considered qualitative studies, first in doing direct data collection, then in doing independent data collection.

5.1.1 Sampling direct data

As previously mentioned, one of the main goals of qualitative research is to see a phenomenon from the perspective of another person, normally involved with it. For this reason, one of the most common ways to collect qualitative data is to use direct data collection by means of observations and interviews. Through observations and interviews we can gather accurate and finely nuanced information, thus exposing participants' perspectives.

However, interviews favor depth over quantity. Interviewing and the consequent non-numerical data analysis are time-consuming tasks, and not all the candidates are willing to participate in investigations and answer interview questions. In the two considered qualitative studies, authors used different approaches to sample interview participants.

Rigby and Storey [43] selected interviewees among the developers of the Apache and Subversion projects. They ranked developers based on the number of email based reviews they had performed, and they sent an interview request to each of the *top five* reviewers on each project. The researchers' purpose was to interview the most prolific reviewers, in order to learn from their extensive experience and daily practice, thus understanding the way in which experts dealt with submitted patches on OSS systems. Overall, the respondents that they interviewed were nine core developers, either with committer rights or maintainers of a module.

Bacchelli and Bird [4] selected interviewees among different Microsoft product teams (*e.g.*, Excel and SQL Server). They sampled developers based on the number of reviews they had done since the introduction of CodeFlow (see Section 3.3): They contacted 100 randomly selected candidates who signed-off between 50 and 250 code reviews. In this case, they do not select the *top* reviewers, but a sample of those with an average mid-to-high activity. In fact, their purpose was to understand the motivation for developers to do code reviews, and by selecting only the most prolific reviewers they could obtain biased the results. The respondents that they interviewed comprised 17 people: five developers, four senior developers, six testers, one senior tester, and one software architect. Their time in the company ranged from 18 months to almost 10 years, with a median of five years.

5.1.2 Sampling indirect data

Although, as shown in the rest of this chapter, indirect code review data can be used entirely when employing quantitative methods, this is not the case for qualitative analyses. In fact, if willing to qualitatively analyze review comments recorded in emails or in archives generated by code review tools, researchers have to sample this data to make it manageable.

In the considered studies, Rigby and Storey [43] analyzed email reviews for six OSS software projects, and Bacchelli and Bird [4] analyzed code review comments recorded by CodeFlow during code reviews done in different Microsoft product groups. In both cases, since no human participants were involved, the researchers could analyze hundreds of documents.

5.1.3 Data saturation

Although a large part of the qualitative analysis is conducted after the data gathering, qualitative researchers also analyze their data throughout their data collection. For this reason, in qualitative research it is possible rely on *data saturation* to verify whether the size of a sample could be large enough for the chosen research purpose [18]. Data saturation happens both in direct data and indirect data collection and occurs when the researcher is no longer seeing, hearing, or reading new information from the samples.

In both studies the number of data points, *e.g.*, reviews and possible interviewees, was much larger than any group of researchers could reasonably analyze. During analysis, when no new patterns were emerging from the data, saturation had been reached and the researchers stopped introducing new data points and started the next stage of analysis.

5.2 Data collection

In the following, we describe how the researchers of the two studies collected data by interviewing and observing the study participants selected in the sampling phase. In both studies, the aim of this data collection phase was to gain an understanding of code reviewing practices by adopting the perspective of the study participants (this is often the main target of qualitative research [24]). Moreover, we also briefly explain how they collected indirect data about code reviews.

5.2.1 Observations and Interviews at Microsoft

In the study conducted at Microsoft, each meeting with participants comprised two parts: an observation, and a following *semi-structured* interview [49].

In the emails sent to candidate participants of the study, Bacchelli and Bird invited developers to notify them back when they received the next review task, so that the researchers could go to the participant's office (this happened within 30 minutes from the notification) to observe how developers conducted the review. To minimize invasiveness and the Hawthorne effect [33], only one researcher went to the meeting and observed the review. To encourage the participants to narrate their work (thus collecting more non-numerical data), the researcher asked the participants to consider him as a newcomer to the team. In this way, most developers thought aloud without need of prompting.

With consent, assuring the participants of anonymity, the audio of the meeting was recorded. Recording is a practice on which not all the qualitative researcher methodologists agree [18]. In this study, researchers preferred to have recorded audio for two reasons: (1) Having the data to analyze for the researcher that was not participating to the meetings; and (2) the researcher at the meeting could fully focus on the observation and interaction with participants during the interview. Since the researchers, as observers, have backgrounds in software development and practices at Microsoft, they could understand most of the work and where and how information was obtained without inquiry.

After the observations, the second part of the meeting took place, *i.e.*, the semi-structured interview. This form of interview makes use of an *interview guide* that contains general groupings of topics and questions rather than a predetermined exact set and order of questions. Semi-structured interviews are often used in an exploratory context to "*find out what is happening* [and] *to seek new insights*" [53].

Researchers devised the first version of the guideline by analyzing a previous internal Microsoft study on code review practices, and by referring to academic literature. Then, the guideline was iteratively refined after each interview, in particular when developers started providing answers very similar to the earlier ones, thus reaching saturation.

After the first 5-6 meetings, the observations reached the saturation point. For this reason, the researchers adjusted the meetings to have shorter observations, which they only used as a starting point for interacting with participants and as a hook to talk about topics in the interview guideline.

At the end of each interview, the audio was analyzed by the researchers, and

then transcribed and broken up into smaller coherent units for subsequent analysis.

5.2.2 Indirect data collection: Review comments and emails

As we have seen in previous sections, code review tools archive a lot of valuable information for data analysis; similarly, mailing lists archive discussions about patches and their acceptance. Although part of the data is numerical, a great deal of information is non-numerical data, for example, code review comments. This information is a good candidate for qualitative analysis, since it contains traces of opinions, interactions, and general behavior of developers involved in the code review process.

Bacchelli and Bird randomly selected 570 code review comments from CodeFlow data pertaining to more than 10 different product teams at Microsoft. They considered only comments within threads with at least two comments, so that they were sure there was interaction between developers. Considering that they were interested in measuring the types of comments, this amount, from a quantitative perspective would have a confidence level of 95% and an error of 8%.

Rigby and Storey randomly sampled 200 email reviews for Apache, 80 for Subversion, 70 for FreeBSD, 50 for the Linux kernel, and 40 email reviews and 20 Bugzilla reviews for KDE. On each project, the main themes from the previous project re-occurred indicating that that saturation had been reached and that it was unnecessary to code an equivalent number of reviews for each project.

5.3 Qualitative analysis of Microsoft data

To qualitatively analyze the data gathered from observations, interviews, and recorded code review comments, Bacchelli and Bird used two techniques: A *card sort* and an *affinity diagram*.

5.3.1 Card Sorting

To group codes that emerged from interviews and observations into categories, Bacchelli and Bird conducted a *card sort*. Card sorting is a sorting technique that is widely used in information architecture to create mental models and derive taxonomies from input data [48]. In their case it helped to organize the codes into hierarchies to deduce a higher level of abstraction and identify common themes. A card sort involves three phases: In the (1) *preparation phase*, participants of the card sort are selected and the cards are created; in the (2) *execution phase*, cards

are sorted into meaningful groups with a descriptive title; and in the (3) *analysis phase*, abstract hierarchies are formed to deduce general categories.

Bacchelli and Bird applied an *open card sort*: There were no predefined groups. Instead, the groups emerged and evolved during the sorting process. In contrast, a closed card sort has predefined groups and is typically applied when themes are known in advance, which was not the case for our study.

Bacchelli created all of the cards, from the 1,047 coherent units generated from the interview data. Throughout the further analysis other researchers (Bird and external people) were involved in developing categories and assigning cards to categories, so as to strengthen the validity of the result. Bacchelli played a special role of ensuring that the context of each question was appropriately considered in the categorization, and creating the initial categories. To ensure the integrity of the categories, Bacchelli sorted the cards several times. To reduce bias from Bacchelli sorting the cards to form initial themes, all researchers reviewed and agreed on the final set of categories.

The same method was applied to group code review comments into categories: Bacchelli and Bird printed one card for each comment (along with the entire discussion thread to give the context), and conducted a card sort, as performed for the interviews, to identify common themes.

5.3.2 Affinity diagramming

Bacchelli and Bird used an *affinity diagram* to organize the categories that emerged from the card sort. This technique allows large numbers of ideas to be sorted into groups for review and analysis [46]. It was used to generate an overview of the topics that emerged from the card sort, in order to connect the related concepts and derive the main themes. For generating the affinity diagram, Bacchelli and Bird followed the five canonical steps: they (1) recorded the categories on post-it-notes, (2) spread them onto a wall, (3) sorted the categories based on discussions, until all are sorted and all participants agreed, (4) named each group, and (5) captured and discussed the themes.

5.4 Applying Grounded Theory to Archival Data to Understand OSS Review

The preceding example demonstrated how to analyze data collected from interviews and observation using a card sort and an affinity diagram. Qualitative analysis can also be applied to the analysis of archival data, such as records of code

review (see Figure 5). To provide a second perspective on qualitative analysis, we describe the methodology used by Rigby and Storey to code review discussion on six OSS projects [43]. In the next section, we describe one of the themes that emerged from this analysis: patchsets. Patchsets are groups of related patches that implement a larger feature or fix and are reviewed together.

The analysis of the sample email reviews followed Glaser's [18] approach to grounded theory where manual analysis uncovers emergent abstract themes. These themes are developed from descriptive codes used by the researchers to note their observations. The general steps used in grounded theory are as follows: note-taking, coding, memoing, sorting, and writing.[2] Below we present each of these steps in the context of the study of Rigby and Storey:

1. *Note Taking* - Note taking involves creating summaries of the data without any interpretation of the events [18]. The comments in each review were analyzed chronologically. Since patches could often take up many screens with technical details, the researchers first summarized each review thread. The summary uncovered high-level occurrences, such as how reviewers interacted and responded.
2. *Coding* - Codes provide a way to group recurring events. The reviews were coded by printing and reading the summaries and writing the codes in the margin. The codes represented the techniques used to perform a review and the types and styles of interactions among stakeholders. The example shown in Figure 5 combines note taking and coding with emergent codes being underlined.
3. *Memos* - Memoing is a critical aspect of grounded theory and differentiates it from other qualitative approaches. The codes that were discovered on individual reviews were grouped together and abstracted into short memos that describe the emerging theme. Without this stage, researchers fail to abstract codes and present 'stories' instead of a high-level description of the important aspects of a phenomenon.
4. *Sorting* - Usually there are too many codes and memos to be reported in a single paper. The researchers must identify the core memos and sort and group them into a set of related themes. These core themes become the grounded 'theory.' A theme was the way reviewers asked authors question about their code.

[2] A simple practical explanation of grounded theory is `http://www.aral.com.au/resources/grounded.html` accessed March 2014

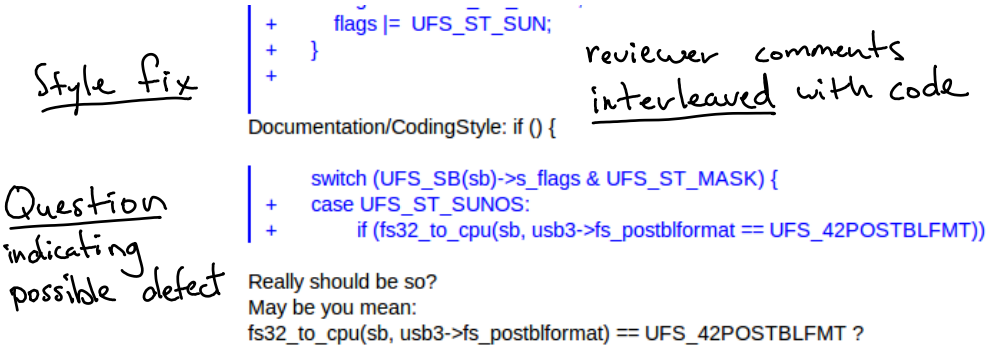

Figure 5: Example fragment of review with three codes written in the margins: a type of fix, a question that indicates a possible defect, and interleaved comments.

5. *Writing* - Writing the paper is simply a matter of describing the evidence collected for each theme. Each core theme is written up by tracing the theme back to the abstract memos, codes, and finally the data points that lead to the theme. One common mistake is to include too many low-level details and quotations [18]. In the work by Rigby and Storey the themes are represented throughout the paper as paragraph and section headings.

6 Triangulation

Triangulation "involves the use of multiple and different methods, investigators, sources, and theories to obtain corroborating evidence" [32]. Since each method and dataset has different strengths and weakness that offset each other when they are combined, triangulation reduces the overall bias in a study. For example, survey and interview data suffer from the biases that participants self-report on events that have happened in the past. In contrast, archival data is a record of real communication and so does not suffer from the self-reporting bias. However, since archival data was collected without a research agenda, it can often be missing information that a researcher needs to answer his or her questions. This missing information can be supplement with interview questions.

In this section, we first describe how Bacchelli and Bird [4] triangulate their findings using follow up surveys. We then triangulate our quantitative findings from Section 4 on multi-commit reviews by first describing a qualitative study of branch reviews on Linux. We then manual code multi-commit reviews as either

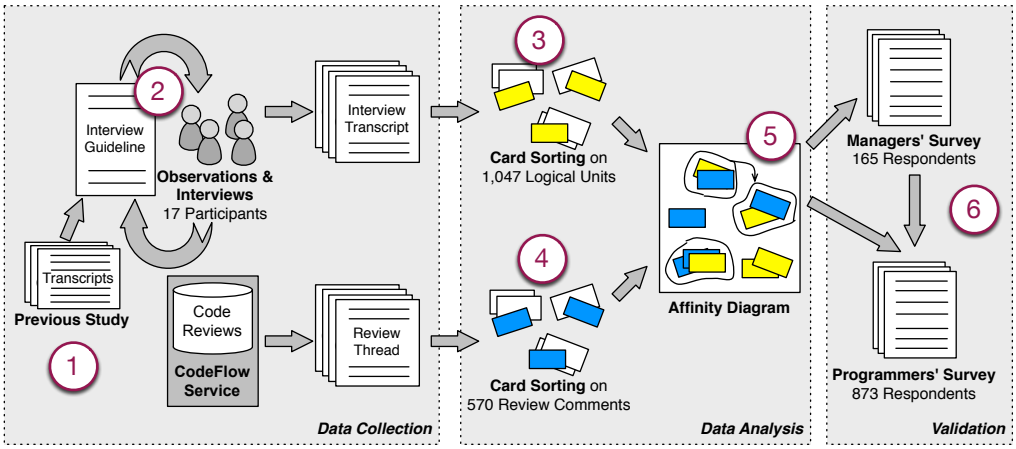

Figure 6: Triangulation-based methodology used by Bacchelli and Bird [4]

branches or revisions on the Gerrit and GitHub projects that we examined in Section 4. We conclude this section, with a qualitative and quantitative examination of why GitHub reviews are rejected.

6.1 Using surveys to triangulate qualitative findings

The investigation of Bacchelli and Bird about expectations, outcomes, and challenges of code review employed a mixed quantitative-qualitative approach, which collects data from different sources for triangulation. Figure 6 shows the overall research method employed, and how the different sources are used to draw conclusions and test theories: (1) analysis of the previous study, (2) meetings with developers (observations and interviews), (3) card sort on meeting data, (4) card sort on code review comments, (5) affinity diagramming, and, (6) survey to managers and programmers.

The path including points 1 to 5 is described through Section 5, because it involves collection and analysis of non-numerical data; here we focus on the usage of surveys for additional triangulation. In Figure 6 we can see that the methodology already includes two different sources of data: direct data collection based on observations and interviews, and indirect data collection based on the analysis of comments in code review archives. Although these two sources are complementary and can be used to learn distinct stories, to eventually uncover the truth behind a question, they both suffer from a limited number of data points. To

overcome this issue, Bacchelli and Bird used surveys to validate—with a larger, statistically significant sample—the concepts that emerged from the analysis of the data gathered from other sources.

In practice, they created two surveys and sent to a large number of participants and to triangulate the conclusions of their qualitative analysis. The full surveys are available as a technical report [3]. For the design of the surveys, Bacchelli and Bird followed the guidelines of Kitchenham and Pfleeger for personal opinion surveys [23]. Although they could have sent the survey in principle to all the employees at Microsoft, they selected samples that were statistically significant, but at the same time would not inconveniently hit an unnecessarily large number of people. Both surveys were anonymous to increase response rates [51].

They sent the first survey to a cross section of managers. They considered managers for whom at least half of their team performed code reviews regularly (on average, one per week or more) and sampled along two dimensions. The first dimension was whether or not the manager had participated in a code review himself since the beginning of the year and the second dimension was whether the manager managed a single team or multiple teams (a manager of managers). Thus, they had one sample of first level managers who participated in review, another sample of second level managers who participated in reviews, *etc.* The first survey was a short survey comprising 6 questions (all optional), which they sent to 600 managers that had at least 10 direct or indirect reporting developers who used CodeFlow. The central focus was the open question asking to enumerate the main motivations for doing code reviews in their team. They received 165 answers (28% response rate), analyzed before devising the second survey.

The second survey comprised 18 questions, mostly closed with multiple choice answers, and was sent to 2,000 randomly chosen developers who signed off on average at least one code review per week since the beginning of the year. They used the time frame of January to June of 2012 to minimize the amount of organizational churn during the time period and identify employees' activity in their current role and team. The survey received 873 answers (44% response rate). Both response rates were high, as other online surveys in software engineering have reported response rates ranging from 14% to 20% [36].

Although the surveys also included open questions, they were mostly based on closed ones, thus could be used as a basis for statistical analyses. Thanks to the high number of respondents, Bacchelli and Bird could triangulate their qualitative findings with a larger set of data, thus increasing the validity of their results.

6.2 How multi-commit branches are reviewed on Linux

In Section 4, we quantitatively compared single and multi-commit reviews. We found that there was no clear pattern of review based on the size of project and the type of review tool (*i.e.*, Gerrit, GitHub, or email-based review). To enhance our findings and to understand the practices that underlie them, we qualitatively examine how multiple commits are handled on Linux. We find that multi-commit reviews contain patches related to a single feature or fix. In the next subsection, we use closed coding to determine if the other projects group commits by features or if multi-commit reviews are indicative of revisions to the original commit.

Instead of conducting reviews in Gerrit or as GitHub pull requests, the Linux Kernel uses mailing lists. Each review is conducted as an email thread. The first message in the thread will contain the patch and subsequent responses will be reviews and discussions of review feedback (Rigby and Storey provide more details on this point [43]). According to code review polices of OSS projects, individual patches are required to be in their smallest, functionally independent, and complete form [42]. Interviews of OSS developers also indicated a preference for small patches, with some stating that they refuse to review large patches until they are split into there component parts. For Iwai, a core developer on the Linux project, "if it [a large patch] can't be split, then something is wrong [*e.g.*, there are structural issues with the change]." However, by forcing developers to produce small patches, larger contributions are broken up and reviewed in many different threads. This division separates and reduces communication among experts, making it difficult to examine large contributions as a single unit. Testers and reviewers must manually combine these threads together. Interviewees complained about how difficult it is to combine a set of related patches to test a new feature.

Linux developers use 'patchsets' that allow developers to group related changes together, while still keeping each patch separate. A patchset is a single email thread that contains multiple numbered and related contributions. The first email contains a high level description that ties the contributions together and explains their interrelationships. Each subsequent message contains the next patch that is necessary to complete the larger change. For example, message subjects in a patchset might look like this:[3]

- Patch 0/3: Fixing and combining foobar with bar [no code modified]

[3] An Linux patchset: `http://lkml.org/lkml/2008/5/27/278` Accessed in January 2014

- Patch 1/3: Fix of foobar
- Patch 2/3: Integrate existing bar with foobar
- Patch 3/3: Update documentation on bar

Patchsets are effectively a branch of small patch commits that implements a larger change. The version control system Git contains a feature to send a branch as a number patchset to a mailing list for code review [17].

Notice how each sub-contribution is small, independent, and complete. Also, the contributions are listed in the order they should be committed to the system (*e.g.*, the fix to foobar must be committed before combining it with bar). Reviewers can respond to the overall patch (i.e., $0/N$) or they can respond to any individual patch (i.e., $n/N, n > 0$). As reviewers respond, sub-threads tackle sub-problems. However, it remains simple for testers and less experienced reviewers to apply the patchset as a single unit for testing purposes. Patchsets represent a perfect example of creating a fine, but functional and efficient division between the whole and the parts of a larger problem.

6.3 Closed coding: branch or revision on GitHub and Gerrit

A multi-commit review may be a related set of commits (a branch or patchset) or a revision to a commit. We conduct a preliminary analysis of 15 randomly sampled multi-commit reviews from each project to understand what type of review is occurring. Of the 15 reviews coded for each of the GitHub based projects, 73%, 86% and 60% of them were branches for Rails, WildFly, and Katello, respectively. These projects conducted branch review manner similar to Linux, but without the formality of describing the changes at a high level. Each change had a one line commit description that clearly indicated its connection to the next commit in the branch. For example, the commits in the following pull request implement two small parts of the same change:[4]

- Commit 1: 'Modified CollectionAssociation to refer to the new class name.'
- Commit 2: 'Modified NamedScopeTest to use CollectionAssociation.'

[4]Example of a pull request `https://github.com/rails/rails/pull/513/commits` accessed in March 2014.

WildFly had the highest percentage of branch reviews, which may explain why it had the largest number of lines changed and the longest review interval of all the projects we examined (see Section 4).

On GitHub, we also noted that some of the multi-commit reviews were of massive merges instead of individual feature changes.[5] Future work that examines 'massive' merge reviews would be interesting.

For Android and Chrome, we were surprised to find that none of the randomly selected reviews were of branches. Each multi-commit review involved revisions to a single commit. While future work is necessary to determine whether this is a defacto practice or enforced by policy, there is a preference at Google to commit onto a single branch [28]. Furthermore, the notion of 'patchset' in the Gerrit review system usually applies to an updated version of a patch rather than a branch, as it does in Linux [14].

6.4 Understanding why pull requests are rejected

As a final example of mixed qualitative-quantitative research involving triangulation, we present new findings on why pull request reviews are rejected on GitHub projects. Previous work has found relatively low rates of patch acceptance on large successful OSS projects. Bird *et al.* [7] found that the acceptance rate in three OSS projects is between 25% and 50%. On the six projects examined by Asundi and Jayant [2] they found that 28% to 46% of non-core developers had their patches ignored. Estimates of Bugzilla patch rejection rates on Firefox and Mozilla range from 61% [21] to 76% [31]. In contrast, while most proposed changes on GitHub pull requests are accepted [19], it is interesting to explore why some are not. Even though textual analysis tools (*e.g.*, natural language processing and topic modelling) are evolving, it is still difficult for them to accurately capture and classify the rationale behind such complex actions as rejecting code under review. For this reason, a researcher needs to resort to qualitative methods.

In the context of GitHub code reviews, we manually coded 350 pull requests and classified the reasons for rejection. Three independent coders did the coding. Initially, 100 pull requests were used by the first coder to identify discrete reasons for closing pull requests (bootstrapping sample), while a different set of 100 pull requests were used by all three coders to validate the identified categories (cross-validation sample). After validation, the two datasets were merged and a further

[5]Example of massive pull request `https://github.com/Katello/katello/pull/1024`

Reason	Description	%
obsolete	The PR is no longer relevant, as the project has progressed.	4
conflict	There feature is currently being implemented by other PR or in another branch.	5
superseded	A new PR solves the problem better.	18
duplicate	The functionality had been in the project prior to the submission of the PR	2
superfluous	PR doesn't solve an existing problem or add a feature needed by the project.	6
deferred	Proposed change delayed for further investigation in the future.	8
process	The PR does not follow the correct project conventions for sending and handling pull requests.	9
tests	Tests failed to run.	1
incorrect implementation	The implementation of the feature is incorrect, missing or not following project standards.	13
merged	The PR was identified as merged by the human examiner	19
unknown	The PR could not be classified due to lacking information	15

Table 4: Reasons for rejecting code under review.

150 randomly selected pull requests were added to the bootstrapping sample to construct the finally analyzed dataset for a total of 350 pull requests. The cross-validation of the categories on a different set of pull requests revealed that the identified categories are enough to classify all reasons for closing a pull request. The results are presented in Table 4.

The results show that there is no clearly outstanding reason for rejecting code under review. However, if we group together close reasons that have a timing dimension (obsolete, conflict, superseded), we see that 27% of unmerged pull requests are closed due to concurrent modifications of the code in project branches. Another 16% (superfluous, duplicate, deferred) are closed as a result of the contributor not having identified the direction of the project correctly and therefore submitting uninteresting changes. 10% of the contributions are rejected with reasons that have to do with project process and quality requirements (process, tests); this may be an indicator of processes not being communicated well enough or a rigorous code reviewing process. Finally, another 13% of the contributions are rejected because the code review revealed an error in the implementation.

Only 13% of the contributions are rejected due to technical issues, which are the primary reason for code reviewing, while a total 53% are rejected for rea-

sons having to do with the distributed nature of modern code reviews or the way projects handle communication of project goals and practices. Moreover, for 15% of the pull requests, the human examiners could not identify the cause of not integrating them. The latter is indicative of the fact that even in-depth, manual analysis can yield less than optimal results.

7 Conclusion

We have used our previous works to illustrate how qualitative and quantitative methods can be combined to understand code review [43, 4, 19, 41]. We have summarized the types of code review and presented a meta-model of the different measures that can be extracted. We have illustrated qualitative methods in Section 5 by describing how Rigby and Storey used grounded theory to understand how OSS developers interact and manage to effectively conduct reviews on large mailing lists [43]. We then contrast this methodology with Bacchelli and Bird's [4] study that used card sorting and affinity diagramming to investigate the motivations and requirements of interviewed manager and developers on the code review tool and processes used at Microsoft [4].

To provide an illustration of a mixed methods study, we presented new findings that contrast multi-commit reviews with single commits reviews. In Section 4 we presented quantitative results. While no clear quantitative pattern emerges when we compare across projects or types of review (*i.e.*, Gerrit, GitHub, and email-based review), we find that even though multi-commits take longer and involve more code than single commit reviews, multi-commit reviews have the same number of reviewers per review. We triangulated our quantitative findings by manually examining how multi-commit reviews are conducted (see Section 6.2 and Section 6.3). For Linux, we found that multi-commit reviews involve 'patchsets', which are reviews of branches. For Android and Chrome, multi-commit reviews are reviews of revisions to single commits. For the GitHub projects, Rails, WildFly and Katello, there is a mix of branch reviews and revisions to commits during review. As a final contribution, we presented new qualitative and quantitative results on why reviews on GitHub pull requests are rejected (Section 6.4).

References

[1] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 361–370, New York, NY, USA, 2006. ACM.

[2] J. Asundi and R. Jayant. Patch review processes in open source software development communities: A comparative case study. In *HICSS: Proceedings of the 40th Annual Hawaii International Conference on System Sciences*, page 10, 2007.

[3] A. Bacchelli and C. Bird. Appendix to expectations, outcomes, and challenges of modern code review. http://research.microsoft.com/apps/pubs/?id=171426, Aug. 2012. Microsoft Research, Technical Report MSR-TR-2012-83 2012.

[4] A. Bacchelli and C. Bird. Expectations, outcomes, and challenges of modern code review. In *Proceedings of the International Conference on Software Engineering*. IEEE, 2013.

[5] O. Baysal, O. Kononenko, R. Holmes, and M. Godfrey. The influence of non-technical factors on code review. In *Reverse Engineering (WCRE), 2013 20th Working Conference on*, pages 122–131, Oct 2013.

[6] M. Beller, A. Bacchelli, A. Zaidman, and E. Juergens. Modern code reviews in open-source projects: Which problems do they fix? In *Proceedings of MSR 2014 (11th Working Conference on Mining Software Repositories)*, pages 202–211, 2014.

[7] C. Bird, A. Gourley, and P. Devanbu. Detecting patch submission and acceptance in oss projects. In *MSR: Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 4. IEEE Computer Society, 2007.

[8] C. Bird, A. Gourley, P. Devanbu, A. Swaminathan, and G. Hsu. Open borders? Immigration in open source projects. In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 6, Washington, DC, USA, 2007. IEEE Computer Society.

[9] R. G. Burgess. *In the Field: An Introduction to Field Research*. Unwin Hyman, 1st edition, 1984.

[10] J. Cohen. *Best Kept Secrets of Peer Code Review*. Smart Bear Inc., 2006.

[11] J. Creswell. *Research design: Qualitative, quantitative, and mixed methods approaches*. Sage Publications, Inc., 2009.

[12] L. Dabbish, C. Stuart, J. Tsay, and J. Herbsleb. Social coding in Github: transparency and collaboration in an open software repository. In *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work*, CSCW '12, pages 1277–1286, New York, NY, USA, 2012. ACM.

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

[14] Gerrit. Web based code review and project management for git based projects. `http://code.google.com/p/gerrit/`.

[15] E. Giger, M. Pinzger, and H. Gall. Predicting the fix time of bugs. In *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*, RSSE '10, pages 52–56, New York, NY, USA, 2010. ACM.

[16] T. Girba, S. Ducasse, and M. Lanza. Yesterday's weather: guiding early reverse engineering efforts by summarizing the evolution of changes. In *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, pages 40 – 49, sep 2004.

[17] git. git-format-patch(1) manual page. `https://www.kernel.org/pub/software/scm/git/docs/git-format-patch.html`.

[18] B. Glaser. *Doing grounded theory: Issues and discussions*. Sociology Press Mill Valley, CA, 1998.

[19] G. Gousios, M. Pinzger, and A. v. Deursen. An exploratory study of the pull-based software development model. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 345–355, New York, NY, USA, 2014. ACM.

[20] A. Guzzi, A. Bacchelli, M. Lanza, M. Pinzger, and A. van Deursen. Communication in open source software development mailing lists. In *Proceedings of MSR 2013 (10th IEEE Working Conference on Mining Software Repositories)*, pages 277–286, 2013.

[21] G. Jeong, S. Kim, T. Zimmermann, and K. Yi. Improving code review by predicting reviewers and acceptance of patches. Technical Memorandum ROSAEC-2009-006, Research On Software Analysis for Error-free Computing Center, Seoul National University, 2009.

[22] P. Kampstra. Beanplot: A boxplot alternative for visual comparison of distributions. *Journal of Statistical Software, Code Snippets 1*, 28:1–9, 2008.

[23] B. Kitchenham and S. Pfleeger. Personal opinion surveys. *Guide to Advanced Empirical Software Engineering*, pages 63–92, 2008.

[24] A. J. Ko. Understanding software engineering through qualitative methods. In A. Oram and G. Wilson, editors, *Making Software*, chapter 4, pages 55–63. O'Reilly, 2010.

[25] S. Kollanus and J. Koskinen. Survey of software inspection research. *Open Software Engineering Journal*, 3:15–34, 2009.

[26] O. Laitenberger and J. DeBaud. An encompassing life cycle centric survey of software inspection. *Journal of Systems and Software*, 50(1):5–31, 2000.

[27] T. C. Lethbridge, S. E. Sim, and J. Singer. Studying software engineers: Data collection techniques for software field studies. *Empirical Software Engineerng*, 10:311–341, 2005.

[28] J. Micco. *Tools for Continuous Integration at Google Scale*. Google Tech Talk, Google Inc., 2012.

[29] M. Mukadam, C. Bird, and P. C. Rigby. Gerrit software code review data from android. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 45–48, Piscataway, NJ, USA, 2013. IEEE Press.

[30] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *Proceedings of the 27th international conference on Software engineering*, ICSE '05, pages 284–292, New York, NY, USA, 2005. ACM.

[31] M. Nurolahzade, S. M. Nasehi, S. H. Khandkar, and S. Rawal. The role of patch review in software evolution: an analysis of the mozilla firefox. In *International Workshop on Principles of Software Evolution*, pages 9–18, 2009.

[32] A. Onwuegbuzie and N. Leech. Validity and qualitative research: An oxymoron? *Quality and quantity*, 41(2):233–249, 2007.

[33] H. M. Parsons. What happened at Hawthorne? new evidence suggests the Hawthorne effect resulted from operant reinforcement contingencies. *Science*, 183(4128):922–932, March 1974.

[34] R. Pham, L. Singer, O. Liskin, F. Figueira Filho, and K. Schneider. Creating a shared understanding of testing culture on a social coding site. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 112–121, Piscataway, NJ, USA, 2013. IEEE Press.

[35] A. Porter, H. Siy, A. Mockus, and L. Votta. Understanding the sources of variation in software inspections. *ACM Transactions Software Engineering Methodology*, 7(1):41–79, 1998.

[36] T. Punter, M. Ciolkowski, B. Freimut, and I. John. Conducting on-line surveys in software engineering. In *International Symposium on Empirical Software Engineering*. IEEE, 2003.

[37] J. Ratcliffe. Moving software quality upstream: The positive impact of lightweight peer code review. In *Pacific NW Software Quality Conference*, 2009.

[38] J. Ratzinger, M. Pinzger, and H. Gall. EQ-mine: predicting short-term defects for software evolution. In *Proceedings of the 10th international conference on Fundamental approaches to software engineering*, FASE'07, pages 12–26, Berlin, Heidelberg, 2007. Springer-Verlag.

[39] P. Rigby, B. Cleary, F. Painchaud, M.-A. Storey, and D. German. Contemporary peer review in action: Lessons from open source development. *IEEE Software*, 29(6):56–61, Nov. 2012.

[40] P. C. Rigby and C. Bird. Convergent contemporary software peer review practices. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 202–212, New York, NY, USA, 2013. ACM.

[41] P. C. Rigby, D. M. German, L. Cowen, and M.-A. Storey. Peer Review on Open Source Software Projects: Parameters, Statistical Models, and Theory.

To appear in the ACM Transactions on Software Engineering and Methodology, page 34, August 2014.

[42] P. C. Rigby, D. M. German, and M.-A. Storey. Open source software peer review practices: A case study of the apache server. In *ICSE '08: Proceedings of the 30th International Conference on Software engineering*, pages 541–550, New York, NY, USA, 2008. ACM.

[43] P. C. Rigby and M.-A. Storey. Understanding broadcast based peer review on open source software projects. In *Proceeding of the 33rd International Conference on Software Engineering*, ICSE '11, pages 541–550, New York, NY, USA, 2011. ACM.

[44] C. Sauer, D. R. Jeffery, L. Land, and P. Yetton. The Effectiveness of Software Development Technical Reviews: A Behaviorally Motivated Program of Research. *IEEE Transactions Software Engineering*, 26(1):1–14, 2000.

[45] R. Schwartz. Interview with Shawn Pearce, Google Engineer, on FLOSS Weekly. `http://www.youtube.com/watch?v=C3MvAQMhC_M`.

[46] J. E. Shade and S. J. Janis. *Improving Performance Through Statistical Thinking*. Mcgraw-Hill, 2000.

[47] E. Shihab, Z. Jiang, and A. Hassan. On the use of internet relay chat (irc) meetings by developers of the gnome gtk+ project. In *MSR: In the Proceedings of the 6th IEEE International Working Conference on Mining Software Repositories*, pages 107–110. IEEE Computer Society, 2009.

[48] D. Spencer. Card sorting: a definitive guide. http://boxesandarrows.com/card-sorting-a-definitive-guide/, April 2004.

[49] B. Taylor and T. Lindlof. *Qualitative communication research methods*. Sage Publications, Incorporated, 2010.

[50] M. Triola. *Elementary Statistics*. Addison-Wesley, 10th edition, 2006.

[51] P. Tyagi. The effects of appeals, anonymity, and feedback on mail survey response patterns from salespeople. *Journal of the Academy of Marketing Science*, 17(3):235–241, 1989.

[52] L. G. Votta. Does every inspection need a meeting? *SIGSOFT Softw. Eng. Notes*, 18(5):107–114, 1993.

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

[54] P. Weißgerber, D. Neu, and S. Diehl. Small patches get in! In *MSR '08: Proceedings of the 2008 international working conference on Mining software repositories*, pages 67–76, New York, NY, USA, 2008. ACM.

[55] K. E. Wiegers. *Peer Reviews in Software: A Practical Guide*. Addison-Wesley Information Technology Series. Addison-Wesley, 2001.

[56] R. K. Yin. *Case Study Research: Design and Methods*, volume 5 of *Applied Social Research Methods Series*. Sage Publications Inc., 3 edition, 2003.