Towards Understanding and Improving Code Review Quality

by

Oleksii Kononenko

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2017

© Oleksii Kononenko 2017

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner	Dr. Thomas Zimmermann Senior Researcher Microsoft Research
Supervisors	Dr. Michael Godfrey Associate Professor David R. Cheriton School of Computer Science University of Waterloo
	Dr. Olga Baysal Assistant Professor School of Computer Science Carleton University
Internal Members	Dr. Meiyappan Nagappan Assistant Professor David R. Cheriton School of Computer Science University of Waterloo
	Dr. Derek Rayside Assistant Professor Electrical and Computer Engineering University of Waterloo
Internal-External Member	Dr. Krzysztof Czarnecki Professor Electrical and Computer Engineering University of Waterloo

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

I would like to acknowledge the names of my co-authors who contributed to the research described in this dissertation. These include:

- Yaxin Cao,
- Tresa Rose,
- Dr. Olga Baysal,
- Dr. Michael W. Godfrey,
- Dr. Latifa Guerrouj, and
- Dr. Reid Holmes.

Abstract

Code review is an essential element of any mature software development project, it is key to ensuring the long-term quality of the code base. Code review aims at evaluating code contributions submitted by developers before they are committed into the project's version control system. Code review is considered to be one of the most effective QA practices for software projects. In principle, the code review process should improve the quality of committed code changes. However, in practice, the execution of this process can still allow bugs to enter into the codebase unnoticed.

Moreover, the notion of the quality of the code review process is not limited to the quality of the source code that passed a review. It goes beyond that, the quality of the code review process can affect how successful a software development project is. For instance, in the world of open source software (OSS), a particular execution code review process may encourage or deter the contributions from "external" developers, the people who are essential to OSS projects.

We claim that by analyzing various software artifacts as well as assessing developers' daily experience, we can create models that represent the established code review processes and highlight potentially weak points in their execution. Having this information, the stakeholders can channel the available resources to address the deficiencies in their code review process. To support such a claim, we perform the following studies.

First, we study the tool-based code review processes of two large OSS projects that use traditional model of evaluating code contributions. We analyse the software artifacts extracted from the issue tracking systems to understand what can affect code review response time and eventual outcome. We found that code review is affected not only by technical factors (e.g., patch size, priority, etc.) but also by non-technical ones (e.g., developers' affiliation, their experience, etc.).

Second, we investigate the quality of contributions that passed the code review process and explore the relationships between the reviewers' code inspections and a set of factors, both personal and social in nature, that might affect the quality of such inspections. By mining the software repository and the issue tracking system of the Mozilla project, as well as applying the SZZ algorithm to detect bug-inducing changes, we were able to find that 54% of the reviewed changes introduced bugs in the code. Our findings also showed that both personal metrics, such as reviewer workload and experience, and participation metrics, such as the number of involved developers, are associated with the quality of the code review process.

Third, we further study the topic of code review quality by studying the developers' attitude and perception of review quality as well as the factors they believe to be important. To accomplish this, we surveyed 88 Mozilla core developers, and applied grounded theory to analyze their responses. The results provide developer insights into how they define review quality, what factors contribute to how they evaluate submitted code and what challenges they face when performing review tasks.

Finally, we examined the code review processes executed in a completely different environment — an industrial project that uses pull-based development model. Our case study was Active Merchant project developed by Shopify Inc. We performed a quantitative analysis of their software repository to understand the effects of a variety of factors on pull request review time and outcome. After that, we surveyed the developers to understand their perception of the review process and how it is different from developers' perception in traditional development model.

The studies presented in this thesis focus on code review processes performed by projects of different nature — OSS vs. industrial, traditional vs. pull-based. Nevertheless, we observed similar patterns in the execution of code review that the stakeholder should be aware of to maintain the long-term health of the projects.

Acknowledgements

Unlike many other students, I was blessed to have not one but two awesome co-supervisors — Olga Baysal and Mike Godfrey — and I would like to start by saying 'thank you' to them. I was a little bit scared when I was starting my PhD — I was moving thousands miles away from my home to a country I had never been to and knew very little about; and to add to this, Mike was about to start his year-long sabbatical in Europe! However, despite the time difference and not being physically present on campus, he was able to take care of all the questions and problems I had during my first year. Once he returned, he continued to be extremely supportive throughout my program, and he was able to find the right balance between giving me the academic freedom and ensuring that I did not feel "free" too much. Mike taught me how to better in academic writing (and I hope I have learned something from him) and he was always willing to listen to and comment on my practice talks before the conferences.

When I first met Olga, she was a PhD student herself though not a rookie one like I was. She became my friend, my colleague, and later my supervisor. As a friend, she gave me numerous pieces of advice about life in Canada and was always there during my ups and downs. We worked together on many different studies, and she was always the first person I would go to if I needed an opinion about the slides I prepared for my talks. As a supervisor, she guided me in the academic world and helped me to shape the direction of my research. Overall, I could not have wished for better supervisors. This thesis would not have been possible without Olga and Mike, thank you for this!!!

I would also like to thank my examination committee members — Dr. Mei Nagappan, Dr. Derek Rayside, Dr. Krzysztof Czarnecki, and Dr. Thomas Zimmermann — for providing useful comments that helped me improve my thesis.

Thanks to Margaret Towel and Paula Roser for their help and assistance with the intricacies of the administrative world. Special thanks go to Wendy Rush for our friendly conversations and her infinite patience with my travel claims.

I would like to acknowledge my undergrad and Master's supervisor, Dmytro A. Kliushyn. Dmytro introduced me to the world of academic research, planted the seeds of knowledge, and taught me many skills that I was using during my PhD. Although I did not continue my work with him, he was very supportive in my PhD application process and (I believe) was a driving force behind my acceptance at Waterloo.

Special thanks to my best friend Andrii Soviak. We did our undergrad and Master's together, and then shared our PhD journeys although at different universities. Every time I was feeling down, Andrii was able to find the right words to cheer me up so I could get

up and continue the journey. It is hard to explain how grateful I am for having such a friend like him.

Finally, I would like to thank my family — my parents Lilia and Borys, my grandmother Valentina, my brother Volodymyr and his wife Maria, and my sister Olga and her husband Pavel — for their support and unconditional love.

To my family.

Table of Contents

List of Tables xix

List of Figures xxi

1 Introduction 1

1.1 Thesis Statement and Research Problems . . . 3

1.2 Contributions . . . 5

1.3 Thesis Organization . . . 6

2 Background and Related Work 7

2.1 Code Review and Contribution Management . . . 7

2.2 Software Quality and Metrics Used to Predict It . . . 9

2.2.1 Identification of Faulty Changes . . . 10

2.3 Code Review and Software Quality . . . 12

2.4 Code Review in Pull-Based Development Model . . . 13

2.5 Bug Reporting and Fixing . . . 13

2.6 Data Analysis . . . 14

2.6.1 Statistical Analysis . . . 15

2.6.2 Machine Learning . . . 15

2.6.3 Grounded Theory . . . 16

3 Factors Affecting Code Review Time and Patch Acceptance 19

3.1 Introduction 20
3.2 Lifecycle Analysis 21
3.3 Methodology 24
3.3.1 Data Extraction 25
3.3.2 Data Pre-Processing 26
3.3.3 Determining Independent Factors 27
3.3.4 Data Analysis 28
3.4 The Case Studies 28
3.4.1 WebKit 29
3.4.2 Blink 40
3.5 Discussion 45
3.5.1 WebKit and Blink Comparison 45
3.5.2 Other Interpretations 46
3.5.3 Threats to Validity 47
3.6 Summary 48

4 Bugginess as a Measure of Code Review Quality 51

4.1 Motivation and Research Questions 52
4.2 The Mozilla Code Review Process 53
4.3 Methodology 54
4.3.1 Studied Systems 55
4.3.2 Data extraction 55
4.3.3 Linking Patches and Commits 57
4.3.4 Data Pre-Processing 58
4.3.5 Identifying Bug-Inducing Changes 60
4.3.6 Determining Explanatory Factors 61
4.3.7 Model Construction and Analysis 62

4.4 Results 63
4.5 Threats to Validity 69
4.6 Conclusion 70

5 Developers' Perception of Code Review Process **73**
5.1 Study Overview 74
5.2 Methodology 75
5.2.1 Survey Design 75
5.2.2 Participants 76
5.2.3 Survey Data Analysis 77
5.3 Results 78
5.3.1 RQ1: How do Mozilla developers conduct code review? 78
5.3.2 RQ2: What factors do developers consider to be influential to review time and decision? 80
5.3.3 RQ3: What factors do developers use to assess code review quality? 84
5.3.4 RQ4: What challenges do developers face when performing review tasks? 90
5.4 Discussion 93
5.5 Threats and Limitations 94
5.6 Summary 94

6 Code Review in Pull-Based Development **97**
6.1 Introduction 98
6.2 Methodology 100
6.2.1 Data Mining 100
6.2.2 Explanatory Factors 102
6.2.3 Data Analysis 103
6.2.4 Survey Design and Participants 104
6.2.5 Card Sorting 105

6.3 Results 105
6.3.1 RQ1: Merge types and their effect on review time. 106
6.3.2 RQ2: Factors affecting merge time and decision. 107
6.3.3 RQ3: How developers perform and assess PR review process 112
6.4 Discussion 117
6.5 Threats to Validity 118
6.6 Conclusions 118

7 Conclusions **121**
7.1 Code Review Process Recommendations 123
7.2 Future Work 125
7.3 Summary of Contributions 126

References **129**

APPENDICES **143**

A Survey of Mozilla Developers **145**

B Survey of Shopify Developers **151**

List of Tables

3.1 Overview of the factors studied. 27
3.2 Effect of factors on response time and positivity 29
3.3 Overview of the numerical factors in WebKit. 30
3.4 Response time for organizations participating on the WebKit project. . . . 37
3.5 Response time for WebKit patch reviewers and writers. 37
3.6 Response time for organizations participating on the Bink project. 37
3.7 Response time for Blink patch reviewers and writers. 37
3.8 Overview of the numerical factors in Blink. 41

4.1 Overview of the studied systems. 55
4.2 A taxonomy of considered technical, personal, and participation metrics used. 59
4.3 Number of code reviews that missed bugs. 64
4.4 MLR models for code review bugginess. 65

5.1 The list of categories that emerged during open coding. 79

6.1 Overview of the factors studied. 103
6.2 Classification of PRs by a merge type. 107
6.3 Models for PR review time and acceptance. 108
6.4 The categories created during open coding. 113

List of Figures

3.1 Mozilla Firefox's patch lifecycle. 23
3.2 WebKit's patch lifecycle. 23
3.3 Blink's patch lifecycle. 23
3.4 Number of revisions for each size group. 31
3.5 Overview of the participation of top five organizations in WebKit. 34
3.6 Acceptance and rejection time. 35
3.7 Positivity values by organization. 38

4.1 Process overview. 54

5.1 Factors influencing code review time. 81
5.2 Factors influencing code review decision. 83
5.3 Factors influencing code review quality. 89

6.1 Factors influencing PR review time. 109
6.2 Factors influencing PR review outcome. 111
6.3 Factors influencing PR review quality. 116

Chapter 1

Introduction

Building and maintaining software systems is expensive in terms of both costs and developer time. While each project stores large volumes of information in different forms — such as source code, bug reports, developer discussions, documentation, etc. — usually much of this information is not available in a form that is immediately useful to various stakeholders to answer questions related to development decisions. Consequently, stakeholders cannot easily base their decisions on the available raw data; indeed, they have to rely largely on their experience, their intuition, and/or the experience of others that they think might be applicable to the current problem.

Moreover, modern software development is a complex social activity. With software systems becoming larger, and developers being distributed across the globe due to different reasons such as company having multiple offices, outsourcing, or a project being open source, the social dimension of software development begins to increase its impact on the process. In addition, new software development methods, such as agile software development, have emerged that put extra emphasis on interactions between all involved parties — customers, mangers, and developers. Measuring the impact of the social dimension of software development is difficult because traditional artifacts, such as source code and bug reports, usually contain only technical information that cannot be used to detect or trace events and decisions that were influenced or triggered by that dimension.

Although the development of each software product is a complicated process with its unique set of requirements, there is one problem that the stakeholders were, are, and will be concerned about — the quality of that software product. One key element that is adopted by virtually every mature software development project in order to address the question of quality is *code review* — the proactive evaluation of code contributions submitted by

developers. Code review is often thought of as one of the most effective practices for ensuring quality and success of the resulting software system; it has been shown to be an effective way of identifying defects in the code changes before they are committed into the project's code base [39]. Reviewers, who are viewed as the gatekeepers of a project's master repository, must carefully validate the design and implementation of new contributions to ensure they meet the project's quality standards.

Initially, code review was presented by Fagan as a formalized and structured process [39]. Software inspection, the name used by Fagan, consists of six sequential phases; each member of an inspection team is assigned one of the four roles that matches their expertise. The actual inspection happens during lengthy group meetings — the team members follow the checklists, thoroughly analyse the source code line-by-line, and fill out several forms when they find defects. Although the inspections were shown to be an effective quality assurance technique [35, 108, 109], they are very heavy on resources and developers' time, and they have not received wide adoption.

Nowadays, many companies and open source software projects perform code review in more relaxed settings: it is less formal in nature, and each review is usually done by a single person, typically a senior developer on the core team. In addition, developers use different tools (e.g., Gerrit) and/or environments (e.g., GitHub) to help themselves with code review tasks. This lightweight approach is also known as Modern Code Review [8].

While some may view modern code review as purely a technical process, we believe that it is fundamentally a social process. In Fagan's code review settings, the technical nature of code review might have overshadowed the social one due to its formality and strict structure. However, such a transition (i.e., from formal and structured to informal and relaxed) will lessen the effect of technical aspects of code review while emphasizing on the social ones. For example, during modern code review, the identity of the patch's author is known to a reviewer, therefore a review and especially the "perceived" quality of the patch might be affected by the amount of knowledge the reviewer has about the patch's author.

Although there is agreement on the positive value that code review has on software development, the implementation of this process is different among projects. The projects differ not only in the policies that govern which changes must be reviewed, and who should perform such reviews but also in a development model (i.e., classic vs. pull-based model) used that implicitly affects the review process. Once the process is established, stakeholders have little means to know how this process affects or is affected by the ongoing software development. In principle, the code review process should improve the quality of code changes before they are committed to the project's master repository. However,

in practice, the execution of this process can still allow bugs to enter into the codebase unnoticed. Although stakeholders have access to all the raw data available to them, it can offer them little help in understanding how the process should be adjusted to improve the quality of the systems they are building.

Because the reviewers stand in the way of the new code contributions to the project's master repository, it is crucial that the reviewers and stakeholders understand how code review affects the needs of the project. For example, in case of open source software, the need for the contributions from "outsiders" might be of the highest importance, so the stakeholders must ensure that the established code review process does not discourage newcomers from making contributions.

Since the investigation of code review processes involves extracting and processing data from a variety of sources, much of the foundational previous research comes from the field of mining software repositories (MSR). Mining software repositories refers to extracting data from multiple sources such as version control systems, bug trackers, discussion boards, Q&A websites, etc. These sources accumulate a variety of artifacts during the lifetime of a project. By studying these artifacts, one can gain knowledge about the evolution of the project, as well as discover meaningful relationships among them. Using mining software repositories techniques — which come from areas such as data mining, machine learning, and statistical analysis — as a workhorse for getting the structured data from different sources, and applying statistical analysis to that gathered data, we can seek to provide facts that are useful, understandable, and address developers' or managers' needs. For example, by analyzing the structured, mined data from the version control repository and the issue tracking system, we can look for factors that slow down the review process; in turn, developers might choose to adjust their day-to-day routines.

1.1 Thesis Statement and Research Problems

This dissertation is focused on the topic of code review quality. To be precise, the thesis of this dissertation is that by analyzing various software artifacts as well as assessing developers' daily experience, we can create models that represent the established code review processes, help evaluate their quality, and highlight potentially weak points in their execution. Having this information, the stakeholders can channel the available resources to address the deficiencies in their code review process.

In our opinion, the definition of the quality of the code review process depends on the stakeholder's point of view, and therefore, the quality can be measured using different

metrics, both quantitative and qualitative in nature, that are tied to a particular definition. For example, reviewers might associate the quality with the number of defects that passed a review while developers might associate the quality with the time it takes to review their code contributions, as well as the likelihood of their patches being accepted.

To support the thesis statement, we study code review in different contexts and address four main research problems:

1. **Understanding the factors affecting code review time and outcome**
   When submitting patches for code review, individual developers are primarily interested in maximizing the chances of their patch being accepted in the least time possible. In principle, code review is a transparent process in which reviewers aim to assess the qualities of the patch on its technical merits in a timely manner; however, in practice the execution of this process can be affected by a variety of factors, some of which are external to the technical content of the patch itself. While the length of the review process might not be of a big interest for developers in the industrial context, it is crucially important for open source projects. In such projects, lengthy reviews might discourage contributions from the non-core developers, and as a result these projects might become less successful.

2. **Understanding the quality aspect of code review**
   One of the key reasons for establishing a code review process is ensuring that new changes to a software project meet the established quality standards and do not introduce defects into existing source code. Unfortunately, in practice, software bugs are sometimes unintentionally missed by the reviewers. Reviewing a code change is not a trivial task for developers — they must carefully check for any mistakes, check that the proposed code adheres to best practices, consider possible impact on the existing code base, etc. Characteristics of a particular code change and the circumstances under which the review is performed can make the reviewer's job easier or harder. In this work, we explore the relationships between the reviewers' code inspections and a set of factors, both personal and social in nature, that might affect the quality of such inspections.

3. **Understanding developer perspective on what review quality means to them**
   Software development is mainly done by humans; therefore, inherently there will be parts of code review process that are heavily influenced by the people performing it. However, software-related data sources — such as version control repositories and issue tracking systems — cannot reveal much about what challenges developers face

when they conduct code review tasks. Developers' perception of code review is a missing piece of the overall picture of review quality. It is needed to find the weaknesses within the review process and to drive the development of new processes to support developers' work. In this work, we investigate how the developers themselves define code review quality, what factors contribute to how they evaluate submitted code, and what challenges they face when performing review tasks.

4. **Code review in a pull-based development model**
   Pull-based software development model has gain a lot popularity among software developers. Hosting services such as GitHub and Bitbucket attracted a huge number of new and existing projects — GitHub alone is estimated to have more than 19.4 million active repositories [48]. Unlike traditional model of evaluation code contributions, in pull-based development model developers make changes to an isolated copy of the project's repository, create a pull request that represent such changes, and submit it to the project for evaluation. It is vital to understand whether pull request review (code review of changes in pull-based model) differs from the traditional code review in factors that affect its quality. Having such knowledge, we can identify similar patterns of code review execution.

1.2 Contributions

Here we highlight the main contributions of this thesis:

- We identified three sets of factors — technical, personal, and organizational — that influence the duration of code review as well as the outcome of that review in two big OSS projects [15].
- We showed that a large percentage of code changes that successfully passed the review process still contain defects [71].
- We investigated which aspects contribute to poor code review quality and found that size of a patch, the number of affected files, the presence of a second reviewer, reviewer workload and experience, as well as developers participation in the discussion of a patch are the factors that affect the effectiveness of code review [71].
- We surveyed professional Mozilla developers and found that they believe that factors such as the experience of developers, the choice of a reviewer, size of a patch, its

quality and rationale affect the time needed for review; while bug severity, code quality and its rationale, presence and quality of tests, and developer personality impact review decisions [70].

- We explored how developers perceive the quality of the code review process, and the problems developers face during code review and found that developer perception of code review quality is shaped by their experience and defined as a function of clear and thorough feedback provided in a timely manner by a peer with a supreme knowledge of the code base, strong personal and inter-personal qualities [70].
- We investigated the factors that affect the timeliness and outcome of pull request reviews and found that a pull request size, the discussion, as well as author experience and affiliation influence both review time and review decision [72].
- We explored how developers perceive pull request quality and found that their perception is defined by pull request description, complexity, and revertability. Pull request review quality is seen as a function of constructive feedback, quality of tests, and generated discussion between author and reviewer [72].

Overall, our work presents an in-depth analysis of code review quality in a variety of settings. We observed similar patterns in the execution of code review that the stakeholder should be aware of to maintain the long-term health of the projects.

1.3 Thesis Organization

The rest of this thesis is organized as follows. Chapter 2 presents the existing research relevant to the topic of this thesis, and provides an overview of data analysis techniques our work relies on. Chapter 3 describes our study of the effects that several sets of factors might have on the code review response time and eventual outcome. In Chapter 4, we present our quantitative study of code review quality. Chapter 5 focuses on developers' and their perception of review quality. Chapter 6 examines the quality of pull request reviews and analyzes the developers' perspective on code review in pull-based development model. Finally, Chapter 7 concludes the thesis by discussing its main contributions as well as highlighting directions of future work.

Chapter 2

Background and Related Work

There is much previous work related to our goal of understanding code review and improving current practices. The research problems of this work corresponds to three research areas: *code review*, *software quality*, and *bug fixing*. In this chapter, we provide an overview of the existing research for each studied problem, as well as present data analysis methods and techniques that can be used to address these problems.

2.1 Code Review and Contribution Management

A large body of work has attempted to assess modern code review as practised in the development of large software systems. Mockus et al. [86] were among the pioneer researchers who studied open source development. By analyzing the Mozilla and Apache projects, they identified the main characteristics of open source communities such as the dependency on the contributions from outside developers, and developers being free to choose tasks to work on. Rigby and German [104] presented a first investigation of code review processes as practised within four-open source projects: GCC, Linux, Mozilla, and Apache. They show the existence of a number of review patterns and quantitatively analyzed the review process of the Apache project. One of the interesting patterns they found is that sometimes the contributions from "outsiders" (i.e., external developers) are simply rewritten by the reviewers instead of providing the authors with a list of comments. Rigby et al. [105] examined peer review techniques used by the Apache server project namely, review-then-commit and commit-then-review. They suggested a series of metrics that produce measures similar to those used in traditional inspection experiments. Specifically, they measured the

frequency of review, the level of participation in reviews, the size of the artifact under review, the calendar time to perform a review, and the number of reviews that find defects. Their findings have shown that Apache adopt a broadcast-based style of code review where reviews are frequent reviews of small, independent, complete contributions conducted by a group of self-selected experts. Later, Rigby and Storey [106] investigated the mechanisms employed by developers of five open source projects to identify code changes they are competent to review. They explored the way stakeholders interact with one another during the code review process. Their findings provide insights to developers about how to effectively manage large quantities of reviews. Additionally, their investigation reveals that the identification of defects is not the sole motivation for modern code review. In effect, other motivations exist including resolving non-technical issues with code under review such as feature, scope, or process issues. For example, they found that some features were not incorporated into the code base simply because no developer was willing to maintain them.

Weissgerber et al. [124] performed data mining on email archives of two open source projects to study patch contributions. They found that the probability of a patch being accepted is about 40% and that smaller patches have higher chance of being accepted than larger ones. They also reported that if patches are accepted, they are normally accepted quickly (61% of patches are accepted within three days). Jiang et al. [62] studied, through the analysis of the Linux Kernel, the relation between patch characteristics and the probability of patch acceptance as well as the time taken for patches to be integrated into the code base. The results of their study showed that developer experience, patch maturity, and prior subsystem churn affect the patch acceptance, while reviewing time is impacted by submission time, the number of affected subsystems, the number of suggested reviewers, and developer experience.

In a qualitative study done at Microsoft [8], Bachelli and Bird studied the motivations, challenges, and outcomes of their code review process. This investigation revealed that while finding defects remains the main motivation for review, other motivations exist as well. For example, they found that for 40% of the interviewed developers, the main motivation behind the code review is code improvement (i.e., not bug-related changes regarding readability, consistency, and comments in the code, etc). Another motivation, identified by 17% of developers, is finding alternative solution to a submitted piece of source code. Rigby and Bird [103] did a quantitative study of six industry-led projects and seven OSS projects; while those projects differed in problem domain, team culture, and development processes, the study found that the code review processes employed by the projects had similar characteristics.

Thongtanunam et al. [119] analyzed three open source projects — Android, Qt, and OpenStack — to find the common characteristics of patches that suffer from low code review participation, i.e., patches that are not selected for a review, patches that are not discussed during the review, and patches that receive a delayed feedback. They found that the number of reviewers of previous versions of a patch and the number of days since the last version of a patch correlate with the likelihood of a patch being selected for a review. The length of patch description as well as the length of prior discussions have a relationship with the likelihood of a patch undergoing a discussion. Finally, the delay of feedback on prior patches as well as the type of a change correlate with the feedback delay of the current patch.

2.2 Software Quality and Metrics Used to Predict It

Researchers have studied a large variety of metrics for predicting the defect-proneness of source code, including technical metrics (e.g., lines of code and number of code chunks), organizational metrics (e.g., number of involved developers and distance between them), and process metrics (e.g., number of previous bugs).

A recent work by Kamaie [64] empirically evaluated a real-time approach, called "Just-In-Time Quality Assurance", to identify potentially risky changes. Their study evaluated change-level prediction through the analysis of six open-source and five commercial projects. They found that process metrics (such as number of past defects) are more accurate than product metrics (such as cyclomatic complexity) when software quality assurance effort is considered. Kim et al. [66] classified changes as being defect-prone or "clean" based on the use of the identifiers in added and deleted source code and the words in change logs. Eyolfson et al. [38] analyzed the relation between a change bugginess and the time of the day the change was committed and the experience of the developer who made the change. They found that changes performed between midnight and 4 AM are more buggy than changes committed between 7 AM and noon, and that developers who regularly make their commits introduce less buggy changes.

Several other metrics have been used to predict defects. For example, Graves et al. [53] rely on the use of change history-based process metrics — such as the number of past defects and number of developers — to build defect prediction models. Jiang et al. [61] have compared the performance of design and code metrics in predicting fault-prone modules. Their work has shown that code-based models are better predictors of fault-prone modules than design-level models. A study by Moser et al. [90] found that process metrics perform similarly to code metrics when predicting defect-prone files in the Eclipse project.

Zimmermann et al. [131] have focused on investigating defect prediction from one project to another using seven commercial projects and four-open source projects. They found no single factor that produced accurate predictions.

Nagappan et al. demonstrated that organizational metrics — such as the number of developers working on a component, organizational distance between developers, as well as organizational code ownership — are better predictors of defect-proneness than traditional measures such as churn, complexity, coverage, dependencies, and pre-release bug metrics [96]. These findings agrees with Conway's law [29], which assume that a software system's design reflects the structure of the organization that develops it.

Rahman and Devanbu suggested the use of defect prediction models to compare the impact of product and process metrics [102]. The results of their research suggest that code metrics are generally less useful than process metrics for prediction. In related work, the same authors found that lines of code involved in a bug fix are more strongly associated with contributions from a single developer than contributions from many developers. This finding suggests that code review is an essential part of the software quality assurance [101]. Mende and Koschke [82] have proposed effort-aware bug prediction models to help allocate software quality assurance efforts including code review. The suggested models factor in the effort required to perform code review or test code when evaluating the effectiveness of prediction models, resulting in more realistic performance evaluations.

Recent works have also investigated source code ownership for software quality. Bird et al. find measures of ownership — such as the number of low-expertise developers, and the proportion of ownership for the top owner — have a relationship with both pre-release faults and post-release failures [20]. Matsumoto et al. have shown that their suggested metrics of ownership (e.g., the number of developers and the code churn generated by each developer) are also good indicators of defect-prone source code files [79].

Existing research indicates that personal factors such as ownership, experience, organizational structure, and geographic distribution significantly impact on software quality. Understanding these factors, and properly allocating human resources can help managers enhance quality outcomes. We hypothesize that a modern code review process often fails to identify buggy changes and that this may be due to several families of factors, including technical, personal, and organizational.

2.2.1 Identification of Faulty Changes

Many studies on assessing defect-proneness of code changes require preliminary knowledge of whether a particular commit is a bug-introducing change or not. To label all changes as

clean or *buggy*, two problems must be solved: identify all changes that fix bugs, and locate the changes that led to those fixes.

The first problem is straightforward, and there are two common ways to address it; both approaches are based on the analysis of the textual description of a commit. The choice depends on the system being studied and the standards adopted by developers of that system. One approach is to search for special keywords like *bug*, *fix*, or *patch* [64, 87]. Another approach is to use regular expressions to find references to the entries in the bug tracking systems, such as "*#1234*" which might be the unique identifier of an issue being tracked by Bugzilla [40, 45, 66, 122]. Śliwerski et al. proposed to use a combination those approaches together with the analysis of bug reports to improve the precision of the process [115].

The second problem has proven to be harder to solve; as yet, there is no approach that provides high precision and recall. However, the SZZ algorithm developed by Śliwerski et al. is the most widely used technique for locating bug-introducing changes within the research community [115]. For each commit that is a bug fix, the algorithm calculates the textual `diff` between the revision of the commit and the previous revision. The output of `diff` corresponds to the list of lines that were added and/or removed between the two revisions. The SZZ algorithm ignores added lines and considers removed lines as locations of bug-introducing changes. Next, the Mercurial `annotate` command (similar to `blame` in Subversion and Git) is executed for the previous revision. For each line of code, `annotate` adds the identifier of the most recent revision that modified the line in question. SZZ extracts revision identifiers for each bug-introducing line found at the previous step, and builds the list of revisions that are candidates for bug-inducing changes. Finally, the algorithm eliminates those candidates that were added to the repository after the bug associated with a commit was reported to the issue tracking system. The remaining revisions are marked as bug-inducing code changes.

Several researchers worked on improving the original algorithm. Kim et al. addressed some limitations of the SZZ algorithm as it may return imprecise results if `diff` contains changes in comments, empty lines, or formatting [67]. By using annotation graphs and filtering out non behaviour and format changes, they managed to improve both precision and recall of the algorithm by 36% and 14% respectively. Williams et al. suggested further improvements tailored towards Java systems [126]. They replaced annotation graphs with a line-number mapping approach [127], and the heuristics for identification of "cosmetic" changes with a syntax-aware diff tool for Java called DiffJ.

SZZ was successfully applied to understand whether refactorings induce bug-fixes [9], as well as to build prediction models that focus on identifying defect-prone software changes

[64,66]. The investigation of Śliwerski et al. to the Mozilla and Eclipse open-source projects shows that defect-introducing changes are generally a part of large transactions and that defect-fixing changes and changes done on Fridays have a higher chance of introducing defects.

2.3 Code Review and Software Quality

Although modern code review has received a significant attention from researchers recently, there is little empirical evidence on how effective code review is at detecting bugs and the extent to which code review is related to factors such as personal ones (e.g., reviewers expertise), technical (e.g., patch characteristics), or temporal (e.g., review time).

Kemerer and Paulk [65] looked into the effect of the code review rate on the reviewers' ability to catch problems in new contributions, as well as on the quality of software products, while controlling for a number of potential confounding factors. As a result of their study, they recommended that to best ensure review quality, reviewers should not proceed faster than 200 LOC per hour.

Recently, McIntosh et al. [80] empirically investigated the relationship between software quality, code review coverage, and code review participation. They found that low code review coverage and participation produce components with up to five additional post-release defects. Thongtanunam et al. [118] studied the code review of defective files in the Qt open source project. They found that both historically defective files and files with future defects reviewed with less scrutiny, have lower team participation, and bigger rate of review than the defect-free files and files without future defects respectively. The results of these studies confirm that poor code review negatively affect software quality.

Mäntylä and Lassenius classified the types of defects found in review on university and three industrial software systems [75] suggesting that code reviews may be most valuable for long-lived software products as the value of discovering evolvability defects in them is greater than for short-lived systems.

Hatton [56] found relevant differences in defect finding capabilities among code reviewers — the "worst" reviewer is ten times less effective than the best reviewer. Moreover, he found almost 50% improvement in defects detection between settings where the source code is inspected by two developers together (76% of faults found) and where the source code is inspected by two developers separately (53% of faults found).

2.4 Code Review in Pull-Based Development Model

Gousious et al. [50] were among the pioneers in research into pull-based development. They quantitatively investigated OSS projects hosted on GitHub to learn the factors that are influential to PR review time as well as to the acceptance of PRs; they found that the merge time is affected by several factors, including the developer's track record and the test coverage in the project, while the PR acceptance is primarily influenced by the "hotness" of code (i.e., the number of recent changes) that PR proposes to modify.

Tsay et al. [121] investigated the code contributions on GitHub to measure the effect of social and technical factors on the likelihood of a contribution being accepted; they found lengthier discussions tended to lead to rejection of PRs, while the developer's previous involvement in the project increased the likelihood of acceptance.

Gousious et al. [52] surveyed the integrators (developers responsible for assessing/merging incoming contributions) of GitHub projects to understand their perspective on pull-based development practices; they found that the integrators face multiple challenges, such as maintaining project quality and deciding which PRs to prioritize.

Marlow et al. [76] studied how core developers from GitHub projects form their opinions of the incoming contributions; they found that integrators use signals such as the contributor's history of coding activity as well as their actions on GitHub (e.g., following other developers).

In another study, Gousious et al. [51] surveyed the most active contributors on GitHub to learn their work practices and the challenges they face; they found that contributors are eager to maintain awareness of the projects to avoid submitting duplicate PRs, and that they communicate changes using PRs as well as issue trackers, emails, and instant messages. The main challenge identified by the contributors is poor responsiveness from core developers.

2.5 Bug Reporting and Fixing

Bug fixing is a large part of the ongoing maintenance and evolution of any long-lived software system. Often, this is aided by a tool, such as Bugzilla or JIRA, that provides support for bug reporting, triage, assignment, discussion, code review, and resolution. Indeed, Mozilla uses Bugzilla for all their projects as a central system for handling all bug-related activities such as bug reporting, bug assignment, discussions of patches, code review, etc. In the case of large projects, a large number of bug reports (e.g., 400 per day

for Mozilla project) are added to those tracking systems every day [123]. As a result of this constant flow of bug reports, the bug triage process takes a lot of effort and time. Most of the research on bug fixing can be linked to one of the three topics: what kinds of bugs the developers should focus on [130], who is the best developer to fix a particular bug [4, 6, 84], and what information should be included in a bug report [18, 68].

A variety of tools and approaches were proposed by the studies focused on automatic assignment of bugs to developers [5, 6, 10, 23, 32]. Usually those tools and approaches are based on information retrieval techniques [23] when textual information from the bug report (e.g., bug description, possible exception messages, class names, etc.) and previous source code changes are used to identify developers who have expertise relevant to that bug. Other tools are based on machine learning techniques instead. Such tools use different classifiers (such as Naive Bayes) trained on a variety of data — such as developers' previous experience and bug descriptions — to predict and recommend developers for fixing a particular bug [6, 32, 60, 129].

Once the bug is fixed, a patch, i.e., the source code change that fixes the problem, is submitted to the issue tracking system. While existing research provides tools and approaches for triaging the bugs, to the best of our knowledge, there are no studies or tools that address the problem of finding an appropriate developer to perform code review of a submitted patch. We plan to investigate the code review process as well as the factors that affect its quality, and to provide the right developers with the right information needed (e.g., which reviewer is an expert for a particular patch, or a warning to a reviewer that the patch's author has low experience in writing patches for that system component) so each path could undergo code review of the highest quality.

2.6 Data Analysis

To better understand the current practice of code review and investigate possible new approaches and tools for handling it, we need to transform the abundance of raw data that comes from a variety of different sources (version control systems, bug tracking systems, release history, email lists, documentation and policies, etc.) into condensed pieces of usable information by extracting, pre-processing, and analyzing the data using different techniques and tools. In this section we describe the main categories of techniques that can be used to address our research problems: statistical analysis, machine learning, and grounded theory.

2.6.1 Statistical Analysis

Researchers use statistics to describe, model, analyze and interpret data [97, 107, 114]. There are two main statistical methodologies: descriptive and inferential statistics. Descriptive statistics aim to summarize a given sample, i.e., a dataset, using measures such as the mean, median, standard deviation, and skewness. Inferential statistics are used to draw conclusions about statistical populations observing sampled datasets. Both of these methodologies are an integral part of data mining.

In empirical studies, researchers often work with multiple datasets and test proposed hypotheses about a population. Therefore in such studies, statistical analysis plays a crucial role because it can provide a meaningful interpretation of the study results [36]. There are two ways in which statistical analysis can be used to interpret the data in question: describing and comparing the data, and making predictions. By looking at the distribution of values, we can describe the shape and the expected range of values of a population, as well as find outliers in the observed dataset. These observations tell us about the underlying data and help us to model that data in the most mathematically appropriate way. However, researchers rarely collect data simply to report the descriptive statistics since these values usually do not have high scientific value. Researches often find themselves comparing two or more datasets (i.e., the distributions behind those datasets) or models built upon the observed datasets. There are multiple statistical tests available for comparing the distributions: the t-test, the Kolmogorov-Smirnov test, ANOVA, the Wilcoxon-Mann-Whitney test, the Kruskal-Wallis test, etc. The selection of one test over another is usually driven by the nature of the observed data (which includes the facts about the data that can be obtained from the descriptive statistics). Software-related data might not follow the normal (Gaussian) distribution or any other probability distribution; therefore, non-parametric tests (the tests that do not make any assumptions about the probability distribution of the studied data), such as the Kolmogorov-Smirnov test, the Wilcoxon-Mann-Whitney test, or the Kruskal-Wallis test, are especially useful. Although parametric tests might be more robust, they are tied to datasets that follow the distributions that the tests were designed for, i.e., applying parametric tests to the datasets with the "wrong" distribution will lead to spurious results.

2.6.2 Machine Learning

Machine learning provides a powerful set of tools for learning from and making data-driven decisions and predictions from empirical data [21, 99, 128]. Machine learning is useful for our research because it can utilize the available historical data to build classification or

prediction models. The training data (i.e., the past data) is used to construct (train) a classifier (such as linear regression, logistic regression, decision trees, support vector machines, etc.) that later is used for classification or prediction on the new data. There are also two main types of machine learning: unsupervised and supervised. In unsupervised machine learning, the training data is not labelled, and thus the learning algorithm tries to identify the structure (i.e., to find clusters) inside given data. Contrary to the unsupervised, in supervised machine learning, every data point from the training dataset is labelled with "desired value". By processing the training data, the learning algorithm builds a function that can be used to label (essentially predict) "unseen" data points (the data points that are not from the training set).

Machine learning has been applied to a number of software engineering research problems. Aversano et al. [7] proposed an approach for predicting whether a new code change is buggy or not. The bug-introducing changes were identified from the commit history and were used to train the classifiers. In their work, the authors also evaluated different classifiers (K-Nearest neighbor, simple logistic regression, multi-boosting, decision trees, and support vector machines) and found that K-Nearest neighbor algorithm yields significantly better trade-offs between precision and recall. Anvik et al. [6] used a support vector machine algorithm to build a classifier for recommending the list of developers for resolving a bug. Past bug reports and project-related heuristics were used to train the classifier in the study. Cubranic et al. [32] used a Naive Bayes classifier to automatically assign bugs to the developers. Similar to Biugie et al. [22] and Panjer [98], Hosseinni [60] studied the prediction of bug lifetimes and compared different classifiers (0-R, 1-R, Naive Bayes, decision tree, logistic regression) that are suitable for this purpose. The study shows that Naive Bayes algorithm outperforms other classifiers by around 2%.

Similar to McIntosh [80] and others [24, 85, 110], in our research on code review quality [71], we used multiple linear regression models to explore the relationship between the explanatory variables (technical, personal, and participation factors of code review) and the dependent variable (the quality of code review). From the analysis of the built models, we were able to identify factors that have statistically significant effect on defect-proneness of code review, as well as to tell whether each factor was associated with positive or negative effect.

2.6.3 Grounded Theory

A lot of data in software engineering can be analyzed using quantitative methods (for example, for source code we can compute different metrics such as size or complexity).

However, a lot of data also comes from sources such as interviews, observations, surveys, blogs, emails, and documentation, and it is more appealing to analyze it using qualitative methods rather than quantitative ones because we can answer broader *how* and *why* questions. Grounded theory method is a systematic methodology in the social sciences involving the discovery of theory through the analysis of data [77]. The idea behind grounded theory approach is to start with the data and develop a substantive theory by going through an iterative and rigorous process of data analysis and theoretical analysis [49]. The key moment here is that researchers start with no theory at all and develop it during the application of the approach. The goal of grounded theory is to generate concepts and categories that emerge from the raw data and are connected to the reality. Grounded theory analyses the data with no preconceived ideas. It is useful when we have questions of the form "what is happening here?, or when we want to learn how people understand and handling particular situations or tasks.

Open coding [30, 31, 83] is the part of grounded theory that deals with identifying, naming, categorizing, and describing phenomena found in the data. Written data from notes or transcripts are conceptualized line by line. Each line, sentence, paragraph etc. is read to understand what it is about and what it is referring to [49, 116]. As a result, open coding generates concepts from the data that a future theory will be based on. Open coding is a repetitive process, a researcher goes back and forth while comparing data, constantly modifying, and improving the developing theory. The researcher continues to collect and examine the data as long as patterns continue to emerge.

The software engineering research community has successfully applied grounded theory on a wide range of software engineering problems. Adolph et al. proposed a model for applying grounded theory for software engineering research, as well as showed the application of grounded theory for studying how people manage software development process [1, 2]. Similar to Coleman et al. [28], Montoni et al. used grounded theory to study the success of Software Process Improvement (SPI) implementations [88]. They built three categories of factors as well as five groups of actions that are critical to the success of SPI implementations. Grounded theory was also used by Hoda et al. to study the human aspects of software engineering [59]. The authors studied how the teams that follow Agile software development self-organize in practise. They identify six informal roles that developers have in Agile teams, as wells practices that help them in self-organization. Dagenais et al. studied project landscapes and integration experience new developers go through when they are joining a new project [33]. Using grounded theory they identified the factors that affect the experience of newcomers, as well as provided recommendations for improving it. Souza et al. used grounded theory to study how individual software developers perform change impact analysis [34]. They identified 21 strategies that developers adopted to minimize

the impact each developer has on others, as well as to incorporate others' impact into the current task they are working on.

Chapter 3

Factors Affecting Code Review Time and Patch Acceptance

When submitting patches for code review, individual developers are primarily interested in maximizing the chances of their patch being accepted in the least time possible. In principle, code review is a transparent process in which reviewers aim to assess the qualities of the patch on its technical merits in a timely manner; however, in practice the execution of this process can be affected by a variety of factors, some of which are external to the technical content of the patch itself. In this chapter, we describe two empirical studies of the code review processes for large, open source, and industry-led projects (WebKit and Google Blink) to learn the effect of different factors on code review time and patch acceptance.

Chapter Organization. In Section 3.1, we first provide an introduction describing the studies done as well the the research questions we tried to answer. Section 3.2 presents patch lifecycle analysis by comparing the Mozilla Firefox, WebKit, and Blink lifecycle models; this is followed in Section 3.3 by a description of the methodology we used in the empirical studies. Section 3.4 presents the two case studies: WebKit (Section 3.4.1) and Blink (Section 3.4.2). Section 3.5 interprets the results and addresses threats to validity. Finally, Section 3.6 summarizes our results.

Related publication. The work described in this chapter has been published in the following paper[1]:

[1] My role in this work included web scraping, data pre-processing and cleaning, and writing.

- Olga Baysal, Oleksii Kononenko, Reid Holmes, and Michael W. Godfrey. Investigating technical and non-technical factors influencing modern code review. *Empirical Software Engineering*, 21(3):932–959, 2016.

3.1 Introduction

Many software development projects employ code review as an essential part of their development process. Code review aims to improve the quality of source code changes made by developers (as patches) before they are committed to the project's version control repository. In principle, code review is a transparent process that aims to evaluate the quality of patches objectively and in a timely manner; however, in practice the execution of this process can be affected by many different factors, both technical and non-technical.

Existing research has found that organizational structure can influence software quality. Nagappan et al. demonstrated that organizational metrics (number of developers working on a component, organizational distance between developers, organizational code ownership, etc.) are better predictors of defect-proneness than traditional metrics such as churn, complexity, coverage, dependencies, and pre-release bug measures [96]. These findings provide support for Conway's law [29], which states that a software system's design will resemble the structure of the organization that develops it.

In this chapter, we have performed empirical studies to gain insight into the different factors that can influence how long a patch takes to get reviewed and a patch's likelihood of being accepted. The factors we analyzed include personal and organizational relationships, patch size, component, bug priority, reviewer/submitter experience, and reviewer load. Since software developers are primarily interested in getting their patches accepted as quickly as possible, we have designed our research questions to align with this perspective:

RQ1 *What factors can influence how long it takes for a patch to be reviewed?*
Previous studies have found that smaller patches are more likely to receive faster responses [62,105,124]. We replicate these studies on our data, and extend the analysis to a number of other potential factors.

RQ2 *What factors influence the outcome of the review process?*
Most studies conclude that small patches are more successful in landing to the project's codebase [105,124]. A recent study showed that developer experience, patch maturity and prior subsystem churn play a major role in patch acceptance [62]. We

further extend these results with additional data that includes various non-technical factors.

In this work, we study the community contributions and industrial collaboration on the WebKit and Google Blink open source projects. WebKit is a web browser engine that powers the Apple's Safari and iOS browsers, was the basis for Google's Chrome and Android browsers, and host of other third-party browsers. WebKit is a particularly interesting project as many of the organizations that collaborate on the project — including Apple, Google, Samsung, and Blackberry — also have competing business interests. In April 2013 — and during the execution of our initial study — Google announced that they had created and would subsequently maintain their own fork of WebKit, called Blink. Therefore, when extending our previous work [14], we have decided to investigate the differences in the velocity of the code reviews on the patches submitted to the Blink project and compare findings of the two case studies.

3.2 Lifecycle Analysis

WebKit is an HTML layout engine that renders web pages and executes embedded JavaScript code. The WebKit project was started in 2001 as a fork of the open source KHTML project. At the time of our study, developers from more than 30 companies actively contributed to this project; Google and Apple were the two primary contributors, submitting 50% and 20% of patches respectively. Individuals from Adobe, BlackBerry, Digia, Igalia, Intel, Motorola, Nokia, Samsung, and other companies were also active contributors.

The WebKit project employs an explicit code review process for evaluating submitted patches; in particular, a WebKit reviewer must approve a patch before it can "land" in (i.e., be incorporated into) the project's version control repository. The set of official WebKit reviewers is maintained through a system of voting to ensure that only highly-experienced candidates are eligible to review patches. A reviewer will either accept a patch by marking it `review+` or ask for further revisions from the patch owner by annotating the patch with `review-`. The review process for a particular submission may include multiple iterations between the reviewer and the patch writer before the patch is ultimately accepted and lands in the version control repository.

Since WebKit is an industrial project, we were particularly interested to compare its code review process to that of other open source projects. To do so, we extracted the WebKit's patch lifecycle (Figure 3.2) and compared it with the previously studied patch

lifecycle of Mozilla Firefox [13] (Figure 3.1). The patch lifecycle captures the various states patches undergo during the review process, and characterizes how the patches transition between these states. The patch lifecycles enable large data sets to be aggregated in a way that is convenient for analysis. For example, we were surprised to discover that a large proportion of patches that have been marked as accepted are subsequently resubmitted by authors for further revision. Also, we can see that rejected patches are usually resubmitted, which might ease concerns that rejecting a borderline patch could cause it to be abandoned.

While the set of states in our patch lifecycle models of both WebKit and Firefox are the same, WebKit has fewer state transitions; this is because the WebKit project does not employ a 'super review' policy [93]. Also, unlike in Mozilla, there are no self-edges on the `Accepted` and `Rejected` states in WebKit; this is because Mozilla patches are often reviewed by two people, while WebKit patches receive only individual reviews. Finally, the WebKit model introduces a new edge between `Submitted` and `Resubmitted`; WebKit developers frequently "obsolete" their own patches and submit updates before they receive any reviews at all. One reason for this behaviour is that submitted patches can be automatically validated by the external test system; developers can thus submit patches before they are to be reviewed to see if they fail any tests. All together, however, comparing the two patch lifecycles suggests that the WebKit and Firefox code review processes are fairly similar in practice.

Blink's patch lifecycle is depicted in Figure 3.3, which shows that 40% of the submitted patches receive positive reviews and only 0.3% of the submitted patches are rejected. Furthermore, a large portion of patches (40.4%) are resubmitted. This is because Blink developers often update their patches prior to receiving any reviews; as with WebKit, this enables the patches to be automatically validated. At first glance, outright rejection does not seem to be part of the Blink code review practice; the `Rejected` state seems to under-represent the number of patches that have been actually rejected. In fact, reviewers often leave comments for patch improvements, before the patch can be accepted.

The model also illustrates the iterative nature of the patch lifecycle, as patches are frequently `Resubmitted`. The edge from `Submitted` to `Landed` represents patches that have been merged into Blink's source code repository, often after one or more rounds of updates. Developers often fix "nits" (minor changes) after their patch has been approved, and land the updated version of the patch without receiving additional explicit approval. The lifecycle also shows that nearly 10% of patches are being neglected by the reviewers (i.e., `Timeout` transition); `Timeout` patches in Blink can be considered as "informal" rejects.

Comparing the patch lifecycle models of WebKit and Blink, we noticed that Blink has fewer state transitions. In particular, the edges from the `Accepted` and `Rejected` back

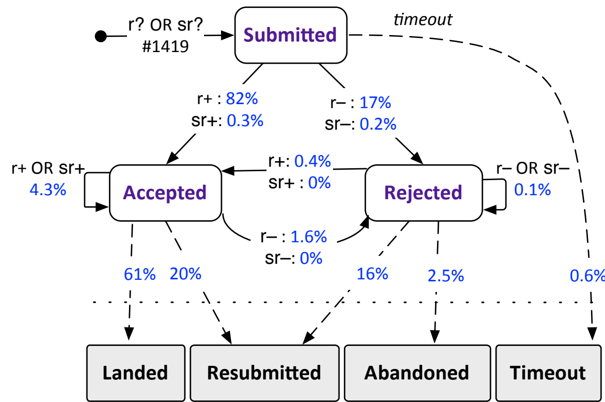

Figure 3.1: Mozilla Firefox's patch lifecycle.

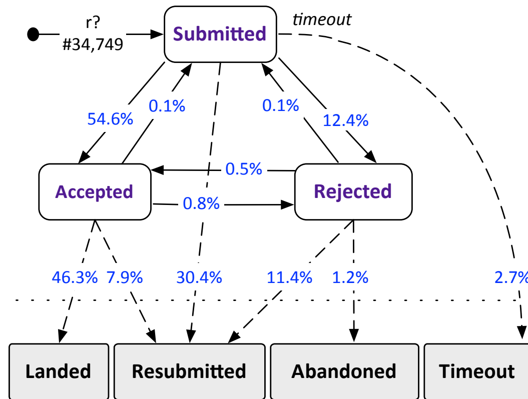

Figure 3.2: WebKit's patch lifecycle.

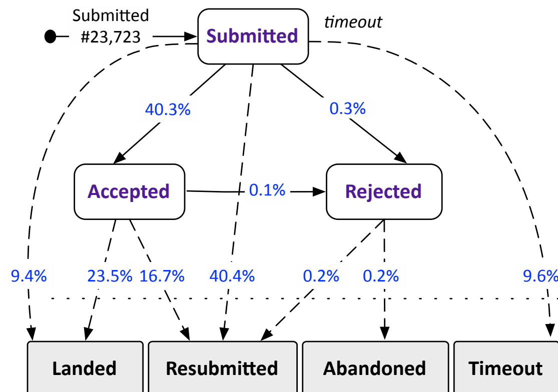

Figure 3.3: Blink's patch lifecycle.

to `Submitted` are absent in Blink. Since Blink does not provide any indication of the review request on patches, we had to reverse engineer this information for all patches by considering the timestamps on each item (patch) in the series. We automated this process by putting the `Submitted` label to the patch at the time the patch was filed to the issue repository.

Blink also accepts a smaller portion of patches (about 40% of all contributions compared to the WebKit's 55% of submitted patches), yet officially rejects less than 1%. While timeouts are more frequent for Blink patches than WebKit ones, timeouts can be viewed as "unofficial" rejects in the Blink project where disapprovals are uncommon.

Blink appears to exhibit a larger portion of patches being resubmitted (a 10% increase compared to the WebKit patches), including resubmissions after patches are successfully accepted (16.7%).

Finally, a new edge is introduced between `Submitted` and `Landed`, accounting for those contributions that were committed to the code base without official approval from the reviewers; these cases typically represent patch updates. Both WebKit and Blink developers frequently "obsolete" their own patches and submit updates before they receive any reviews at all.

Comparing the two patch lifecycle models suggests that the WebKit and Blink code review processes are similar in practice; at the same time, it appears that Google's review policy may not be as strict as the one employed by Apple on the WebKit project. While no code can be checked into trunk without successfully passing code review, no formal approval is required for trivial patch updates.

3.3 Methodology

To investigate our research questions, we first extracted code review data from the Bugzilla issue tracking system (for the WebKit project) and the Chromium code review repository (for the Blink case study); we then pre-processed the data, identified factors that may affect review delays and outcomes, and performed our analysis. To avoid repetition in explaining two similar processes, we describe each step of our approach for the first case study only. For any deviations in the methodology of the Blink project (e.g., description of the extracted dataset and data filtering), we refer readers to the beginning of the Section 3.4.2.

3.3.1 Data Extraction

Every patch contributed to the WebKit project is submitted as an attachment to the project's issue repository[2]; we extracted this data by "scraping" Bugzilla for all public patches submitted between April 12, 2011 and December 12, 2012. We use the same time interval value as in our previous study on Firefox [13] to be able to compare code review processes of two projects. The data we retrieved consists of 17,459 bugs, 34,749 patches, 763 email addresses, and 58,400 review-related flags. We tracked a variety of information about issues such as name of the person who reported the issue, the date the issue was submitted, its priority and severity, as well as a list of patches submitted for the issue. For each patch, we saved information regarding its owner, submission date, whether a patch is obsolete or not, all review-related flags along with the files affected by the patch. For each patch we also recorded the number of lines added and removed along with the number of chunks for each changed file. All details were stored in a relational database.

All fields in the database, except those related to affected files, were extracted directly from the issue tracker. To create the list of affected files, we needed to download and parse the individual patches. Each patch file contains one or more `diff` statements representing changed files. In our analysis we ignored `diff` statements for binary content, e.g., images, and focused on textual `diffs` only. From each statement we extracted the name of changed file, number of lines marked added and removed, and number of code chunks. Here a code chunk is a block of code that represents a local modification to a file as it defined by the `diff` statement. We recorded total number of lines added and removed per file in total, and not separately for each code chunk. We did not try to interpret the number of changed lines from the information about added/removed lines.

Almost every patch in our database affects a file called `ChangeLog`. Each `ChangeLog` file contains description of changes performed by the developer for a patch and is prepared by the patch submitter. Although patch files contain `diff` statements for `ChangeLog` files and we parsed them, we eliminated this information when we computed the size.

There are three possible flags that can be applied to patches related to code review: `review?` for a review request, `review+` for a review accept, and `review-` for a review reject. For each flag change we also extracted date and time it was made as well as an email address of the person who added the flag.

As the issue tracker uses email addresses to identify people, our initial data set contained entries for many individuals without names or affiliations. Luckily, the WebKit team

[2] `https://bugs.WebKit.org/`

maintains a file called `contributors.json`[3] that maps various developer email addresses to individual people. We parsed this file and updated our data, reducing the number of people in our database to 747.

We next determined developers' organizational affiliations. First, we parsed the "WebKit Team" wiki webpage[4] and updated organizational information in our data. We then inferred missing developers' affiliations from the domain name of their email addresses, e.g., those who have an email at "apple.com" were considered individuals affiliated with Apple. In cases where there was no information about organization available, we performed a manual search on the web. For those individuals where we could not determine an affiliated company, we set `company` field to "unknown"; this accounted for 18% of all developers but only 6% of patches in our data.

3.3.2 Data Pre-Processing

In our analysis we wanted to focus as much as possible on the key code review issues within the WebKit project. To that end we performed three pre-processing steps on the raw data:

1. We focused only on the patches that change files within the `WebCore` portion of the version control repository. Since WebKit is cross-platform software, it contains a large amount of platform-specific source code. The main parts of WebKit that all organizations share are in WebCore; these include features to parse and render HTML and CSS, manipulate the DOM, and parse JavaScript. While the platform-specific code is actively developed, it is often developed and reviewed by a single organization (e.g., the Chromium code is modified only by Google developers while the RIM code is modified only by the Blackberry developers).

   Therefore we looked only at the patches that change non-platform-specific files within WebCore; this reduced the total number of patches considered from 34,749 to 17,170. We also eliminated those patches that had not been reviewed, i.e., patches that had only `review?` flag. This filter further narrowed the input to 11,066 patches.

2. To account for patches that were "forgotten", we removed slowest 5% of WebCore reviews. Some patches in WebCore are clear outliers in terms of review time; for example, the slowest review took 333 days whereas the median review was only 76

[3] `http://trac.WebKit.org/browser/trunk/Tools/Scripts/WebKitpy/common/config/contributors.json`

[4] `http://trac.WebKit.org/wiki/WebKit%20Team`

minutes. This filter excluded any patch that took more than 120 hours (≈5 days), removing 553 patches. 10,513 patches remained after this filter was applied.

3. To account for inactive reviewers, we removed the least productive reviewers. Some reviewers performed a small number of reviews of WebCore patches. This might be because the reviewer focused on reviewing non-WebCore patches or had become a reviewer quite recently. In ordering the reviewers by the number of reviews they performed, we excluded those developers performed only 5% of the total reviews. This resulted in 103 reviewers being excluded; the 51 reviewers that remained each reviewed 31 patches or more. This resulted in an additional 547 patches being removed from the data.

The final dataset consists of 10,012 patches and was obtained by taking the intersection of the three sets of patches described above.

3.3.3 Determining Independent Factors

Previous research has suggested a number of factors that can influence review response time and outcome [62, 105, 124]. Table 3.1 describes the factors (independent variables) that were considered in our study and tested to see if they have an impact on the dependent variables such as *time* and *outcome (positivity)*. We grouped the factors into two categories: *technical* and *non-technical*. Our choice of selecting independent factors is determined by the availability of the data stored in the projects' issue tracking systems.

Independent Factor	Type	Description
Patch Size	technical	number of LOC added and removed
Component	technical	top-level module in /Source/WebCore/
Priority	technical	assigned urgency of resolving a bug
Organization	non-technical	organization submitting or reviewing a patch
Review Queue	non-technical	number of pending review requests
Reviewer Activity	non-technical	number of completed reviews
Patch Writer Experience	non-technical	number of submitted patches

Table 3.1: Overview of the factors studied.

Based on the advice of WebKit developers, we identified the WebKit component the patch changes by examining the patches directly rather than using the issue tracker component. This was because the issue tracker was frequently incorrect.

WebKit does not employ a formal patch assignment process; in order to determine review queues of individual reviewers at any given time, we had to reverse engineer patch assignment and answer the following questions:

- *When did the review process start?* We determined the date when a request for a review was made (i.e., `review?` flag was added to the patch). This date was referred as "review start date". While there might be some delay from this to the time the reviewer started working on the patch, we have no practical means of tracking when the developer actually received the request or started to perform the review in earnest.
- *Who performed code review of a patch?* The reviewer of a patch is defined as the person who marked the patch with either `review+` or `review-`. Having this, we added the assignee to each review request.

We computed a reviewer queue by considering the reviews a developer eventually completed. The review queue is defined as the number of patches that were 'in flight' for that developer at the time a patch was submitted.

3.3.4 Data Analysis

Our empirical analysis used a statistical approach to evaluate the degree of the impact of the independent factors on the dependent variables. First, we tested our data for normality by applying Kolmogorov-Smirnov tests [78]. For all samples, the $p < 0.05$, showing that the data is not normally distributed. We also graphically examined how well our data fits the normal distribution using Q-Q plots. Since the data is not normally distributed, we applied non-parametric statistical tests: Kruskal-Wallis analysis of variance [73] for testing whether the samples come from the same distribution, followed by a post-hoc non-parametric Mann-Whitney U (MWW) test [74] for conducting pairwise comparisons.

All our reported results including Figures and Tables are statistically significant with the level of significance defined as $p < 0.05$.

3.4 The Case Studies

Ultimately, we investigated the impact of seven distinct factors on the code review process both in terms of response time and review outcome or positivity for the WebKit and Blink projects; this is summarized in Table 3.2.

Factor	WebKit		Blink	
	Time	Positivity	Time	Positivity
Patch Size	✓	N/A	✓	N/A
Priority	✓	✓	N/A	N/A
Component	✓	×	✓	×
Organization	✓	✓	✓	×
Review Queue	✓	✓	×	×
Reviewer Activity	✓	×	✓	×
Patch Writer Experience	✓	✓	✓	✓

Table 3.2: Effect of factors on response time and positivity:
✓- statistically significant; × - not statistically significant.

3.4.1 WebKit

To answer our research questions, we performed two empirical studies. We start with demonstrating the results of our analysis of the WebKit dataset and highlighting its main findings. The overview of the numerical factors is summarized in Table 3.3.

Patch Size

The size of the patch under review is perhaps the most natural starting point for any analysis, as it is intuitive that larger patches would be more difficult to review, and hence require more time; indeed, previous studies have found that smaller patches are more likely to be accepted and accepted more quickly [124]. We examined whether the same holds for the WebKit patches based on the sum of lines added and removed as a metric of size taken from the patches.

To determine the relationship between patch size and the review time, we performed a (non-parametric) Spearman correlation. The results showed that the review time was weakly correlated to the patch size, r=0.09 for accepted patches and r=0.05 for rejected patches, suggesting that patch size and response time are only weakly related, regardless of the review outcome.

With a large dataset, outliers have the potential to skew the mean value of the data set; therefore, we decided to apply two different outlier detection techniques: Pierce's criterion and Chauvenet's criterion. However, we found that removal of the outliers did not improve the results, and we ultimately rejected their use.

Factor	Min	Median	Mean	Max
Patch Size	1	31	131.7	25928
Number of Components	0	1	1.691	18
Review Queue	0	0	0.566	11
Reviewer Activity	1	281	401.7	1955
Patch Writer Experience	1	68	113.3	836

Table 3.3: Overview of the numerical factors in WebKit.

Next we split the patches according to their size into four equal groups: A, B, C, and D where each group represents a quarter of the population being sampled. Group A refers to the smallest patches (0–25%) with the average size of 4 lines, group B denotes small-to-medium size changes (25–50%) on average having 17 lines of code, group C consists of the medium-to-large changes (50–75%) with the mean of 54 LOC, and group D represents the largest patches (75–100%) with the average size of 432 lines. A Kruskal-Wallis test revealed a significant effect of the patch size group on acceptance time ($\chi^2(3)$=55.3, $p < 0.01$). Acceptance time for group A (the median time is 39 minutes, the mean is 440 minutes) is statistically different compared to the time for groups B (with the median of 46 minutes and the mean of 531 minutes), C (the median of 48 minutes and the mean of 542 minutes) and D (the median is 64 minutes, the mean time is 545 minutes).

In terms of review outcome, we calculated the positivity values for each group A–D, where we define positivity as $\texttt{positivity} = \sum\texttt{review+} \ / \ (\sum\texttt{review-} + \sum\texttt{review+})$. The median values of positivity for groups A–D are 0.84, 0.82, 0.79, 0.74 respectively. Positivity did decrease between the quartiles, matching the intuition that reviewers found more faults with larger patches, although this result was not significant.

However, review time for a single patch is only part of the story; we also wanted to see whether smaller patches undergo fewer rounds of re-submission. That is, we wanted to consider how many times a developer had to resubmit their patch for additional review. We calculated the number of patch revisions for each bug, as well as the size of the largest patch. Figure 3.4 illustrates the medians of the patch revisions for each size group, the median of the revisions for group A and B is 1, for group C is 2, and for D is 3. The results show that patch size has a statistically significant, strong impact on the rounds of revisions. Smaller patches undergo fewer rounds of revisions, while larger changes have more re-work done before they successfully land into the project's version control repository.

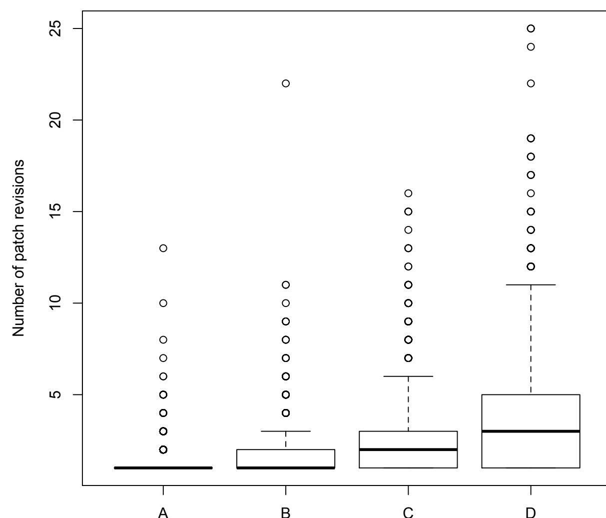

Figure 3.4: Number of revisions for each size group.

Priority

A bug priority is assigned to each issue filed with the WebKit project. This field is created to help developers define the order in which bugs should be fixed[5]. The Webkit project defines five priority levels, ranging from the most important (`P1`) to the least important (`P5`). We were surprised when we computed the distribution of patches among priority levels: `P1` – 2.5%, `P2` – 96.3%, `P3` – 0.9%, `P4` and `P5` – 0.1% each. Looking at these numbers one might speculate that the priority field is not used as intended. Previous work of Herraiz et al. also found that developers use at most three levels of priority and the use of priority/severity fields is inconsistent [58]. The default value for priority is `P2`, which might also explain why the vast majority of patches have this value assigned. Also, in our discussion with WebKit developers we found that some organizations maintain internal trackers that link to the main WebKit bug list; while the WebKit version has the default priority value, the internal tracker maintains the organization's view on the relative priority. In our analysis we discarded priorities `P4` and `P5` because they did not have enough patches.

A Kruskal-Wallis test demonstrated a significant effect of priority on time ($\chi^2(2)$=12.70, $p < 0.01$). A post-hoc test using Mann-Whitney tests with Bonferroni correction showed the significant differences between P1 and P3 with median time being 68 and 226 minutes respectively, $p < 0.05$, and between P2 and P3 with median time being 62 and 226 minutes

[5] `https://bugs.webkit.org/page.cgi?id=fields.html`

respectively, $p < 0.01$. While patches with priority P2 receive faster response that the ones with P1, the difference is not statistically significant.

To analyze positivity, we considered each review by a developer at a given priority and computed their acceptance ratio. To reduce noise (e.g., the data from reviewers who only reviewed one patch at a level and hence had a positivity of 0 or 1), we discarded those reviewers who reviewed only four or fewer patches for a given priority.

We found a statistically significant correlation between priority levels and positivity ($\chi^2(2)$=10.5, $p < 0.01$). The difference of the review outcome for patches of P1 (median value is being 1.0) compared to the ones of P2 (median is 0.83) is statistically significant ($p < 0.01$), indicating that patches of higher priority are more likely to land to the project's codebase. Although reviewers are more positive for patches of higher priority, we caution about the interpretation of these results because the vast majority of patches are `P2`.

Component

WebCore represents the layout, rendering, and DOM library for HTML, CSS, and SVG. WebCore consists of several primary components (`bindings`, `bridge`, `css`, `dom`, `editing`, `html`, `inspector`, `page`, `platform`, and `rendering`).

While it is natural to assume that some components may be more complex than others, we wanted to find out whether contributions to certain components are more successful or are reviewed more quickly. To answer this, we selected the components that undergo the most active development: `inspector` (1,813 patches), `rendering` (1,801 patches), `html` (1,654 patches), `dom` (1,401 patches), `page` (1,356 patches), `bindings` (1,277 patches), and `css` (1,088 patches).

The difference in the response time between components was statistically significant ($\chi^2(6)$=29.9, $p < 0.01$), in particular the `rendering` component takes longer to review (the median time is 101 minutes) compared to `bindings` (72 minutes), `inspector` (58 minutes), and `page` (58 minutes). The difference in reviewing time of patches submitted to the `page` and `dom` components was also significant with the medians being 58 minutes vs. 91 minutes respectively.

Although the positivity values vary among components and range between 0.73–0.84, we found no relation between positivity and the component. From the developer's perspective, we can tell that it is more difficult for developers to land a patch to `page` (the value of positivity is 0.73), while patches to `inspector` are more likely to be successful (the value of positivity is 0.84).

Review Queue Length

Our previous qualitative study of Mozilla's process management practices found that developers often try to determine current work loads of reviewers prior to making a decision as to who would be the best choice to request a review from [11]. Thus, we investigated the relationship between review queue size and review response time, expecting to find that reviewers having shorter queues would provide quicker reviews.

We calculated queue sizes for the reviewers at any given time (the process is described in Section 3.3.3). The resulting queues ranged from 0 to 11 patches.

Since the average queue was 0.6 patches, we distributed patches into three groups according to the queue size: shortest queue length ranging from 0–1 patches (group A), medium length consisting of 2–3 patches (group B) and longer queues ranging from 4–11 patches (group C).

We found a significant effect of review queue size on reviewing time ($\chi^2(2)$=15.3, $p < 0.01$). The medians of queue size for group A, B and C are being 0, 2, and 5 patches respectively. A post-hoc test showed significant differences between group A and group C (with median time being 63 and 158 minutes respectively, $p < 0.01$) and group B and C (with median time being 90 and 158 minutes respectively, $p < 0.05$).

Studying the impact of the queue size on the reviewer positivity (with the Kruskal-Wallis effect being $\chi^2(2)$=15.8, $p < 0.01$), we found a significant difference between A and C groups (the median positivity being 0.84 and 1 respectively, $p < 0.01$), as well as B and C groups (with median positivity being 0.88 and 1.0 respectively, $p < 0.05$).

Thus, we found that the length of the review queue influences both the delay in completing the review as well as the eventual outcome: the shorter the queue, the more likely the reviewer is to do a thorough review and respond quickly; and a longer queue is more likely to result in a delay, but the patch has a better chance of getting in.

Organization

Many companies that participate in the WebKit development are business competitors. An interesting question is whether patches are considered on their technical merit alone or if business interests play any role in the code review process, for instance by postponing the review of a patch or by rejecting a patch for a presence of minor flaws. We have considered the top five organizations that contribute patches to the WebKit repository. Figure 3.5 provides an overview of the participation of these organizations on the project

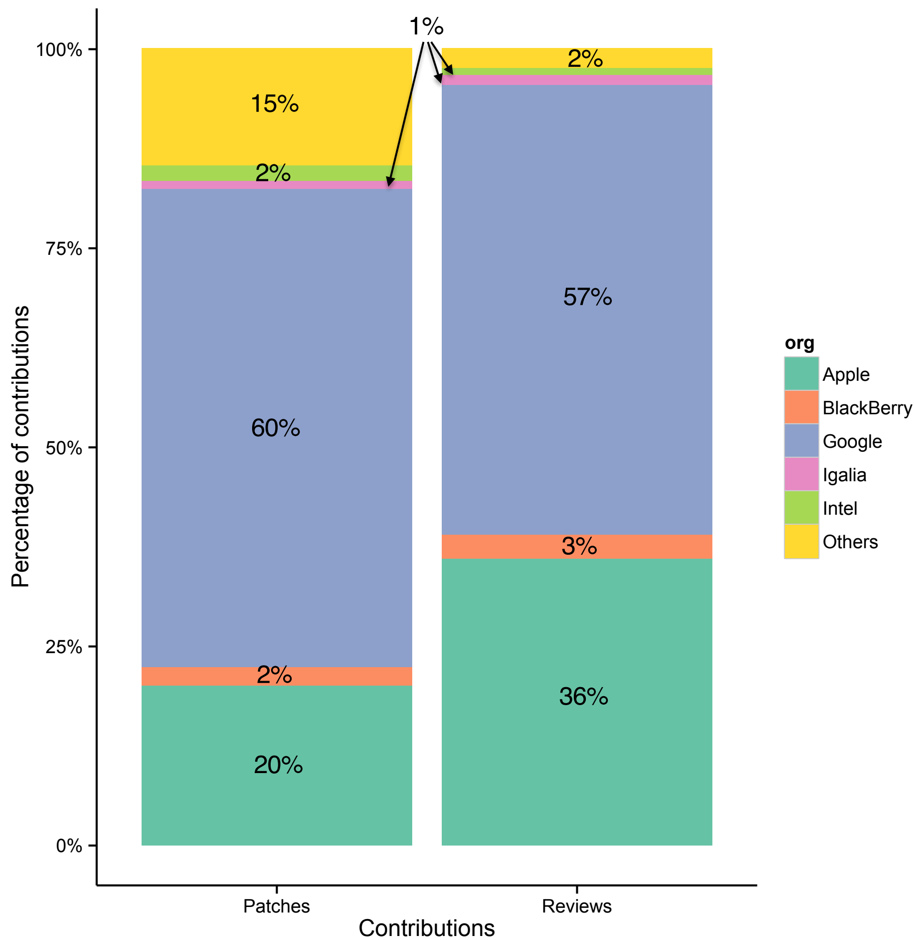

Figure 3.5: Overview of the participation of top five organizations in WebKit.

with respect to the percentage of the total patches they submit and the percentage of the patches they review. It is clear that two companies play a more active role than others; Google dominates in terms of patches written (60% of total project's patches) and patches reviewed (performing 57% of all reviews) while Apple submits 20% of the patches and performs 36% of the reviews. While we analyzed all possible pairs of organizations (36 of them; the top 5 organizations + "Others"), for the sake of brevity we discuss only Apple, Google, and "the rest".

Figure 3.6 represents review time for each pair of organizations. The first letter in the label encodes a reviewer's affiliation, the second encodes submitter's affiliation; for example, A-G represents Apple reviewing a Google patch. Analysis of the patches that

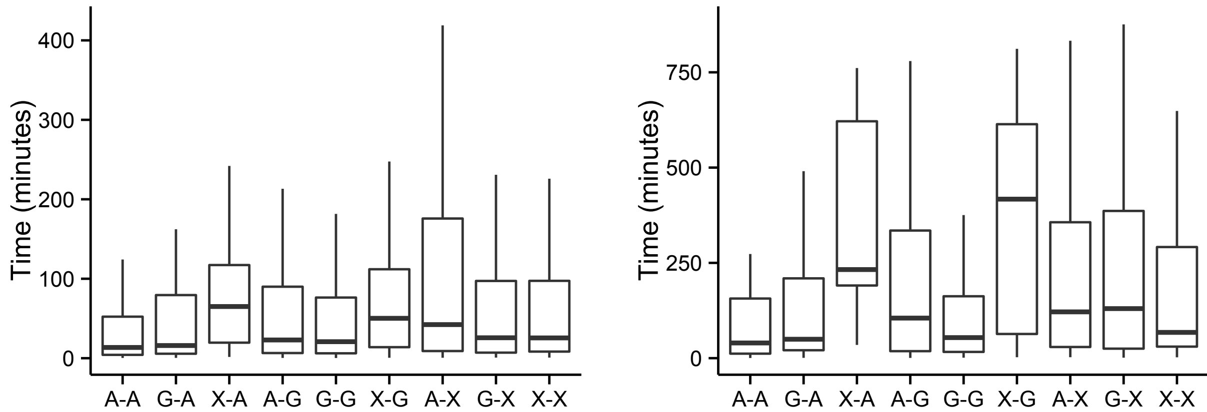

Figure 3.6: Acceptance time (left), rejection time (right). Organization: A=Apple, G=Google, X=Rest.

received a positive review showed that there is a correlation between review time and the organization affiliated with the patch writer.

To identify where the correlation exists, we performed a series of pair-wise comparisons. We discovered that there is a statistically significant difference between how Apple approves their own patches (`A-A`) and how Google approves their own patches (`G-G` column). Another statistically significant difference was found between time Apple takes to accept their own patches and time it takes to accept Google patches (`A-G`). However, we found no statistical difference in the opposite direction — between the time for Google to accept their own patches compared to patches from Apple (`G-A`).

The correlation between review time and company was also found for patches that received a negative review. The pair-wise comparison showed almost the same results: statistical difference between Apple-Apple and Apple-Google, and no statistical difference between Google-Google and Google-Apple. At the same time the difference between Apple-Apple and Google-Google is no longer present. Based on these findings, it appears that Apple treats their own patches differently from external patches, while Google treats external patches more like their own. Pairs involving "the rest" group exhibited no statistically significant differences for both review decisions.

Since statistical tests can report only a presence of statistical difference, we also report the means and medians of review time required for each company pair (Table 3.4). To ease comparison of the differences in the response time for organizations, patch reviewers and writers between the two projects, we placed Tables 3.4, 3.5, 3.6, and 3.7 on one page (p. 37). According to the data, Apple is very fast in reviewing its own patches, but is relatively

slow in reviewing Google patches (3–4 times difference in medians, 1.5–2 times difference in means). At the same time Google exhibits the opposite behaviour, i.e., provides faster response to the patches from Apple than their own developers. While both means and medians are almost the same for positive reviews, the median and the mean values of review time for negative review for Apple patches are 20 and 200 minutes less respectively than for Google own patches.

To compute the positivity of various organizations, we cleansed the data as we did for the priority analysis above; we removed any reviewer who had reviewed less than 10 patches (i.e., removed 5% of least active reviewers) to avoid an overabundance of positivities of 0 or 1. The box plot with this filter applied is shown in Figure 3.7. Statistical tests showed that there is a correlation between the outcome of the review and patch owner's affiliation ($\chi^2(2)$=10.7, $p < 0.01$). From the pair-wise comparison, we found that there is statistically significant difference between positivity of Apple reviewers towards their own patches (`A-A` column) compared to the patches of both Google (`A-G` column) and "the rest" (`A-X` column). The other pair that was statistically different is positivity of Google reviewers between their own patches (`G-G` column) and patches from "the rest" (`G-X` column).

Quantitatively, there are some interesting results. First, the positivity of Apple reviewers towards their own patches clearly stands out (the median is ≈0.92). Possible explanations for this include that there is a clear bias among Apple reviewers, or that Apple patches are of extreme quality, or that Apple applies some form of internal code review process. We also observed that both Apple and Google are more positive about their own patches than 'foreign' patches; while this could be a systematic bias, Apple and Google are also the two most experienced committers to WebKit and this may account for this difference. Finally, the positivity of Apple reviewers towards Google patches (the median is ≈0.73) is lower than the positivity of Google reviewers towards Apple patches (the median is ≈0.79).

Reviewer Activity

WebCore has 51 individuals performing code reviews of 95% of patches. The breakdown of the reviewers by organization is as follows: 22 reviewers from Apple, 19 reviewers from Google, 3 reviewers from BlackBerry, Igalia and Intel are being represented by one reviewer each, and 5 reviewers belong to the group "others". Comparing reviewing efforts, we noticed that while Apple is predominant in the number of reviewers, it reviews only 36% of all patches; by comparison, Google developers perform 57% of the total number of reviews. Since WebKit was originally developed and maintained by Apple, it is perhaps unsurprising

Reviewer → Writer	Accepted		Rejected	
	Median	Mean	Median	Mean
Apple → Apple	25	392	60	482
Apple → Google	73	617	283	964
Google → Google	42	484	102	737
Google → Apple	45	483	80	543

Table 3.4: Response time (in minutes) for organizations participating on the WebKit project.

Group	Reviewer		Writer	
	Median	Mean	Median	Mean
A	84	621	102	682
B	76	634	76	632
C	46	516	43	491
D	57	496	48	478

Table 3.5: Response time (in minutes) for WebKit patch reviewers and writers.

Reviewer → Writer	Accepted		Rejected	
	Median	Mean	Median	Mean
Google → Google	57	385	169	716
Google → Other	95	473	428	737
Other → Other	66	351	n/a	n/a
Other → Google	48	399	n/a	n/a

Table 3.6: Response time (in minutes) for organizations participating on the Bink project.

Group	Reviewer		Writer	
	Median	Mean	Median	Mean
A	71	434	106	547
B	71	490	51	384
C	42	338	59	394
D	91	362	56	287

Table 3.7: Response time (in minutes) for Blink patch reviewers and writers.

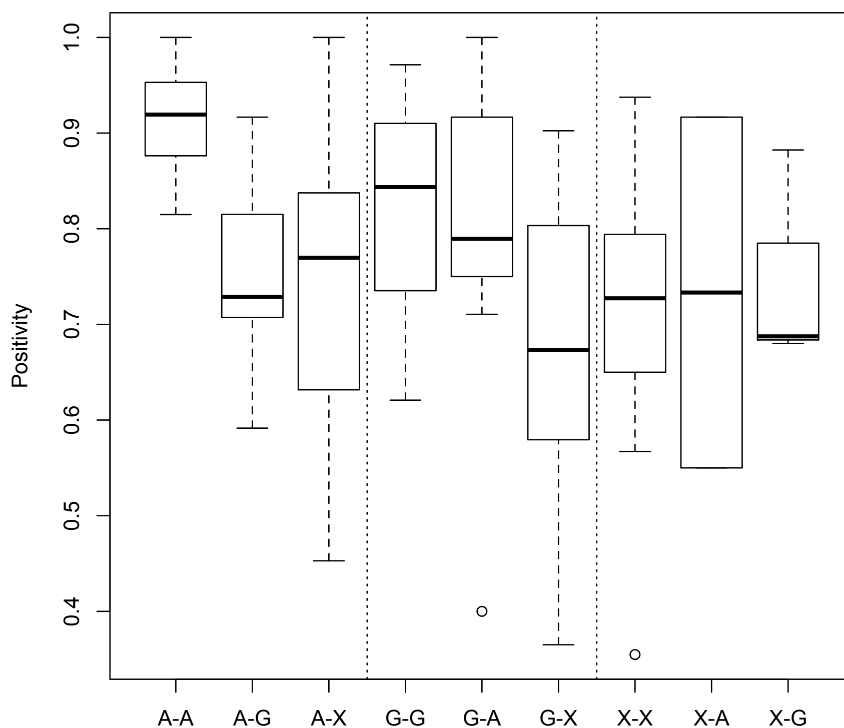

Figure 3.7: Positivity values by organization: A=Apple, G=Google, X=Rest.

that Apple remains a key gatekeeper of what lands in the source code. However, we can see that Google has become a more active contributor on the project, yet has not surpassed the number of Apple reviewers.

To find out whether reviewers have an impact on review delay and outcome, for each reviewer we calculated the number of previously reviewed patches and then discretized them according to their reviewing efforts using quartiles. Applying statistical tests, we determined that the difference for response time for A and B groups of reviewers (i.e., the less active ones) is statistically significant when compared to C or D groups (i.e., the more active ones). Since the distribution of delays is very skewed, we report both the median and mean values for reviewers' timeliness (see Table 3.5). The results show that the choice of reviewers plays an important role on reviewing time. More active reviewers provide faster responses (with median being 57 minutes and mean being 496 minutes) compared to the individuals who performed fewer code reviews (the median for time is 84 minutes and 621 minutes for the mean).

Considering that reviewers' work loads appear to affect their response rate, WebKit contributors may wish to ask the most active reviewers to assess their patches in order to get a quick response. With respect to the question whether there are reviewers who are inclined to be more positive than negative, we found that there is no correlation between the amount of reviewed patches on the reviewer positivity: 0.83 for group A, 0.84 for group B, 0.75 for group C, and 0.83 for group D. This suggests that WebKit reviewers stay true and unbiased in their role of ensuring the quality of code contributions. This observation is important since reviewers serve as gatekeepers protecting the quality of the project's code base.

Patch Writer Experience

The contributions to WebCore during the period studied came from 496 individuals among which 283 developers filing 95% of patches (submitting 5 patches or more). Considering our top five organizations, we identified that WebCore patches were submitted by 50 developers from Apple, 219 individuals from Google, 20 BlackBerry developers, 16 developers from Intel, 10 from Igalia, and 181 developers come from other organizations.

Noticing good contributor diversity in the WebCore community, we wondered if patches from certain developers have a higher chances of being accepted. To assess whether developer experience influences review timeliness and acceptance, we performed a similar procedure (as described in 3.4.1) of calculating the number of submitted changes for each developer and then discretizing patch owners according to their contributions.

We achieved similar results in the differences of response time for A and B groups of submitters (occasional contributors) is statistically significant compared to more experience developers in C or D groups. From Table 3.5 we conclude that more experienced patch writers receive faster responses (with median in group D being 48 minutes and mean being 478 minutes) compared to those who file fewer patches (the median for time in group A is 102 minutes and 682 minutes for the mean).

Investigating the impact of developer experience on positivity of the outcome, we found correlation between two variables ($\chi^2(3)$=17.93, $p < 0.01$). In particular, statistical difference was found between group A (least active developers) and groups C and D (more active developers) with the median positivity values being 1.0, 0.73 and 0.81 respectively, as well as group B (less active developers) compared to the group D (most active ones) with the median positivity being 0.63 and 0.81 respectively. These findings suggest that the WebKit community has a positive incentive for newcomers to contribute to the project as first-patch writers (i.e., group A with the median number of patches submitted being 1)

are likely to get a positive feedback. For developers of group B (where contributions range between 3–6 patches) it is more challenging to get their patches in, while contributing to the project comes with the improved experience of landing patches and as a result with more positive outcomes.

Our findings show that developer experience plays a major role during code review. This supports findings from our previous work, where we have seen faster response time for core developers compared to the casual contributors on the project [13]. This appears to show that active developers are being rewarded with both faster response and more positive review outcome for their active involvement in the project.

3.4.2 Blink

Google forked WebKit to create the Blink project in April 2013. Google decided to fork WebKit because they wanted to make larger-scale changes to WebKit to fit their own needs that did not align well with the WebKit project itself. Several of the organizations who contributed to WebKit migrated to Blink after the fork [100]. Our data demonstrates that the following companies participate on Blink by submitting patches: Samsung, Opera, Intel, Adobe, Igalia, Yandex, and BlackBerry (this list is ordered by the number of contributions).

Every Blink patch is submitted to the project's issue repository. [6] We extracted all patches by scraping the public Blink issue tracking system for the patches submitted between April 03, 2013 and January 07, 2014. The extracted dataset consists of 18,177 bugs, 37,280 patches, 721 emails and 64,610 review flags. We extracted the same information about issues, patches, and files as we did for the WebKit data (described in Section 3.3.1).

The reviewers on the Blink project approve patches by annotating it "LGTM" ("Looks Good To Me", case-insensitive) on the patch and reject patches by annotating "not LGTM". In this work, we consider WebKit's `review+`/`review-` flags and Blink's "lgtm"/"not lgtm" annotations as equivalent. Since Blink does not have an explicit review request process (e.g., `review?`), we infer requests by adding a `review?` flag a patch as soon as it is submitted to the repository. Since patches are typically committed to the version control system by an automated process, we define landed patches as those followed by the automated message from the "commit bot". The last patch on the issue is likely to be the patch that eventually lands to the Blink's source code repository. Committers could optionally perform a manual merge of the patches to the version control system, although we do not consider these due to their infrequence.

[6] `code.google.com/p/chromium/issues/`

Factor	Min	Median	Mean	Max
Patch Size	1	31	195.3	61314
Number of Components	0	1	1.871	22
Review Queue	0	0	0.738	19
Reviewer Activity	1	185	416.7	2393
Patch Writer Experience	1	107	175.4	1333

Table 3.8: Overview of the numerical factors in Blink.

To determine developer's organizational affiliations, we first inferred affiliations from the domain name of their email addresses. Then, we asked a Blink developer to confirm our guesses. To be consistent with our WebKit dataset, we marked independent developers contributing to the Blink project as "unknown".

Data Filtering. To clean up our our dataset, we performed pre-processing steps on the raw data similar to that of the WebKit study:

1. We considered only patches that affect files within `Source/core` portion (the Blink team refactored `WebCore` to this directory) of the source repository reducing the total number of patches from 37,280 to 23,723.
2. We further eliminated those patches that had not been reviewed, narrowing the input to 9,646 patches.
3. We removed 5% of slowest patches, eliminating those that took more than 80 hours to review.
4. We eliminated the least active reviewers: those who performed less than 10 code reviews; this resulted in retaining a set of 174 reviewers out of the 223 individual reviewers performing code reviews for Blink.

After applying data filtering, the final Blink dataset consisted of 8,731 patches.Table 3.8 contains the overview of the numerical factors.

Patch Size

To investigate the correlation between patch size and review response time we again split the patches according to their size into four equal groups (quartiles): A, B, C, and D.

Group A refers to the smallest patches (0–25%) with the average size of 4 lines, group B denotes small-to-medium size changes (25–50%) on average having 18 lines of code, group C consists of the medium-to-large changes (50–75%) with the mean of 63 LOC, and group D represents largest patches (75–100%) with an average size of 698 lines of code. A Kruskal-Wallis test revealed a significant effect of the patch size group on acceptance time ($\chi^2(3)$=44.16, $p < 0.01$). Acceptance time for group A (the median time is 47 minutes, the mean is 368 minutes) is statistically different compared to the time for group C (with the median of 76 minutes and the mean of 444 minutes), and D (the median of 69 minutes and the mean of 388 minutes).

The median positivity values for groups A–D are all 0.99. Reviewers' positivity remains quite high and does not appear to be affected by the size of the contributions.

Investigating the relationship between patch size and the number of patch revisions, we considered all the bug IDs that we have the patches for after applying our data filters. We calculated the number of patch revisions for each bug, as well as the size of the largest patch. Our results demonstrate statistically significant, strong impact of patch size on the rounds of revisions ($\chi^2(3)$=1473.7, $p < 0.01$. The median of the patch revisions for smaller patches of group A (under 22 LOC) is 1, while the highest number of resubmissions is 7. Group B (patch size ranges between 22–71 LOC) and group C (with the size between 72–205 LOC) has the same median value of resubmissions (on average 2 patches per issue), the highest number of patch revisions is 11 for group B and 23 for group C. The largest patches (on average of around 1,000 LOC) have more revisions than smaller patches with a median of 3 and a maximum of 30 resubmissions from group D.

Component

Blink's source code is organized similar to the WebKit's except that `bindings` and `bridge` have been removed.

We selected the same number of top actively developed components: `dom` (5,856 patches), `rendering` (5,732 patches), `page` (4,936 patches), `html` (4,934 patches), `css` (4,517 patches), `inspector` (2,938 patches), `loader` (2,305 patches). The difference in the response time between components was statistically significant ($\chi^2(6)$=40.75, $p < 0.01$); similar to the WebKit study, the `rendering` component takes longer to review (the median time is 89 minutes) compared to any other component including `inspector` (49 minutes), `page` (70 minutes), `dom` and `html` (58 minutes), and `css` (63 minutes).

We found no relationship between positivity and the component factor; the average positivity values for the components are quite high (0.98-0.99), suggesting that patches have high chance of being landed to these actively developed components.

Review Queue Length

Similar to the WebKit study, we calculated queue sizes for the reviewers at any given time (the process is described in Section 3.4.1). The resulting queues ranged from 0 to 14 patches and the average queue was 0.7 patches. Statistical tests showed that there is no significant effect of review queue size on neither reviewing time ($\chi^2(14)$=21.63, $p = 0.086$) nor positivity of the reviewers ($\chi^2(14)$=20.20, $p = 0.124$).

Thus, we found that the length of the review queue affects neither the review response time nor its outcome.

Organization

While Google developers submit 79.3% of all patches, other organizations also contribute to the project including Samsung (8.7%), Opera (3.8%), Intel (2.9%), Adobe (1.9%) and Igalia (0.2%), as well as independent individuals (3.1%). To assess whether organization influences review response and outcomes, we grouped non-Google contributions together and labelled them as "other" and then compared this group against Google-only contributions.

We discovered that there is a statistically significant relationship between response time and which organization submits the patch. Regardless of the review outcome, patches from Google developers receive faster responses than patches coming from other organizations (57 minutes vs. 95 minutes). Table 3.6 reports the mean and medians of both accepted and rejected review times for each group. Google patches are accepted or rejected faster that patches from others.

In terms of the positivity, we found no difference in review outcomes for Google vs. "other" patches. This finding is somewhat expected since 98% of reviews ares done by Google reviewers who appear to provide mainly positive reviews (see Section 3.4.2).

Comparing review response times, we noticed that the median values of both acceptance and rejection increase while the mean values decrease for Google reviewers participating on the Blink project vs. the WebKit project. While we do not have insights on why this happened, we speculate that Google focuses on rather accepting good contributions (the positivity values being very high) and providing constructive feedback to patch writers than just hurling quick negative feedbacks to the developers.

Reviewer Activity

Blink has 174 active reviewers performing code reviews. While the majority of the contributions to the Blink repository are reviewed by Google (98%), other organizations perform the remaining 2% of code reviews; Intel reviewed 75 patches, Samsung reviewed 41 patches, Adobe reviewed 13 patches, and independent developers reviewed 29 patches.

To determine whether reviewer activity as a factor has an effect on the response time, we calculated the number of previously reviewed patches for each reviewer and discretized reviewers according to their reviewing experience using quartiles (similar to the procedure we performed for WebKit). Statistical test showed that the difference for response time of less experienced reviewers (i.e., A and B groups of reviewers with the median response time of 71 minutes) is statistically significant ($\chi^2(3)$=62.14, $p < 0.01$) compared to more active ones (group C with median value of the reviewing time being 42 minutes). The difference in the response time for group C was also statistically significant compared to group D (median time is 91 minutes). We note that group D consists of one most active reviewer on the Blink project who reviewed 1,392 patches (15% of all reviewed patches). Table 3.7 reports both the median and mean values for reviewers' timeliness.

Looking at the reviewers' involvement on the project and whether it affects their positivity, we found no correlation between reviewers' positivity and their activity on the project. Positivity values remain similar for the group A, B, C and D with medians ranging between 0.99–1.0. This shows that reviewers provide positive feedback to almost all patches they review; it seems that Google reviewers use positive reinforcement when assessing contributions. If a patch is not quite ready to land to the source code, reviewers would discuss potential flaws and expect the patch to be resubmitted again for further review. Such behaviour is likely to increase response from developers submitting their patches.

Patch Writer Experience

The contributions to Blink's core during the studied period came from 394 developers. While Blink is developed and maintained mainly by Google — they submit 78% of all patches — other organizations also contribute patches to the Blink repository, including Samsung (757 patches), Opera (328 patches), Intel (256 patches), Adobe (170 patches), and Igalia (21 patches) and other independent developers (274).

To understand whether the experience of a patch writer affects the timeliness and outcome, we grouped developers according to their contributions (similar to the step described in 3.4.1). We found that the differences of response time for group A of submitters is statistically significant ($\chi^2(3)$=109.04, $p < 0.01$) when compared to more experienced developers

(B, C and D groups). From the Table 3.7 we conclude that more experienced patch writers are more likely to get faster responses (the median for groups B, C, and D being 51, 59 and 56 minutes respectively) than those who have not gained enough experience in filing project contributions, individuals who submitted fewer than 30 patches (the median for group A of submitters is 106 minutes).

When investigating the impact of developer experience on the likelihood of patch acceptance, we found correlation between two variables ($\chi^2(3)$=32.65, $p < 0.01$). In particular, a statistical difference was found between group A (least active contributors) and groups C and D (more active developers), as well as group B compared to the group D (most active ones). However, the median and mean values for the groups are almost same, 1.0 for the median and mean values ranges between 0.98-0.99. The statistical difference accounts for the distribution of the positivity scores within each group, showing that the developers who are more actively involved on the project almost certainly can expect their patches to be accepted. On the other hand, the least active developers receive a fair amount of rejections. This conclusion also supports the overall code review culture that we have seen from the lifecycle model (shown in Figure 3.3) — Google reviewers are inclined to accept patches with only very small portion (0.3%) of the submitted patches receiving negative reviews; patches that need reworking are simply resubmitted again, after reviewers provide their comments about the potential flaws.

Our findings show that developer experience is a key factor when it comes to review outcomes and timeliness. Similar to the WebKit study, we see that more active developers on the Blink project receive faster responses and their patches have high chances of being approved.

3.5 Discussion

In this section, we discuss the results from two empirical studies. We start with highlighting similarities and differences between the WebKit and Blink findings and provide our answers to the research questions. Further, we offer other interpretations of the results and discuss threats to validity.

3.5.1 WebKit and Blink Comparison

When studying each factor individually, we found that almost all of the studied factors have a statistically significant effect on the review time across both projects: `review queue`

showed no effect on time for Blink patches, and `priority` was not studied for Blink patches because they do not have priority levels.

In terms of review positivity, we detected more differences between two projects. Only `patch writer experience` has a statistically significant effect on positivity in both WebKit and Blink. Another two factors, `organization` and `review queue`, have a statistically significant effect on positivity only in WebKit. The last two factors, `component` and `reviewer activity`, showed no statistically significant effect on positivity in both projects.

We now present our answers to the research questions stated at the beginning of our work.

RQ1: What factors can influence how long it takes for a patch to be reviewed?

> *Based on the results of two empirical studies, we found that both technical (**`patch size`** and **`component`**), as well as non-technical (**`organization`**, **`patch writer experience, and reviewer activity`**) factors affect review timeliness when studying the effect of individual variables. While **`priority`** appears to influence review time for WebKit, we were not able to confirm this for Blink.*

RQ2: What factors influence the outcome of the review process?

> *Our findings from both studies suggest that **`patch writer experience`** affects code review outcome. For the WebKit project, factors like **`priority`**, **`organization`**, and **`review queue`** also have an effect on the patch acceptance.*

3.5.2 Other Interpretations

Drawing general conclusions from empirical studies in software engineering carries risk: any software development process depends on a potentially large number of relevant contextual variables, which are often non-obvious to outsiders. While our results show that certain non-technical factors have a statistically significant effect on the review time and outcome of patch submissions, understanding and measuring the practical significance of the results remains challenging. Processes and developer behaviour around their contributions to the WebKit project depend on the organization, its culture, internal structure, settings, internal development cycles, time pressures, etc. According to Beller et al. [17], the type of

a change (maintainability vs. functionality) might also account for the variations in time and outcome of code review.

Any of these "hidden" factors could potentially influence patch review delays and outcomes; for example, let us consider time pressures. It is our understanding that Apple prefers strict deadlines for shipping hardware, and the supporting software needs to match the projected delivery dates of the new hardware. This results in Apple developers prioritizing internal development goals over external ones, and thus prioritizing patches that help them meet their short-term objectives.

Organizational and geographical distribution of the developers may also provide insights into review delays. We understand that WebKit developers at Apple are co-located within the same building, which may account for a better visibility of the patches that their co-workers are working on; conversely, WebKit developers at Google tend to be more geographically distributed, which may result in a poorer awareness of the work of others.

In summary, understanding the reasons behind observable developer behaviour requires an understanding of the contexts, processes, and the organizational and individual factors that can influence code review and its outcome. Thus, while our results may be statistically valid, care must be taken in interpreting their meaning with respect to actual developer behaviour and intent. We consider that much work remains to be done in studying how best to interpret empirical software engineering research within the context of these "hidden" contextual factors.

3.5.3 Threats to Validity

Internal validity concerns with the rigour of the study design. In our study, the threats are related to the data extraction process, the selection of the factors that influence code review, and the validity of the results. While we have provided details on the data extraction, data filtering, and any heuristics used in the study, we also validated our findings with the WebKit developers and reviewers. We contacted individuals from Google, Apple, BlackBerry, and Intel and received insights into their internal processes (as discussed in Section 3.5.2). To ensure that we are on the correct track in interpreting the results of our studies, we talked to the WebKit and Blink developers via email (for Apple, Intel, Google), as well as had face-to-face meetings with Google and Blackberry developers at the respective local offices in Waterloo, Ontario (Canada). Face-to-face meetings included a presentation of the main findings followed by the discussion about possible explanations and insights into the "hidden" factors affecting code review process and practice.

When investigating the relation between patch size and the number of patch revisions, we assumed that patches are independent; this might have introduced some bias since several different patches can often be associated with the same bug ID and "mentally" form one large patch. However, for both studies we considered the size of the largest patch due to the lack of indication of which patches are actually comprising a larger patch and which patches are being resubmits.

Unlike Bugzilla's issue tracking — which is used by both Mozilla and WebKit to carry out code review tasks — Blink's code review system does not support history tracking of patch changes and lacks any explicit review requests. We overcome these limitations by inferring the review start times of Blink patches by considering the most recent patch (in terms of time) in a list of the patches followed by a review flag. This heuristic is a "best effort" approximation; unfortunately, accurate timestamps of the review starting point cannot be determined by scraping the data from the existing code review system.

Our empirical study is also subject to *external validity* concerns; we cannot generalize our findings to say that both organizational and personal factors affect code review in all open source projects. While we compared WebKit's code review process with that of Mozilla Firefox, and found that its patch lifecycle is similar to open source projects, the fact that WebKit is being developed by competing organizations makes it an interesting case yet a rather obvious exception. Hence, more studies on similar projects are needed.

Statistical conclusion validity refers to the ability to make an accurate assessment of whether independent and dependent variables are related and about the strength of that relationship. To determine whether relationships between variables are statistically significant, we performed null hypothesis testing. We also applied appropriate statistical tests (analysis of variance, post-hoc testing, and Spearman's correlation).

3.6 Summary

In this chapter we presented two empirical studies of WebKit and Blink software projects that aimed to analyze the effect of various factors, both technical and non-technical in nature, on the review time and chances of a patch to be accepted. Both studied projects are comprised of complex communities to which a variety of organizations contribute; these organizations compete at a business level while collaborating at a technical level. Ideally, the contributions of these organizations to be treated equally, based on their technical merits alone. While some influencing factors include the size of the patch itself or the part of the code base being modified, other non-technical factors have significant impact on

the code review process. Our results provide empirical evidence that organizational and personal factors influence both review timeliness as well as the likelihood of a patch being accepted. Additionally, we found significant differences in how long a patch took to be reviewed based on the organizations that wrote and reviewed a given patch along with the final outcome of the review.

Ultimately, the most influential factors of the code review process on both review time and patch acceptance are the organization a patch writer is affiliated with and their level of participation within the project. The more active role a developer decides to play, the faster and more likely their contributions will make it to the code base.

Chapter 4

Bugginess as a Measure of Code Review Quality

In Chapter 3, we studied the code review process from the developer's perspective. We analyzed the effects of various factors on the time it takes to a review a contribution as well as on the likelihood of that contribution being accepted. We now consider code review from the perspective of the reviewers themselves as well as the project owners; in particular, we pay close attention to how reviewing helps to ensure quality of the committed code. In principle, code review should improve the quality of code changes (patches) before they are committed to the project's master repository. In practice, bugs are sometimes unwittingly introduced during this process. In this chapter, we report on an empirical study investigating code review quality for Mozilla, a large open-source project. We explore the relationships between the reviewers' code inspections and a set of factors, both personal and social in nature, that might affect the quality of such inspections.

Chapter Organization. We start by providing motivation and the research questions this chapter is focused on in Section 4.1. Then we present some background about the Mozilla code review process in Section 4.2. Section 4.3 describes the methodology followed in this study. In Section 4.4, we present and discuss the results of our three research questions. In Section 4.5, we address the threats to validity. Finally, Section 4.6 summarizes our results.

Related publication. The work described in this chapter has been published in the following paper:

- Oleksii Kononenko, Olga Baysal, Latifa Guerrouj, Yaxin Cao, and Michael W. Godfrey. Investigating code review quality: Do people and participation matter? In

Proceedings of the International Conference on Software Maintenance and Evolution (ICSME), pages 111–120. IEEE, 2015.

4.1 Motivation and Research Questions

Code review is an essential element of any mature software development project; it aims at evaluating code contributions submitted by developers. Code review is considered to be one of the most effective QA practices for software projects; it is relatively expensive in terms of time and effort, but provides good value in identifying defects in code changes before they are committed into the project's code base [39]. Reviewers are the gatekeepers of a project's master repository; they must carefully validate the design and implementation of patches to ensure they meet the expected quality standards.

In principle, the code review process should improve the quality of code changes (patches) before they are committed to the project's master repository. However, in practice, the execution of this process can still allow bugs to enter into the codebase unnoticed. This work aims to investigate the quality of code review.

In this chapter, we have studied code review of a large open source system: the Mozilla project. For Mozilla, code review is a vital part of their development process, since contributions may come not only from core Mozilla developers but also from the greater user community. The Mozilla community embraces code review to help maintain consistent design and implementation practices among the many casual contributors and across the various modules that comprises the Mozilla codebase [92]. They have found that code review increases code quality, promotes best practices, and reduces regressions [93].

The Mozilla code review process requires that every submitted patch be evaluated by at least one reviewer [94]. Reviewers are advised to grant approval to a patch if 1) they believe that the patch does no harm, and 2) the patch has test coverage appropriate to the change. If a reviewer feels unable to provide a timely, expert review on a certain patch, they can re-direct the patch to other reviewers who may be better able to perform the task. However, even given the careful scrutiny that patches undergo, software defects are still found after the changes have been reviewed and committed to the version control repository. These post-release defects naturally raise red flags about the quality of code reviews. Poor-quality reviews that permit bugs to sneak in unnoticed can introduce stability, reliability, and security problems, affecting the user's experience and ultimately their satisfaction with the product.

Previous research on code review has examined topics such as the relation between code coverage/participation and software/design quality [80, 89]; however, the topic of

code review quality remains largely unexplored. In this chapter, we perform an empirical case study of a large open source Mozilla project including its top three largest modules: *Core*, *Firefox*, and *Firefox for Android*. We apply the SZZ algorithm [115] to detect bug-inducing changes that are then linked to the code review data extracted from the issue tracking system. We address the following overarching research questions:

RQ1 *Do code reviewers miss many bugs?*
The goal of code review is to identify problems (e.g., the code-level problems) in the proposed code changes. Yet, software systems remain bug-prone.

RQ2 *Do personal factors affect the quality of code reviews?*
Previous studies found that code ownership has a strong relationship with both pre- and post-release defect-proneness [20, 79, 101].

RQ3 *Does participation in code review influence its quality?*
A recent study demonstrated that low review participation has a negative impact on software quality [80].

4.2 The Mozilla Code Review Process

Mozilla employs a two-tiered code review process for assessing submitted patches: *review* and *super review* [92]. A *review* is performed by the owner of the module (or peers of the module) in question; reviewers thus have domain expertise in the specific problem area of concern. *Super reviews* are required if the patch involves integration or modifies core Mozilla infrastructure (e.g., major architectural refactoring, changes to API, or changes that affect how code modules interact). Currently, there are 30 super-reviewers for all Mozilla modules [93], 162 reviewers for the *Core* module, 25 reviewers for *Firefox*, and 11 reviewers for *Firefox for Android* (also called "Fennec") [95]. However, any person who is not on the list of designated reviewers but has level three commit access — i.e., core product access to the Mercurial version control system — can review a patch.

The Mozilla team reviews every patch using the Bugzilla issue-tracking system, which records and stores all information related to code review tasks. The process works as follows. First, a developer submits a patch to Bugzilla that contains their proposed code changes; they then request a review from a designated reviewer of the module where the code will be checked in. Patches that have significant impact on the design of Mozilla may trigger a super review. After the code has been reviewed, the reviewer will indicate

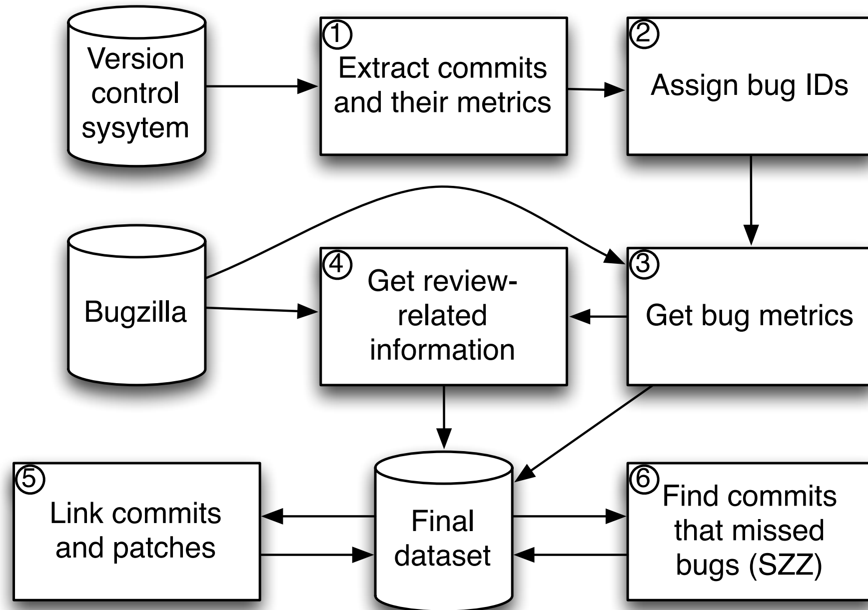

Figure 4.1: Process overview.

a positive or negative outcome, and may provide feedback comments. Once the reviewers approve a patch, the code changes are committed to Mozilla's source code repository.

As we can see, Mozilla employs a strict review policy. Investigating what makes developers miss bugs in code during review process is the topic of our work.

4.3 Methodology

To address our research questions we followed a data mining process shown in Figure 4.1 that consists of the following stages. First, we extracted commits from the Mozilla's version control repository (step 1). We then linked these commits to the corresponding bug reports in the Bugzilla issue tracking system (step 2). After that, we extracted information about linked bug reports and review-related information for patches attached to them (steps 3 and 4). Finally, we established the links between commits and reviewed patches (step 5) and identified bug-inducing commits (step 6).

Table 4.1: Overview of the studied systems.

System	Commits	Reviews	Writers	Reviewers
Mozilla-all	27,270	28,127	784	469
Core	18,203	18,759	544	362
Firefox	2,601	2,668	214	110
Firefox for Android	2,106	2,160	108	72

4.3.1 Studied Systems

Mozilla uses Mercurial as their version control system and maintains several repositories, with each repository built around a specific purpose and/or set of products. We considered `mozilla-central`[1] as the main repository; it contains the master source code for Firefox and Gecko, Mozilla's layout engine.

For our study, we took all code changes that were committed to `mozilla-central` between January 1, 2013 and January 1, 2014. In this chapter, we use terms "code change" and "commit" interchangeably. We studied four systems: *Mozilla-all* (full set of commits), as well as the three largest modules: *Core*, *Firefox*, and *Firefox for Android*. Table 4.1 describes the main characteristics of these systems; the numbers represent "clean" datasets that we obtained after performing the steps described in Sections 4.3.2, 4.3.3, and 4.3.4. We report the number of commits, reviews, writers, and reviewers for our Mozilla-all, Core, Firefox, and Firefox for Android datasets.

4.3.2 Data extraction

We extracted a total of 44,595 commits from `mozilla-central`. During the extraction phase, we collected a variety of information about each commit including its unique identifier, the name and email address of the person who made this commit, the date it was added to the repository, the textual description of a change, and the size statistics of the commit. To calculate the size statistics, we analyzed the `diff` statements of each commit. We looked only at textual `diffs`, and we excluded those that describe a change in binary files such as images. We recorded the number of files changed, the total number of lines that were added and removed, and the total number of code chunks found in the investigated `diffs`.

[1] `http://hg.mozilla.org/mozilla-central`

Linking revisions to bugs. Prior to identifying bug-inducing changes, we had to detect changes that aim to fix bugs. For that, we linked commits in the version control repository to bugs in the issue tracking system using the commit descriptions. Manual inspection of commit summaries confirmed that developers consistently include a bug ID in the commit summary, and also tend to use the same formatting. Based on this finding, we wrote a set of case-insensitive regular expressions to extract bug ID values. If a regular expression found a match, we checked whether a commit description contains any review flags to eliminate matches from unreviewed commits. If such flags were found and commits contained bug ID numbers, we linked bug ID numbers to them.

As a result of this, we were able to assign bug ID values to 35,668 (80%) commits. As suggested by Kim et al. [66], we manually checked summaries of both matched and non-matched commits and found no incorrectly assigned bug IDs. The analysis of non-matched commits (8,927 in total) showed that 2,825 commits (6.3%) were *backed out* commits, 5,520 (12.3%) commits were *merges*, 413 (1%) of them were *"no bug"* commits, and 169 of them were *other* commits.

Getting additional data from Bugzilla. We scraped Bugzilla for each linked bug ID to get detailed information, including the date when the bug was submitted, the name and email address of the person who reported the bug, the bug severity and priority, the module affected by the bug, and the list of proposed patches. For each patch, we recorded the author of the patch, the submission date, and the review-related flags. For each review-related flag, we extracted the date and time it was added, as well as the email address of the person who performed the flag update.

Out of 22,015 unique bug IDs assigned to the commits, we were unable to extract the data for 188 bugs that required special permissions to access them. For 490 bugs, we found no patches with review-related flags. Such a situation might arise in only two cases: if we incorrectly assigned bug ID in the first place, or if a patch landed into the code base without undergoing the formal code review process. To investigate this, we performed a manual check of several randomly-selected bug IDs. We found no examples of incorrect assignment: all of the checked bug IDs were bugs with no reviewed patches in Bugzilla. Since commits having no information about reviews can not contribute to our study, we disregarded them, reducing the number of unique bug IDs by 678 and the number of commits in the dataset to 34,654.

4.3.3 Linking Patches and Commits

Since each commit in the version control system is typically associated with a single patch, we linked each commit to its corresponding patch and its review-related information. However, establishing these links requires effort. The best matching of a patch to a commit can be achieved by comparing the content of the commit to the contents of each patch, and then verifying if the two are the same. However, this approach does not work in an environment where developers constantly make commits to the repository independently from one another. For example, suppose that a patch p_1 was added to Bugzilla at time t_1 and was committed to the repository at time t_2. If there were no changes to the files affected by the patch between t_1 and t_2, then the commit and the patch would be the same. If another patch p_2 changing some of those files was committed to the repository during that time frame, then the content of the commit of p_1 might not match the content of the patch p_1 itself. This might happen if (a) the line numbers of the changed code in p_1 were different at t_1 and t_2, e.g., p_2 added a line at the beginning of a file shifting all other content down, or (b) p_1 changed lines that had been changed by p_2, i.e., the removed lines in the `diff` statements of p_1 would be different from the removed lines in the `diff` statements of the commit of p_1. The most precise way of matching patches and commits would be to employ some code cloning techniques to detect matches on the string level; however, we decided against this approach as it is expensive in terms of time and effort.

Instead, we opted for a less precise but conservative way of mapping commits to patches. For each commit with a bug ID attached, we took all reviewed patches ordering them by their submission date. We then searched for the newest patch such that (1) the last review flag on that patch was `review+` or `super-review+`, and (2) this review was granted before the changes were committed to the version control system. Previous research showed that patches can be rejected after they receive a positive review [13, 14]. The first heuristic makes sense as patches with last review flags being `review-` are unlikely to land into the code. On the contrary, patches that were first rejected and later accepted (e.g., another more experienced reviewer reverted a previous negative review decision) are likely to be incorporated into the code base. The second heuristic ensures that changes cannot be committed without being reviewed first; it facilitates proper mapping when several commits in the version control system are linked to the same bug, and there are multiple patches on that bug. For example, a bug can be fixed, reopened, and fixed again. In this case, we would have two different patches linked to two commits; without the second heuristic, the same patch would be linked to both commits.

By applying these heuristics, we were able to successfully link 28,888 out of a total of 34,654 (i.e., 83%) commits to appropriate patches. The manual inspection of the remaining

17% of the commits revealed that the main reason why we did not find corresponding patches in Bugzilla was incorrect date and time values of the commits when they were added to the version control system. For example, a commit with ID `147321:81cee5ae7973` was "added" to the repository on 2013-01-28; the bug ID value assigned to this commit is 904617. Checking this bug history in Bugzilla revealed that the bug was reported on 2013-08-13, almost *7 months after* it was fixed.

4.3.4 Data Pre-Processing

Prior to data analysis, we tried to minimize noise in our data. To eliminate outliers, we performed data cleanup by applying three filters:

1. We removed the largest 5% of commits to account for changes that are not related to bug fixes but rather to global code refactoring or code imports (e.g., libraries). Some of the commits are obvious outliers in terms of size. For example, the largest commit ("Bug 724531 - Import ICU library into Mozilla tree") is about 1.1 million lines of code, while the median value for change size is only 34 lines of code. This procedure removed all commits that were larger than 650 lines (1,403 commits in total).

2. Some changes to binary files underwent code review. However, since the SZZ algorithm can not be applied to such changes, we removed the commits containing only binary `diffs` (52 commits in total).

3. We found that for some changes the submission date of their associated patches was before the start of our studied period. We believe that these patches "fell on the floor" but later were found and reviewed. To eliminate these outliers, we removed all commits representing patches that were submitted before 2012-09-01. This filter excluded 163 commits.

Our final dataset[2] contains 27,270 unique commits, which corresponds to 28,127 reviews (some linked patches received multiple positive reviews, thus, commits can have more than one review).

[2] `http://swag.cs.uwaterloo.ca/~okononen/bugzilla_public_db.zip`

Table 4.2: A taxonomy of considered technical, personal, and participation metrics used.

Type	Metric	Description	Rationale
Technical	Size (LOC)	The total number of physical lines of code that were added or removed.	Large commits are more likely to be bug prone [124]; thus the intuition is it is easier for reviewers to miss problems in large code changes.
	Chunks	The total number of isolated places (as defined by `diff`) inside the file(s) where the changes were made.	We hypothesize that reviewers are more likely to miss bugs if the change is divided into multiple isolated places in a file.
	Number of files	The number of modified files.	Similarly, reviews are more likely to be bug prone if the changes are spread across multiple files.
	Module	Mozilla module name (e.g., Firefox).	Reviews of changes within certain modules are more likely to be bug prone.
	Priority	Perceived urgency of the bug.	Our intuition is that patches with higher priority are more likely to be rushed in and thus be more bug prone than patches with lower priority levels.
	Severity	Perceived severity of the bug (i.e., how much it may affect the system)	We think that changes with higher levels of severity introduce fewer bugs because they are often reviewed by more experienced developers or by multiple reviewers.
	Super review	Indicator of whether the change required super review or not	Super review is required when changes affect core infrastructure of the code and, thus, are more likely to be bug prone.
	Number of previous patches	The number of patches submitted before the current one on a bug.	Developers can collaborate on resolving bugs by submitting improved versions of previously rejected patches.
	Number of writer's previous patches	The number of previous patches submitted by the current patch owner on a bug.	A developer can continue working on a bug resolution and submit several versions of the patch, or so called resubmits of the same patch, to address reviewers concerns.
Personal	Review queue	The number of pending review requests.	While our previous research [14] demonstrated that review loads are weakly correlated with review time and outcome; we were interested to find out whether reviewer work loads affect code review quality.
	Reviewer experience	The overall number of completed reviews.	We expect that reviewers with high overall expertise are less likely to miss a bug.

Type	Metric	Description	Rationale
Personal	Reviewer experience for module	The number of completed reviews by a developer for a module.	Reviewers with high reviewing experience in a certain module are less likely to miss defects; and on the contrary, reviewers with no past experience in performing code reviews for some modules are more likely to fail to catch bugs.
	Writer experience	The overall number of submitted patches.	Developers who contribute a lot to the project — have high expertise — are less likely to submit buggy changes.
	Writer experience for module	The number of submitted patches for a module.	Developers who make few changes to a module are more likely to submit buggy patches.
Participation	Number of developers on CC	The number of developers on the CC list at the moment of review decision.	Linus' law states that "given enough eyeballs, all bugs are shallow" [37].
	Number of comments	The number of comments on a bug.	The more discussion happens on a bug, the better the quality of the code changes [80].
	Number of commenting developers	The number of developers participating in the discussion around code changes.	The more people are involved in discussing bugs, the higher software quality [80].
	Average number of comments per developer	The ratio of the comment count over the developer count.	Does the number of comments per developer has an impact on review quality?
	Number of reviewer comments	The number of comments made by a reviewer.	Does reviewer participation in the bug discussion influence the quality of reviews?
	Number of writer comments	The number of comments made by a patch writer.	Does patch writer involvement in the bug discussion affect review quality?

4.3.5 Identifying Bug-Inducing Changes

To answer our research questions, we had to identify reviews that missed bugs, i.e., the reviews of the patches that were linked to bug-inducing commits. We applied the SZZ algorithm proposed by Śliwerski et al. [115] to identify the list of bug-inducing changes. For each commit that is a bug fix, the algorithm executes `diff` between the revision of the commit and the previous revision. In Mercurial, all revisions are identified using both a sequential numeric ID and an alphanumeric hash code. Since Mercurial is a distributed version control system, `RevisionId - 1` is not always a previous revision and thus cannot be used in the algorithm. To overcome this problem, we extracted the parent revision identifier for each revision and used it as a previous revision value for executing `diff`. The output of `diff` produces the list of lines that were added and/or removed between the two

revisions. The SZZ algorithm ignores added lines and considers removed lines as locations of bug-introducing changes.

Next, the Mercurial `annotate` command (similar to `blame` in Git) is executed for the previous revision. For each line of code, `annotate` adds the identifier of the most recent revision that modified the line in question. SZZ extracts revision identifiers for each bug-introducing line found at the previous step, and builds the list of revisions that are candidates for bug-inducing changes.

Kim et al. addressed some limitations of the SZZ algorithm as it may return imprecise results if `diff` contains changes in comments, empty lines, or formatting [67]. The problem with false positives (precision) occurs because SZZ treats those changes as bug-introducing changes even though such changes have no effect on the execution of the program. Since we implemented SZZ according to the original paper, i.e., without any additional checks, we wanted to find out how many false positives are returned by SZZ. To assess the accuracy of the SZZ algorithm, we performed a manual inspection of the returned results (that is, potential candidates returned by SZZ) for 100 randomly selected commits. We found 9% (39 out of 429 candidates) of false positives with 19 of those being changes in formatting and the remaining 20 candidates being changes in comments and/or empty lines. While we think the percentage of false positives is relatively small, the limitations of SZZ remain a threat to validity.

Finally, the SZZ algorithm eliminates those candidates that were added to the repository after the bug associated with a commit was reported to the issue tracking system. The remaining revisions are marked as bug-inducing code changes.

We ran the SZZ algorithm on every commit with a bug ID, and obtained the list of changes that led to bug fixes. Some of the changes might have been "fixed" outside of our studied period and thus would not be marked as bug-inducing. To account for such cases, we also analyzed the changes that were committed within a six-month time frame after our studied period: we assigned bug ID values, scraped Bugzilla for bug report date, and executed the SZZ algorithm; the results were added to the list of bug-inducing commits. The commits from the dataset were marked as bug-inducing if they were present in this list; otherwise, they were marked as bug-free commits.

4.3.6 Determining Explanatory Factors

Our previous work suggests that various types of metrics can affect code review time and outcome [14]. Similarly, we grouped our metrics into three types: *technical*, *personal*, and *participation*.

Table 4.2 describes the metrics used in our study and provides rationale for their selection. The metrics for technical factors were calculated on our dataset. However, the personal and participation metrics could not be extracted from our data due to its fixed time frame. For example, one developer started to participate in code review in 2013, while another one has been performing review tasks since 2010; if we compute their expertise on the data from our dataset (i.e., a 12-month period of 2013), the experience of the second developer will be incorrect, i.e., his experience for previous three years (2010–2012) will be not taken into account. To overcome this problem, we queried an Elastic Search cluster containing the complete copy of the data from Bugzilla [91]. The nature of how Elastic Search stores the data allowed us to get the "snapshots" of Bugzilla for any point in time and to accurately compute the personal and participation metrics. While computing the review queue values, we found that many developers have a noticeable number of "abandoned" review requests, i.e., the requests that were added to their loads but never completed. Such requests have no value for the `review queue` metric; therefore, any pending review request on the moment of 2014-01-01 was ignored when calculating developer review queues.

The metrics of the three types presented in this section served as explanatory variables for building our models that we describe next.

4.3.7 Model Construction and Analysis

To study the relationship between personal and participation factors and the review quality of the studied systems, we built Multiple Linear Regression (MLR) models. Multiple linear regression attempts to model the relationship between two or more explanatory variables and a response variable by fitting a linear equation to observed data [27]. The model is presented in the form of $y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + ... + \beta_n x_n$, where y is the response variable and x_1, x_2,... x_n are explanatory variables. In our MLR models, the response variable is the code review quality (buggy or not) and the explanatory variables are the metrics described in Table 4.2. The value of the response variable ranges between 0 and 1 — we used the value of 1 for bug-prone reviews and the value of 0 for bug-free inspections. While MLR is not typically used with binary dependent variables, it was demonstrated to be still applicable in such cases [57, 117]. Our goal was to explain the relationship (if any) between the explanatory variables (personal and participation metrics) and the response variable (code review quality). In our models we control for several technical dependent factors (size, number of files, etc.) that are likely to influence the review quality. We build our models similar to the ones described in [14, 24, 80, 85].

Transformation. To eliminate the impact of outliers on our models, we applied a log transformation $log(x + 1)$ to the metrics whose values are natural numbers (e.g., size, chunks, number of files, experience, review queues, etc.). Since categorical variables can not be entered directly into a regression model and be meaningfully interpreted, we transform such variables (e.g., priority, severity, etc.) using a "dummy coding" method, which is a process of creating dichotomous variables from a categorical variable. For example, if we have a categorical variable such as `priority` that has 5 levels (P1–P5), then four dichotomous variables are constructed that contain the same information as the single categorical variable. By using these dichotomous variables we were able to enter the data presented by categorical metrics directly into the regression model.

Identifying Collinearity. Collinearity, or excessive correlation among explanatory variables, can complicate or prevent the identification of an optimal set of explanatory variables for a statistical model. We identified collinearity among explanatory variables using the variance inflation factor (VIF). A VIF score for each explanatory variable is obtained using the R-squared value of the regression of that variable against all other explanatory variables. After calculating VIF scores, we removed those with high values. The VIF score threshold was set to 5 [42], thus if the model contained a variable with VIF score greater than 5, this variable was removed from the model and VIF scores for the variables were recalculated. We repeated this step until all variables in our model had VIF scores below the threshold.

Model Evaluation. We evaluated our models by reporting the Adjusted R^2 values. We also performed a stepwise selection [41], a method of adding or removing variables based solely on the t-statistics of their estimates. Since we had many explanatory variables, it was useful to fine tune our model by selecting different variables. Our goal was to identify the best subset of the explanatory variables from our full model. For the stepwise variable selection, we applied both the "forward" and "backward" methods.

4.4 Results

In this section, we present and discuss the results of our empirical study performed on various Mozilla systems.

RQ1: Do code reviewers miss many bugs?

In theory, code review should have a preventive impact on the defect-proneness of changes committed to the project's source code. Yet, code review might induce bugs since identifying the code-level problems and design flaws is not a trivial task [8]. We determine

the proportion of buggy code reviews for the different projects by computing the number of bug-inducing code reviews for each Mozilla module.

As indicated in Table 4.3, we find that overall 54% of Mozilla code reviews missed bugs in the approved commits. This value proved to be remarkably consistent across the different modules we looked at: the Core module contained 54.3% buggy reviews, Firefox contained 54.2%, and Firefox for Android contained 56%. While the studied systems are of widely varying sizes and have different numbers of commits and reviewers (as reported in Table 4.1), the proportion of "buggy" code reviews in these modules is almost identical.

Table 4.3: Number of code reviews that missed bugs.

System	# Reviews	# Buggy Reviews	% Buggy Reviews
Mozilla-all	28,127	15,188	54.0 %
Core	18,759	10,184	54.3 %
Firefox	2,668	1,447	54.2 %
Firefox for Android	2,160	1,210	56.0 %

While we were surprised to see such minute variation across the different modules, the proportion of buggy changes (54–56%) we observed is within the limits of the previously reported findings. Kim et al. [66] reported that the percentage of buggy changes can range from 10% to 74% depending on the project; with Mozilla project having 30% of buggy changes for the 2003–2004 commit history when Mozilla code base was still growing. Śliwerski, Zimmerman, and Zeller [115] found 42% of bug-inducing changes for Mozilla and 11% for Eclipse projects (the dataset contained changes and bugs before 2005).

RQ1: About 54% of the patches after being reviewed and approved, still contain problems that required bug fixes later on.

RQ2: Do personal factors affect the quality of code reviews?

Intuitively, one would expect that an experienced reviewer would be less likely to miss design or implementation problems in code; also, one would expect smaller work loads would allow reviewers to spend more time on code inspections and, thus, promote better code review quality. To investigate if these are accurate assumptions, we added technical and personal metrics from Table 4.2 to our MLR model.

Table 4.4a shows that review queue length has a statistically significant impact on whether developers catch or miss bugs during code review for all the four studied systems.

Table 4.4: Model statistics for fitting data. Values represent regression coefficients associated with each variable.

(a) Technical and personal factors.

	Mozilla	**Core**	**Firefox**	**Firefox for Android**
Adjusted R^2	0.128	0.123	0.173	0.138
Size (LOC)	0.102***	0.098***	0.108***	0.115***
Chunks	†	†	†	†
Number of files	0.058***	0.059***	0.109***	0.062*
Module	⋆	n/a	n/a	n/a
Priority	⋆	⋆	‡	·
Severity	‡	‡	·	‡
Super review	-0.139**	-0.177***	·	n/a
Review queue	0.017***	0.0204***	0.038**	0.045**
Reviewer exp.	-0.013***	-0.012***	-0.029***	-0.041***
Reviewer exp. (mod.)	†	†	‡	0.018*
Writer exp.	·	-0.004*	‡	‡
Writer exp. (module)	†	†	‡	·
# prev patches	†	†	†	-0.045***
# writer patches	-0.012***	·	·	†

(b) Technical and participation metrics.

	Mozilla	**Core**	**Firefox**	**Firefox for Android**
Adjusted R^2	0.134	0.128	0.173	0.147
Size (LOC)	0.105***	0.103***	0.105***	0.117***
Chunks	†	†	†	†
Number of files	0.060***	0.059***	0.090***	0.067***
Module	⋆	n/a	n/a	n/a
Priority	‡	⋆	‡	⋆
Severity	⋆	‡	⋆	‡
Super review	-0.124***	-0.160***	‡	n/a
# of devs on CC	0.053***	0.056***	‡	0.049*
# comments	†	†	†	†
# commenting devs	-0.124***	-0.102***	-0.075***	-0.176***
# comments/ # dev	-0.039***	-0.029**	‡	‡
# reviewer comments	0.010**	‡	0.026*	‡
# writer comments	·	·	·	-0.047**

†Disregarded during VIF analysis (VIF coefficient > 5).
‡Disregarded during stepwise selection.
⋆ "It's complicated": for categorical variables see explanation of the results in Section 4.4.
Statistical significance: '***' $p < 0.001$; '**' $p < 0.01$; '*' $p < 0.05$; '·' $p \geq 0.05$.

The regression coefficients of the review queue factor are positive, demonstrating that reviewers with longer review queues are more likely to submit poor-quality code evaluations. These results support our intuition that heavier review loads can jeopardize the quality of code review. A possible improvement would be to "spread the load on key reviewers" [11] by providing a better transparency on developer review queues to bug triagers.

For all studied systems, reviewer experience seems to be a good predictor of whether the changesets will be effectively reviewed or not. Negative regression coefficients for this metric demonstrate that less experienced developers — those who have conducted relatively fewer code review tasks — are more likely to neglect problems in changes under review. These results follow our intuition about reviewer experience being a key factor to ensure the quality of code reviews. It was surprising to us that writer experience (overall or module-based) does not appear to be an important attribute in most of the models (with the exception of Core). We expected to see that less active developers having little experience in writing patches would be more likely to submit defect-prone contributions [38, 101] and thus, increase the chances of reviewers in failing to detect all defects in poorly written patches.

Factors such as the number of previous patches on a bug and the number of patches resubmitted by a developer seem to have a positive effect on review quality for one of the four systems: Firefox for Android (and also on the overall Mozilla-all). A possible explanation is that Firefox for Android is a relatively new module, and the novelty of the Android platform may attract a variety of developers to be more involved in contributing to the Android-based browser support building on each other's work (i.e., improving previously submitted patches). However, we have not attempted to test this hypothesis.

Among the technical factors, size of the patch has a statistically significant effect on the response variable in all four models. Its regression coefficients are positive, indicating that larger patches lead to a higher likelihood of reviewers missing some bugs. Similarly, number of files has a good explanatory power in all four systems. The need for a super review policy is well explained, as super reviews have a positive impact on the review quality. This shows that such reviews are taken seriously by Mozilla-all and Core projects (our dataset contains no super reviews for Firefox for Android). It is not surprising as the role of super reviewer is given to highly experienced developers who demonstrated their expertise of being a reviewer in the past and who has a greater overall knowledge of the project's code base.

When examining the impact of `module` factor on code review effectiveness, we noticed that for some Mozilla modules such as Core, Fennec Graveyard, Firefox, Firefox for Metro, Firefox Health Report, Mozilla Services, productmozilla.org, Seamonkey, Testing,

and Toolkit, the model contains negative regression coefficients that are statistically significant; this indicates that these modules maintain a better practice of ensuring high quality of their code review process.

We found that while the bug priority level is associated with a decrease of poorly conducted reviews (P5 patches for Mozilla-all with regression coefficient being -0.13, $p <$ 0.05 and P3 patches for Core module with the regression coefficient = -0.10, $p < 0.05$), it does not have a significant impact on other two modules.

RQ2: Reviewers with higher workload are more likely to miss bugs than reviewers who have fewer patches to review. Novice reviewers (i.e., those who have done fewer reviews) are more likely to approve a buggy patch than experienced reviewers.

RQ3: Does participation in code review influence its quality?

Previous research found that the lack of participation in code review has a negative impact on the quality of software systems [80]. To investigate whether code review quality is affected by the involvement of the community, we added metrics that relate to developer participation in review process, described in Table 4.2 to our models.

Table 4.4b shows that the number of developers on the CC list has a statistically significant impact on review bugginess for three of the four systems; and its regression coefficients are positive, indicating that the larger number of developer names is associated with the decrease in review quality. This may sound counterintuitive at first. However, from the developer perspective, their names can be added to CC for different reasons: developer submitted the bug, wrote a patch for the bug, wants to be aware of the bug, commented on the bug, or voted on the bug. Thus, we think that CC is negatively associated with review quality due to its ambiguous purpose: "CC is so overloaded it doesn't tell you why you are there" [11].

The number of commenting developers has a statistically significant impact on the models of all four of the studied systems. The regression coefficients are negative, indicating that the more developers that are involved in the discussion of bugs and their resolution (that is, patches), the less likely the reviewers are to miss potential problems in the patches. A similar correlation exists between review quality and the metric representing average number of comments per developer and having statistically significant negative coefficients for two of the four systems (Mozilla-all and Core). This shows that reviews that are accompanied with a good interactions among developers discussing bug fixes and patches are less prone to bugs themselves. The number of comments made by patch owners is also demonstrated to have a statistically significant negative impact on review bug-proneness

in the model for Firefox for Android only. These results reveal that the higher rate of developer participation in patch discussions is associated with higher review quality.

While any developer can collaborate in bug resolution or participate in critical analysis of submitted patches, reviewers typically play a leading role in providing feedback on the patches. Thus, we expected to see that the number of comments made by reviewers has a positive correlation with review quality. However, in Table 4.4b we can see that while having a statistically significant impact in the models for two of the four systems, the regression coefficients are positive, indicating that more reviewers participate in discussing patches, the more likely they would miss bugs in the patches they review. A possible explanation of these surprising results is that if a reviewer posts many comments on patches, it is possible that he is very concerned with the current bug fix (its implementation, coding style, etc.). Or, as our previous qualitative study revealed, the review process can be sensitive due to its nature of dealing with people's egos [11]. As one developer mentions "there is no accountability, reviewer says things to be addressed, there is no guarantee that the person fixed the changes or saw the recommendations." Code review is a complex process involving personal and social aspects [26].

Table 4.4b demonstrated that while developer participation has an effect on review quality, technical attributes such as patch size and super review are also good predictors. All models suggest that the larger the code changes, the easier it is for reviewers to miss bugs. However, if changes require a super review, they are expected to undergo a more rigorous code inspections. For two of the three studied systems, super review has negative regression coefficients; but it does not have a significant impact for Firefox (Firefox for Android patches have no super reviews).

Code reviews in `modules` such as Core, Fennec Graveyard, Firefox, Firefox for Metro, Firefox Health Report, Mozilla Services, productmozilla.org, Seamonkey, Testing, and Toolkit are statistically less likely to be bug prone; the regression coefficients for these modules have negative values and are -0.209 ($p < 0.05$), -0.339 ($p < 0.01$), -0.191 ($p < 0.05$), -0.205 ($p < 0.05$), -0.197 ($p < 0.05$), -0.263 ($p < 0.05$), -0.679 ($p < 0.001$), -0.553 ($p < 0.01$), -0.204 ($p < 0.05$) and -0.297 ($p < 0.001$) respectively. Similar to the findings for RQ2, code inspections performed in these modules appear to be more watchful than in other components.

Priority as a predictor has a statistically significant impact on review outcome for Core and Firefox for Android only. For the Core module, priority P3 has a negative effect (regression coefficient = -0.09, $p < 0.05$), i.e., the patches with P3 level are expected to undergo more careful code inspections. For Firefox for Android, patches with priority P1 are more likely to be associated with poor reviews (regression coefficient = 0.17, $p < 0.001$).

Among severity categories, we found that patches of trivial severity have statistically significant negative impact on review bug-proneness in the models for Mozilla-all and Firefox (regression coefficients =-0.125 and =-0.385 $p < 0.05$, respectively). Developers find that "priority and severity are too vague to be useful" [11] as these fields are not well defined in the project. But since these metrics are associated with the review quality, developer should be given some estimation of the risks involved to decide how much time to spend on patch reviews.

> *RQ3: The number of developer names on bug's CC list is correlated with the decrease in review quality. The number developers involved in patch discussion and the number of comments per developer are correlated with the increase in review quality, i.e., the more developers comment on a patch and more actively they are doing it, the more likely that all defects will be identified.*

While the predictive power of our models remain low (even after rigorous tune-up efforts), the best models appear to be for Firefox (Adjusted $R^2 = 0.173$ for fitting technical and personal factors, as well as technical and participation metrics). The goal in this study is not to use MLR models for predicting defect-prone code reviews but to understand the impact our personal and participation metrics have on code review quality, while controlling for a variety of metrics that we believe are good explainers of review quality. Thus, we believe that the Adjusted R^2 scores should not become the main factor in validating the usefulness of this study.

4.5 Threats to Validity

External validity. Our findings cannot be generalized across all open source projects. While we only study one (large) open source community, we considered various Mozilla modules including Core, Firefox, and Firefox for Android. Our goal was to study a representative open source system in detail. Nevertheless, further research studies are needed to be able to provide greater insight into code review quality.

Internal validity concerns with the rigour of the study design. In this study, the heuristics used, as well as the data filtering techniques adopted may be a threat. We mitigate such a threat by providing details on the data extraction and filtering and by using a well-known outliers filtering procedure. The choice of metrics may be seen as a threat. We selected widely used metrics characterizing code review activities, bugs, code changes (patches/commits), and developer attributes.

We assume that a code review is documented and communicated via Bugzilla issue tracking system. While this assumption holds in most cases, some code review tasks can be carried out via other channels such as email, face-to-face meetings, etc. When investigating the relation between code change and reviewer, we assumed that patches are independent; this might have introduced some bias since several different patches can often be associated with the same bug ID and "mentally" form one large patch. In our study we considered that the most recent patch is the one that gets incorporated into the code.

We assume that all bugs tied to to particular patch and that they appear in the software system because they were not caught during code review of that patch. Although this assumption is likely to hold for most bugs, it is possible that (due to some external changes) some defects might be found in patches that were initially bug free.

In Bugzilla, bug reports per se actually serve several purposes: they can be bug-fix requests, or requests for adding new functionality, or documentation-related changes, etc. Since Bugzilla does not provide mechanisms for distinguishing between "true" bugs and new feature requests, we treat all changes as bug fixes.

When calculating review queue length of developers, we assume that at any given point the number of review requests for a developer defines his or her current review load. This heuristic is a "best effort" approximation; accurate review loads are hard to determine by scraping the data from the existing code review system.

Conclusion validity is the degree to which conclusions we reach about relationships in our data are reasonable. Proper regression models were built for the sake of showing the impact of studied factors on the code reviews bugginess. In particular, we built our MLRs for two types of metrics and evaluated them based on the appropriate measures such as the deviance explained and Adjusted R^2.

4.6 Conclusion

Code review explicitly addresses the quality of contributions before they are integrated into project's codebase. Due to volume of submitted contributions and the need to handle them in a timely manner, many code review processes have become more lightweight and less formal in nature. This evolution of review process increases the risks of letting bugs slip into the version control repository, as reviewers are unable to detect all of the bugs.

In this chapter, we have explored the topic of code review quality by investigating what factors might affect it. We investigated what aspects contribute to poor code review quality to help software development projects better understand their current review processes.

We built and analyzed MLR models to explain the relationships between personal characteristics of developers, team participation and involvement in code review, and technical properties of contributions on the effectiveness of code review.

Our findings suggest that developer participation in discussions surrounding bug fixes and developer-related characteristics such as their review experience and review loads are promising predictors of code review quality for all studied systems. Among technical properties of a change, its size, the number of files it affects, its impact on the rest of the project's code (or the need to perform a super review) have also a significant link with the review bug-proneness. We believe that these findings provide practitioners with strong empirical evidence for revising current code review policies and promoting better transparency of the developers' review queues and their expertise on the modules.

Chapter 5

Developers' Perception of Code Review Process

In the previous chapter, we quantitatively explored the degree to which different factors affect the reviewer's ability to catch defects in contributions under the review. A study purely based on the raw mined data cannot draw a full picture of the review process. Therefore, in this chapter, we perform a qualitative study of code review practices of a large, open source project, and we investigate how the developers themselves perceive code review quality.

Chapter Organization. This chapter starts with Section 5.1 providing an overview of the study as well as the research questions asked. Section 5.2 describes our methodology including the survey design, participants, and data analysis. Section 5.3 presents the results of the qualitative study. Section 5.4 discusses implications of our work and suggests possible future research directions. Section 5.5 addresses limitations of our work. Finally, Section 5.6 concludes the chapter.

Related publications. The work described in this chapter has been published in the following papers:

- Oleksii Kononenko and Olga Baysal. A Qualitative Exploratory Study of How OSS Developers Define Code Review Quality. Technical Report CS-2015-14, University of Waterloo, Waterloo, Canada, August 2015.
- Oleksii Kononenko, Olga Baysal, and Michael W Godfrey. Code Review Quality: How Developers See It. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, pages 1028–1038. ACM, 2016.

5.1 Study Overview

In Chapter 4, we explored the topic of code review quality by conducting a quantitative investigation of what factors may influence the quality of evaluating code contributions. The study presented there was of quantitative nature as it employed data mining and analysis of project's repositories. While we found that both technical and personal attributes are associated with the review quality, many other factors such as organization, its culture and structure, development cycles, time pressures, etc., can potentially influence how reviewers assess code changes. Since these "hidden" factors are difficult to take into account in a quantitative analysis because such data is not available, easily accessible, or extractable from the available artifacts, we decided to employ qualitative research methods to fill the gap in the knowledge we had about the developer perception and attitude towards the code review quality.

Our qualitative study is organized around an exploratory survey that we design based on the state-of-the-art qualitative research [16, 52, 113] and our own observations of the Mozilla code review process and interactions with Mozilla developers during our previous research project [12]. We conducted an exploratory survey with 88 Mozilla core developers. In this chapter, we will refer to patch writers and reviewers as developers. Our qualitative analysis of the survey data aims at addressing the following research questions:

RQ1 *How do Mozilla developers conduct code review?*

Existing literature offers several case studies of how code review processes are employed by various software development projects and organizations [8, 13, 50, 86, 104, 106].

RQ2 *What factors do developers consider to be influential to review time and decision?*

Code review is a complex process that involves people, their skills and social dynamics, as well as development artifacts and environments; thus, it can be affected by both technical [62, 86, 103, 106] and non-technical factors [14, 52, 76, 121].

RQ3 *What factors do developers use to assess code review quality?*

While the quality assessment of code contributions is an active research area, the topic of code review quality remains largely unexplored. To better understand developer perception and attitudes towards the quality of the process that evaluates code changes, we further refine this research question into the following subquestions:

RQ3.1 *How do reviewers assess the quality of a patch?*
RQ3.2 *How do developers define a well-done code review?*

RQ3.3 *What factors are perceived to contribute to the review quality?*

RQ4 *What challenges do developers face when performing review tasks?*

We believe that it is important to understand what ongoing problems developers deal with to provide them with better tools to support their daily tasks and activities.

Our main findings reveal that the review quality is primarily associated with the thoroughness of the feedback, the reviewer's familiarity with the code, and the perceived quality of the code itself. As expected, we found that different factors including technical, personal and social signals, are perceived to contribute to the review quality. Also, we found that reviewers often find it difficult to keep their technical skills up-to-date, manage personal priorities, and mitigate context switching.

The chapter makes the following contributions:

- A qualitative study with the professional OSS developers who participated in our survey on the topic of code review quality.
- A detailed survey analysis that offers insights into the developer perception of the code review quality and factors affecting it, as well as identifies of the main challenges developers face when conducting review tasks.
- A publicly available dataset of 88 anonymized survey responses[1].

5.2 Methodology

We conducted an exploratory qualitative study that involved data collection through a survey with professional developers. This section describes the survey design, the participants, and the analysis of the responses in detail.

5.2.1 Survey Design

Our survey consisted of three main parts: nine questions about the developer's demographic background and work practices, three Likert-scale questions related to different aspects of code review, and seven follow-on open-ended questions to allow developers to elaborate on issues raised by the multiple choice questions. The full text of the survey questions can be found in Appendix A. Participants were asked to spend 5–10 minutes to complete the survey.

[1] `http://swag.cs.uwaterloo.ca/~okononen/review_quality/`

The main goal of conducting the survey was to solicit developer feedback on the perceived quality of code reviews and factors affecting review time, decision, and quality. We also wished to identify key problem areas within the existing review process.

5.2.2 Participants

We decided to continue our work within the Mozilla project developer community for several reasons: much of our previous work has studied this project and we have good intuition about the system and its development practices, we have made good contacts within the project who are supportive of our research goals, and because Mozilla is a well known, very large, and long-lived open source project.

To identify potential participants for our study, we looked at the 12 month history (from May 10, 2014 to May 10, 2015) of all contributions to the Mozilla project as they are recorded in Bugzilla issue tracking system. Because of the Bugzilla's limitations on the search results, we directly queried Mozilla's Elastic Search cluster that contains the up-to-date copy of Bugzilla data [91]. By processing the queried data, we extracted 3,142 unique email addresses (Bugzilla uses an email address as a unique user identifier). After that, we queried the cluster for each email address to extract information about each developer's activity: the number of contributions they have submitted for review, and the number of patches that the developer had reviewed during the studied period. Finally, we used Bugzilla's REST API to extract developers' real names.

We decided to limit our survey to experienced developers who were not new to the project. We computed an *experience* value as the sum of submitted and reviewed patches. We set a threshold for the experience value at 15, meaning that anyone with a combined experience of at least 15 patches will pass the filter; this reduced the list of potential participants to 843 (27%) people. To filter out developers who were new to the Mozilla project — regardless of their experience level — we defined *familiarity* as having contributions (submitted and/or reviewed patches) at least 6 months prior to the beginning of the studied period; this filter further reduced the list of experienced developers to 403 (13%) people.

Once we selected developers whom we wanted to survey, we sent out 403 personalized emails. Each email contained the number of contributions submitted or reviewed during the 12 months period and an invitation to participate in the survey. The survey was open for 3 weeks (from May 29 to June 19, 2015) and we received 88 responses (22% response rate).

The beginning of the survey consisted of background-related questions. By analyzing the responses, we found that we had successfully targeted highly experienced developers: about 48% of respondents said that they have more than 10 years of software development experience, while another 26% of them have between 7 and 10 years of experience. Most of the respondents have been performing code review for more than 3 years (67%).

5.2.3 Survey Data Analysis

We applied a grounded theory methodology to analyze the survey data; as we had no predefined groups or categories, we used an open coding approach. As we analyzed the quotes, themes and categories emerged and evolved during the open coding process [83].

Researcher Kononenko created all of the "cards", splitting 88 survey responses into 938 individual quotes; these generally corresponded to individual cohesive statements. In further analysis, researchers Kononenko and Baysal acted as coders to group cards into themes, merging themes into categories. For each open-ended question, we proceeded with this analysis in three steps:

1. The two coders independently performed card sorts on the 20% of the cards extracted from the survey responses to identify initial card groups. The coders then met to compare and discuss their identified groups.

2. The two coders performed another independent round, sorting another 20% of the quotes into the groups that were agreed-upon in the previous step. We then calculated and report the coder reliability to ensure the integrity of the card sort. We selected two of the most popular reliability coefficients for nominal data: percent agreement and Cohen's Kappa. Coder reliability is a measure of agreement among multiple coders for how they apply codes to text data. To calculate agreement, we counted the number of cards for each emerged group for both coders and used ReCal2 [44] for calculations. The coders achieved a substantial degree of agreement; on average two coders agreed on the coding of the content in 96% of the time (the average percent agreement varies across the questions and is within the range of 94.2–97.2%; while the average Cohen's Kappa score is 0.68).

3. The rest of the card sort (for each open-ended question) — 60% of the quotes — was performed by both coders together.

5.3 Results

During the open coding process, 30 main categories emerged; this includes one we labelled "irrelevant". Table 5.1 presents these categories in detail reporting the number of quotes, the number of respondents, the question numbers, the totals, and the average percent agreement for each question.

5.3.1 RQ1: How do Mozilla developers conduct code review?

First, we wanted to understand the current practices of performing code review tasks at Mozilla. We asked developers several multiple choice questions with an option of providing their own detailed response. The first pair of questions focused on the workload that developers face: the average number of patches they write and the average number of reviews they perform each week. While the answers to these two questions are skewed towards smaller workloads (fewer than 5 patches per week submitted or reviewed, 69% and 57% respectively), we received many more responses for the heavier review workload than for the patch workload. About 10% of the respondents reported that they review 11 to 20 patches each week, while another 4% said that they review more than 21 patches each week. The analysis of a contingency table for these two variables shows that developers with high workloads (i.e., over 10 patches/reviews per week) tend to concentrate their efforts on a single task type, i.e., either writing patches or reviewing them. The need for "dedicated" reviewers is pursued to bring their unique knowledge and expertise, e.g., overall architecture or domain knowledge, to the project to ensure the correctness and fit of code contributions. This finding mirrors Mozilla's notion of super reviewers — a small set of developers enlisted by Mozilla who provide an additional review for certain kinds of changes [93].

The remaining two questions focused on where developers perform code review (i.e., within what environment) and where they discuss patches under review. While all review-related information is stored in Bugzilla, there is no requirement in the Mozilla's code review policies on where a review should be performed. Surprisingly, although Mozilla provides their developers with a code review platform called MozReview [125], only 5% of the respondents said that they are using it. The majority of the respondents (80%) conduct their code review tasks inside Bugzilla itself, while another 8% copy a patch locally into their IDE. As for the locations of patch discussions, developers were allowed to select multiple of the proposed answers and/or their own answer. The two overwhelmingly popular answers were Bugzilla and IRC channel (99% and 78% respectively), while VoIP,

Table 5.1: The list of categories that emerged during open coding.

Category	**Q11**		**Q13**		**Q14**		**Q15**		**Q17**		**Q18**	
	#Q	#R	#Q	#R	#Q	#R	#Q	#R	#Q	#R	#Q	#R
Code quality	49%	57%	24%	30%	31%	65%	9%	22%	8%	15%	1%	2%
Testing	13%	28%	6%	9%	12%	36%	7%	15%	8%	15%	–	–
Time constraints	1%	4%	–	–	–	–	8%	20%	14%	19%	17%	25%
Change scope/rationale	11%	22%	9%	12%	26%	58%	–	–	4%	8%	10%	15%
Understanding code change/base	–	–	6%	9%	–	–	21%	30%	20%	31%	31%	38%
Human factors	–	–	–	–	–	–	17%	28%	14%	23%	11%	16%
Tools	–	–	4%	5%	–	–	–	–	6%	12%	9%	9%
Communication	–	–	2%	2%	–	–	–	–	8%	12%	1%	2%
Change complexity	–	–	18%	23%	10%	31%	–	–	8%	15%	10%	15%
Relationship/trust	5%	9%	3%	5%	–	–	–	–	–	–	–	–
Usefulness	–	–	–	–	1%	3%	1%	3%	–	–	–	–
Workload	–	–	–	–	–	–	–	–	4%	8%	4%	5%
Submitter related	2%	6%	3%	5%	–	–	–	–	–	–	–	–
Architecture/design	–	–	–	–	5%	15%	6%	10%	–	–	–	–
Reviewer related	9%	13%	–	–	–	–	–	–	–	–	–	–
Discussion	1%	4%	–	–	–	–	–	–	–	–	–	–
Conformance to project goals	6%	7%	–	–	–	–	–	–	–	–	–	–
Bug type	–	–	9%	12%	–	–	–	–	–	–	–	–
Selecting correct reviewer	–	–	7%	12%	–	–	–	–	–	–	–	–
Performance	–	–	–	–	2%	8%	–	–	–	–	–	–
Integration into code base	–	–	–	–	4%	15%	–	–	–	–	–	–
Security	–	–	–	–	1%	3%	–	–	–	–	–	–
Memory management	–	–	–	–	1%	2%	–	–	–	–	–	–
Familiarity with the author	–	–	–	–	1%	5%	–	–	–	–	–	–
Thorough feedback	–	–	–	–	–	–	23%	38%	–	–	–	–
Catching bugs	–	–	–	–	–	–	4%	8%	–	–	–	–
Organizational factors	–	–	–	–	–	–	–	–	4%	4%	–	–
Documentation	–	–	–	–	–	–	–	–	2%	4%	–	–
Context switch	–	–	–	–	–	–	–	–	–	–	6%	10%
Irrelevant	3%	7%	9%	12%	5%	15%	4%	10%	–	–	–	–
Total:	141	54	67	43	290	86	219	86	50	26	81	55
Average percent agreement	**97.2%**		**96.6%**		**95.5%**		**94.2%**		**94.2%**		**95.0%**	

Notes: #Q: the number of quotes, #R: the number of respondents, Q11: factors affecting decision, Q13: factors affecting time, Q14: patch quality, Q15: characteristics of code review quality, Q17: other factors affecting review quality, Q18: challenges.

email, and face-to-face discussions received a similar number of responses (around 22% each). While this wide adoption of IRC might be influenced by Mozilla itself, it also might be explained by the fact that IRC allows them to have real-time, less formal discussions with ability to bring in more people into a conversation as needed.

RQ1: While most of developers write patches as well as review them, a dedicated group of developers is responsible for reviewing code changes. The majority of reviewers conduct code review in Bugzilla despite having access to a custom built code review tool, and use various communication channels for discussing code modifications.

5.3.2 RQ2: What factors do developers consider to be influential to review time and decision?

We asked developers about the factors that they believe are most likely to affect the length of time needed to review a patch, as well as the decision of the review (i.e., accept or reject). For each aspect (review time and decision), we solicit developers' opinions via a 5-point Likert-scale question and probe more in-depth information via an optional follow-on open-ended question. The proposed answers to Likert-scale questions were compiled from the factors that were previously reported in the literature [14, 62, 124] to have an impact on time and outcome. The open-ended questions provided developers an opportunity to specify any other factors not covered by the Likert-scale question.

1) Time. The analysis of the Likert-scale question (Figure 5.1) shows that size-related factors (patch size, the number of modified files, and the number of code chunks) are the ones the developers feel are most important (100%, 95%, and 95% of positive responses respectively). This finding is consistent with several previous quantitative studies that demonstrate the correlation between the size of the code change and the review time (i.e., smaller patches are more likely to receive faster responses) [14, 62, 124]. The second most positive group is experience — reviewer experience (96%) and patch writer experience (91%). Again, this also mirrors previous research that found that the increase in experience leads to faster reviews. While all other proposed factors received more than 50% of positive responses, the two factors with the biggest numbers of negative responses stand out: bug priority and severity received 36% and 30% of negative responses respectively. Such high values speak against the very idea of bug triage. It may be because Mozilla developers use the priority and severity fields inconsistently [58], or because these fields are not used as intended (for example, in our previous study, we found that over 96% of all patches in WebKit project are assigned the same priority value [14]).

The manual coding analysis of the open-ended question revealed several categories that developers believe have an impact on code review time. The biggest theme identified in the responses is *code quality*, which includes *code quality* and *change complexity* categories. As explained by R67, *"The amount of in-code comments describing what the patch does.*

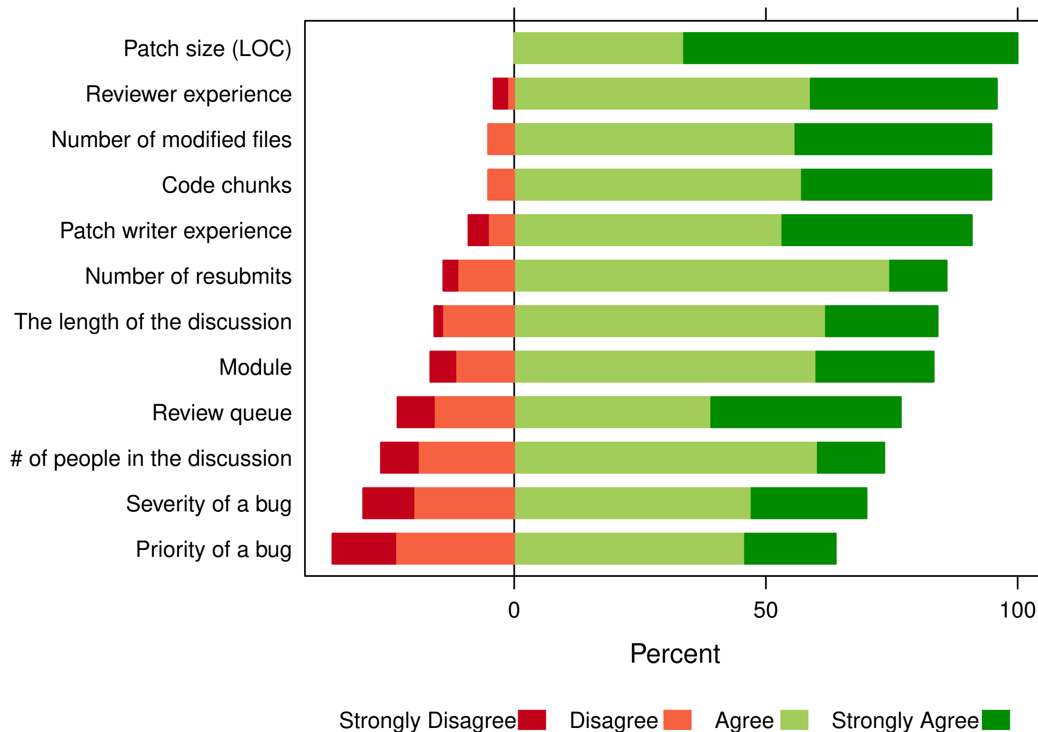

Figure 5.1: Factors influencing code review time.

Readability/variable-naming affecting how hard it is to understand any particular hunk of the patch on its own." When reviewing patches, the developers stated that *"patches dependency"* (R39) and *"changes to the API surface between modules"* (R32) affect the review time.

Perhaps surprisingly, developers identified that the *bug type* category also plays a role during the review of a patch and affects its time. According to respondent R76, *"When the cause of the bug is obscure, it takes time to review"*, while R74 said *"nature of the bug — some bugs require time-consuming manual testing"*.

Another category that emerged from the responses is *patch scope and rationale*. Here, the scope also includes granularity: *"Whether the patch is broken up into self-contained pieces or whether it's one big patch touching lots of different areas — 5 individual patches are much faster to review in total than one big merged patch of those pieces"* (R19). Developers believe that the clarity of explanation of what is being changed and why affects the review time: *"clearly identified goal for the patch"* (R11) and *"what is patch trying to do, and should we even be doing that?"* (R55). *Understanding the code base* category goes

along with the scope of a patch. Several developers stated that amount of knowledge that a reviewer has about the code being changed affects the review time.

Several of the emerged categories can be combined into a *social* theme. One of the categories here is *selecting the correct reviewer*. There are different characteristics that identify the suitability of a reviewer. For R87 it is *"the personality of a reviewer"*, while for R52 it is presence of *"personal backlog of work, and personal priorities"*. Moreover, sometimes the reviewers themselves question their suitability for reviewing a patch: *"am I the best person to be reviewing this patch?"* (R55). Developers also identified the importance of previous *relationship* with an author of a patch: *"if someone has a good track record I won't think about the code in quite as much detail compared to someone with a track record of breaking things often"* (R13). The other categories in this theme are about *submitter type* (e.g., newcomer or not) and the ease of *communication* between a patch writer and a reviewer.

2) Decision. Contrary to the answers to the Likert-scale question about the review time, we found no agreement between developers (i.e., strong prevalence of either positive or negative answers) for the majority of factors in the case of the review decision (Figure 5.2). Similarly to the previous question, both patch writer experience and reviewer experience are the factors with the most positive answers (86% and 84%). At the same time, the size-related factors (patch size, the number of modified files, and the number of code chunks) no longer have an overwhelming number of positive answers; instead, the respondents are more likely to disagree with the statement that these factors affect review decisions. Surprisingly, bug severity and priority are now the third and the fourth the most agreed factors. Another interesting finding is related to reviewer workload: about 81% of respondents disagree that workload affects the decision in any way. This demonstrates that developers think of reviewers as highly capable of carefully analyzing every patch regardless of the time pressure they might face. While such attitude describes the project's culture, this result contradicts our previous finding that suggest that reviewers with shorter review queues are more likely to reject a patch [14].

Several categories emerged during the analysis of the open-ended question related to review decision. The highest impact on the review decision is perceived to be *code quality* of a submitted patch. While developers associate different meanings with the term "code quality"; they can be grouped into several sub-categories. The first one is adherence to the code style (R57 – *"the quality of the code, and whether it adheres to accepted style and practices"*), as well as spelling (R38 – attention to *"details such as spelling, grammar and code formatting"*). Other two sub-categories are readability and simplicity of a patch (R34 – *"ease of understanding of code/changes, i.e., simplicity of code"*), and presence and quality of design or architectural changes. Finally, developers associate the code correctness and its maintainability with code quality.

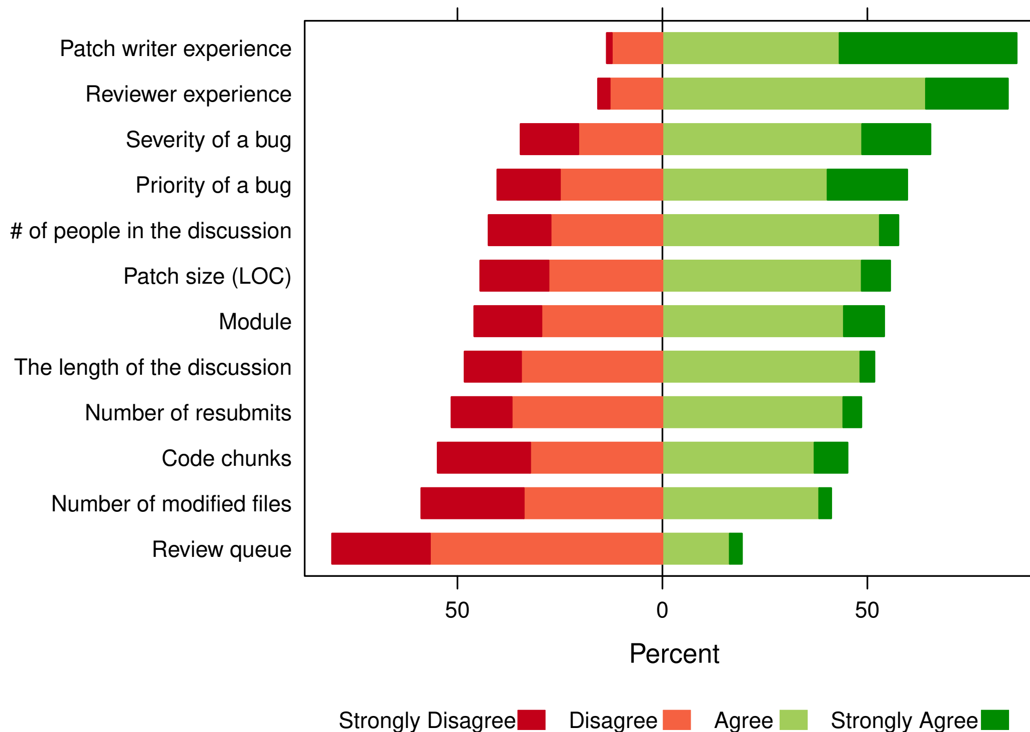

Figure 5.2: Factors influencing code review decision.

The second biggest category identified from the answers is *testing*. When developers submit a patch they can include the results of running existing tests, as well as include the tests they wrote specifically for that patch. The two sub-categories that we identified reflect the option patch writers have. The first sub-category is focused on the presence of automated tests in a patch: *"... changes that are accompanied by tests are much more likely to be accepted"* (R20). Moreover, developers identified that the actual completeness of tests is also important: *"thoroughness of tests included in patch"* (R37). The other sub-category represents the presence of test results for a patch: *"including test results as a message on the bug tracker can either give the reviewer more confidence to accept the patch (if the tests pass) or likewise lead them to reject the patch (if the tests fail)"* (R38).

Change scope and rationale is believed to be an of influential factor for reviewers making their decisions. Reviewers first look for the actual appropriateness of change to be incorporated into the code base: *"Does the feature fit in with the product (for patches submitted out of the blue)"* (R29). As R10 explains: *"... not all fixes or improvements are a good idea to actually land, even if they're correct"*. Also, reviewers expect a clear explanation

of the reasoning behind the proposed change, how it solves a problem, and why an author chose a particular way of doing it. According to R38, including such information *"can have a significant impact on some reviewers' confidence to accept the patch"*.

Similarly to the previous question, we have a theme of social categories. The reviews are done by humans, so the process is likely to be influenced by their personalities. Indeed, we identified *reviewer related* factors. Several developers report that with some reviewers it is more difficult to get a patch accepted than with others: *"a new reviewer might feel inclined to find a fault to prove that they done due duty in reviewing the patch"* (R38), and *"the perfectionist syndrome (Can you try ... ?)"* (R49). In addition to that, *"individual quirks/preferences of the reviewer"* (R20) play a role as well. *Relationship/trust* between reviewer and patch writer is found to play a critical role in decision making. Several respondents stated that interpersonal relationship is important for the review outcome. As explained by R36: *"If it's someone you trust you don't have to check things as rigorously"*. And finally, *contributor type* (i.e., whether he is new, mentored, or experienced contributor) can influence reviewers' decisions: *"if the patch writer is a new or first-time contributor, the reviewer may be inclined to encourage them by accepting their patch more readily (after identifying any obvious problems that need fixing)"* (R38).

RQ2: Developers believe that factors such as the experience of developers, the choice of a reviewer, size of a patch, its quality and rationale affect the time needed for review; while bug severity, code quality and its rationale, presence and quality of tests, and developer personality impact review decisions.

5.3.3 RQ3: What factors do developers use to assess code review quality?

Quality is one of the key attributes of ensuring high standards of both code and project development. With this research question, we explore how developers perceive the quality of a patch and what characteristics they believe to be essential in contributing to a well-done code review. To answer this question, we analyzed two mandatory and one optional open-ended questions, as well as one multiple-choice question of the survey.

1) Perception of a patch quality.

One of the top attributes for developers when evaluating patch quality is *code quality*. Code quality has many interpretations. For some developers it is associated with coding style such as *"the names of things need to be descriptive"*, readability, compactness,

maintainability (*"lack of redundant or duplicated code"*, *"strong and unverified coupling"*), *"consistent indentation and style"*, and *"elegance and lack of hacks"*. While for others code quality is about the presence of meaningful comments (*"comments should tell why not what"*), documentation and *"clear and helpful"* commit messages, *"I'm looking for a thoughtful summary that instructs me, reviewer, what is going on and what to expect"* (R28). Some developers find that *"adherence [of the code] to project module standards"* was equally important to ensure the changes are consistent and conformant to the Mozilla Coding Standards.

Change rationale is the second top property that reviewers look for. Patches are assessed for their correctness, *"does [it] actually implements what it intends to do?"* (R19), associated risk and possible alternative solutions, *"are there easier, less risky ways to achieve the same thing?"* (R35), functionality and errors (e.g., *"correct handling of exceptional cases"* (R33), *"are all cases handled?"* (R56)). Reviewers examine whether the patch author understands the source of the problem and the problem domain, without *"introducing any other bugs"* (R62) or ambiguity. Reviewers often think of their own solution to the fix before reviewing it and then compare it with the submitted patch. They also try to understand how much time the author spent on the patch and *"how well the solution has been thought through: does it needlessly reinvent the wheel, does it rewrite everything from scratch even though a spot fix would have been better, ... does it use "clever" tricks that others will struggle to understand"* (R64). In a nutshell, a high quality patch *"usually provides a robust solution for the problem"* (R42).

Change complexity is also perceived as an important property of the patch quality. Developers often look for simple solutions: *"simpler is better"* (R20), *"simplicity of code makes a big difference. Code that is complicated often is the result of not being able to distill the problem down to its core. Also, reducing the cognitive load required to understand the code means it's easier to maintain, and less likely to have bugs in it"* (R34). If a patch is trying to resolve more than one issue, it is expected that submitter split it into multiple patches: *"if the patch is addressing 3 or 4 different things it is lower quality than 3 or 4 separate patches for the individual issues"* (R13). Many developers agree that size of the change is correlated to the bug-proneness: *"small, focused changes are easier to assess than large ones. If bug rate is proportional to lines of code, quality is inversely proportional to patch size. So, small patches preferred"* (R28).

Testing is also a key indicator of quality for developers when they evaluate patches. Reviewers expect code changes to come with a corresponding test change. The lack of such tests is a good sign that *"test coverage is lacking and we're taking a risk accepting the patch"* (R28). The presence of tests in the patch also boosts developers confidence that the patch actually fixes the problem. Many developers run and test patches locally, or when

testing is not practical, they perform manual testing as well. As a part of manual testing, developers often perform an operational proof such as code walks through: *"I walk through the changes, executing it as I imagine the machine would, with as much variety of inputs and states as I can imagine. I look for edge cases. I try to consider what is not in the patch (things that are being affected by the patch but are not directly changed by the patch)"* (R21).

Reviewers pay careful attention on how the patch fits into the existing code base. Integration into the code base can be examined by checking how the patch *"melds with the existing code or how it replaces the existing code"* (R23), *"how much change there is and how far spread the change is"* (R12), or whether *"the patch breaks web compatibility"* (R4). Submitters are often expected to be able to anticipate the upcoming surrounding changes and have an overall understanding of the impact of the change on other areas of the code. To support code maintainability, submitters are expected to conduct refactoring tasks if they see the need for it. Reviewers can request to perform necessary refactoring if they find that the patch is *"contributing to code rot"* (R38).

When reviewing patches, developers often examine whether *software architecture and design* meet expectations. For example, whether a code change *"meets other design considerations (e.g., PEP8 for Python code) (R67)"*. It is expected that submitted changes keep the architecture of the code base intact to facilitate code comprehension and maintenance: *"does it continue the architecture of the existing code or diverge in a way that makes future maintenance difficult?"* (R81), *"I look for architectural impact to see if it is making the code cleaner or messier"* (R87). If the code changes rely on APIs, reviewers check whether they are used appropriately: *"could the new APIs be misused?"* (R65).

Among other characteristics that developers consider when assessing changes are *memory management* such as *"no leaks, no unsafe memory usage"* (R4), *"no accesses to dead objects"* (R9), *security* such as security related checks and return types, *performance* that relates to *"the order of algorithms used"* (R38), *"the right trade-offs between simplicity of code and performance"* (R24), efficiency and speed.

Social factors such as *familiarity with the author* play an important role in evaluating patches. Previous relationships with the submitters, their experience and reputation within the project can determine the fate of their patches: *"I set a baseline based on any previous relationship with the submitter, and the area of code concerned. If I know the submitter I have both some idea of what to check and a better idea if they'll be around later to fix subsequent issues"* (R56), *"past experience of patch author is a big factor"* (R43).

2) Characteristics of a well-done code review.

This research question investigates developer perception of the key characteristics contributing to a well-done code review. Through an open-ended question, we asked developers' opinion on what a high quality review means to them.

The majority of the developers (38%) responded that *clear and thorough feedback* is the key attribute of a well-done review. Reviewers are expected to provide feedback that 1) is clear to understand; 2) is not only *"about code formatting and style"* (R6); 3) provides constructive advice, e.g., *"points out major correctness issues first, and points our minor issues that can be clearly fixed without another round of review"* (R24), *"highlighting potential problems ... and how to fix them"* (R42), *"saying 'this is the worst code I've ever seen' is not constructive"* (R81); 4) is done by the correct reviewer who *"has the domain knowledge to properly evaluate the change"* (R55); 5) is delivered via proper communication: *"good code reviews are dialogues between the reviewer and patch author"* (R50); and 6) provides mentoring and training for patch authors: *"providing detailed mentoring to help them improve faster"* (R56), *"to help the author of the patch become a better programmer in the long term"* (R35).

Developers expect reviewers to have *understanding of the code*, in particular to know *"the code that's being changed and what other pieces of code interact with it and what their assumptions are ("what else could break?") ..."* (R19), *"knowledge of the code is paramount because otherwise reviews are superficial"* (R30). Submitters want reviewers to know the outcome, the impact and *"the side effects of the modified code"* (R49), as well as to ensure that the logic of the patch makes sense. Reviewers are also expected to have an overall understanding of the project's code base: *"enough domain knowledge is always the first criteria for a well-done code review"* (R61) and *"familiarity with utilities in other parts of the repository that could be re-used"* (R38).

We found that *human factors* play a crucial role for developers when receiving feedback. Developers associate good reviews with the reviewers who possess (1) personal attributes such as being *"supportive, yet strict"* (R9), *"patient and stable"* (R61), *"punctual and tactful"* (R28), *"helpful and encouraging, especially when rejecting a patch"* (R55), *"expressing appreciation for contributions"* (R38) especially if contributions come from the newcomers to the OSS community, and (2) inter-personal qualities such as being able to *"establish clear and open-minded communication"* (R73), *"trust the programmer to be competent enough to fix the problems"* (R64), provide positive and constructive feedback *"with the comments written in such a way that the patch author does not take them personally"* (R9) delivered in a *"constructive tone that respects/acknowledges the efforts of the patch writer"* (R21). From the developer perspective, code review relies on the participation of everyone on the

project, and an ideal review process is described as the one that *"allows the author and the reviewer to work together to produce better code than either could on their own, maintain quality standards, and build familiarity with the code base"* (R56).

Code quality, once again, is found to be a vital part of the review process. The review quality is associated with patch writers taking into consideration coding style and formatting, preserving code maintainability, embracing *"current best practice within the project"* (R38). While reviewers are responsible for *"not allowing messy code in just because of time"* (R23), ensuring *"the patch achieves what it was intended to achieve"* (R31) and *"the code adheres to community standards"* (R82).

Quick *turnaround time* is also important for the responses as they report that both parties, reviewers and submitters, are expected be done in a timely manner, *"the value of the review feedback is in the proportion to the cost in terms of delays and time spend"* (R87). However, reviewers are to avoid shipping their feedback *"under stress or when there's a deadline"* (R7) as this introduces risks of missing problems. Some developers noted that Mozilla code review suffers from non-responsive reviewers due to overload or too few reviewers available. As a result, the speed of reviews might overweight the risks: *"depending on what module, a faster yet less thorough review is probably going to be OK, and worth the risk"* (R34).

Testing is seen as a feature that helps to accomplish the review process. During the review, developers are expected to apply the patch locally and test it to make sure it causes no regression. Thorough and careful testing of the patch ensures *"it is doing what is is supposed to and not introducing regressions"* (R42). Among other factors contributing to a well-done review are design and code pattern considerations, providing architectural recommendations (e.g., interaction wit other subsystems, use of correct APIs), and catching the bugs left in the patch.

3) Factors affecting code review quality.

Through a mandatory multiple choice question and an optional open-ended question, we asked participants to express their opinion on the factors they find to influence code review quality. The results of the relevant Likert-scale survey question are summarized in Figure 5.3. The vast majority of the developers agrees that factors such as reviewer experience and technical properties of the patch (patch size, code chunks, number of modified files) are strong indicators of code review quality. Most developers (76-85%) also consider that personal factors such as patch writer experience, reviewer workloads, developer participation in the discussion of code changes, module and number of resubmitted patches are more likely to affect the quality of reviews. While developers have mixed feelings

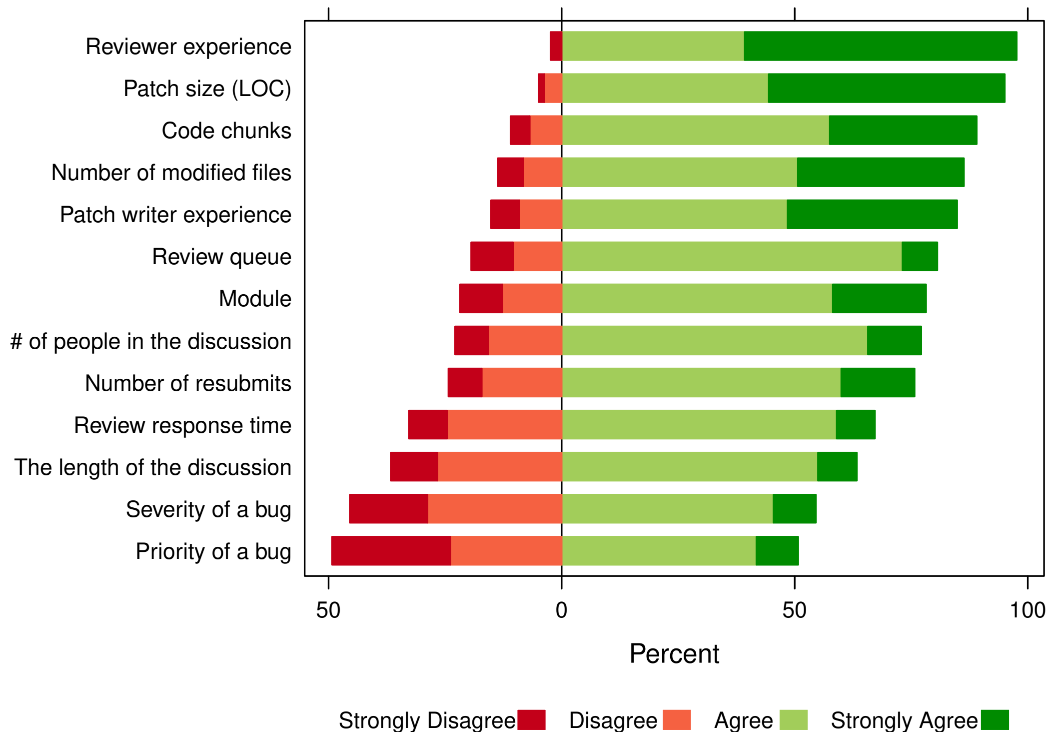

Figure 5.3: Factors influencing code review quality.

about whether severity and priority of a bug, review response time, and the length of the discussion have an affect on code review and its quality.

From the open-ended question, we found a number of additional factors that respondents think are influencing the review quality (but were not present in our multiple-choice question). As we have seen from previous findings, developers consider *understanding of the code base* as an important property that characterizes the review quality: *"domain expertise of both the author and reviewer"* (R35), and *"experience of the reviewer and familiarity with the code or domain are pretty important"* (R85).

Human factors such as reviewer mood, personality, experience, communication skills and style, style of making reviews, and productivity (stress level) are seen as ones of the highest determining factors in the quality of the review. *Time-related* factors such as the time of the day the reviewer gets to do a review, time pressure and deadlines, release schedules, release management, and priorities of other tasks are also among the factors that influence reviewer ability to deliver efficient reviews. Code quality and patch complexity,

presence of tests, tool support, organizational overhead are mentioned as other potential factors affecting the review quality.

Developer perception of the factors affecting code review quality matches the insights we obtained in our qualitative analysis of the data from the project's repositories [71]. Reviewer experience and their work loads, number of previous patches, discussion around issues, as well as technical characteristics of a code change such as its size and the number of files it spreads across are found to be strong indicators of the code review quality.

RQ3: Developer perception of code review quality is shaped by their experience and defined as a function of clear and thorough feedback provided in a timely manner by a peer with a supreme knowledge of the code base, strong personal and inter-personal qualities.

5.3.4 RQ4: What challenges do developers face when performing review tasks?

This research question identifies key challenges developers face when conducting code review tasks. We identified two categories of challenges: *technical challenges* affect reviewers' ability to execute effective reviews, while *personal challenges* relate to their self-management and context switching. We also report the responses to an optional open-ended question about the desired tool support that could help developers with their code review activities.

1) Technical challenges

The biggest challenge for developers is *gaining familiarity with the code.* Since reviewers are often asked to review the code they do not own, understanding the unfamiliar code that the patch is modifying can be challenging, *"our module boundaries are broad, so patches often touch areas that I'm not up-to-date on"* (R37). Developers also find that decision-making of whether the change is good can be difficult, *"it's really important that I understand what the patch does"* (R62). Related to this, reviewers often have to assess whether they are capable of reviewing a particular patch or whether they should delegate it to a different reviewer, *"deciding whether I am the most appropriate reviewer or if my knowledge of the area of code is good enough to be an effective reviewer"* (R57). Reviewers are also expected to *fully understand the problem*, which can be a time-consuming process in particular if they review code in diverse areas of the code base. Reviewers have to not only understand the change but also understand its interactions with the existing code and

being able to determine what code has to be co-changed, as well as *"spot now-redundant code"* (R74).

Another category of the technical challenges is related to *code complexity*. Reviewers are often required to evaluate large patches. The size of the patch is correlated with the quality of the reviews. Large patches are difficult to review because it can be difficult for developers to see the big picture: *"long patches are hard to review - attention wanes, quality of the review goes [down]"* (R21). R12 mentions that *"if large patches are broken up it can still be difficult to understand the bigger picture"*. The complexity of the pre-existing code can add up to this problem. Nevertheless, being able to see the big picture can be troublesome yet very critical for reviewers.

Finally, many reviewers complained about the current *tool support* available to perform review tasks. Some of them mentioned that reviewing in Bugzilla is difficult, while others refer to Bugzilla as *"a pretty good tool"* (R62). Since running automated tests is a part of the review, developers find applying the patch locally and testing it time-consuming. Reviewers mention that existing tools are good at visualizing line-by-line change ("diff" tools) but fall short in providing a summary of what a patch is changing (not on the file level).

2) Personal challenges

Reviewers often find themselves struggling with *time management skills* such as setting personal priorities and beating procrastination: *"convincing yourself that reviews should have higher priority than whatever other work you're doing"* (R19), *"how to get reviewing on a first priority and still getting your own things done"* (R40). All reviewers have other non-review tasks to conduct such as writing patches, participating in discussions, attending meetings (in person or remotely), engaging and recruiting other members to the community, or educating and training the new generation of hackers. Thus, balancing time to perform reviews, as well as all other daily activities can be a struggle. As R38 says *"I try to respond within 24 hours but sometimes a review for 15 patches can just show up out of the blue requiring a full day to review. That throws all other plans out of schedule"*.

On a personal level, reviewers often feel the pressure of *keeping up the personal technical skill level*: *"I need to constantly improve, so I can help others too"* (R52). While reviewers understand the importance of providing guidance and support to new contributors, they admit that this can be very time-consuming and carry risks of landing bug-prone patches to the code repository. As R54 explains: *"reviewing patches by new contributors where hand-holding is needed... it becomes tempting to simply land the patch to end the hassle for both the contributor and the reviewer. This frequently results in buggy code landing in the tree"*.

Several reviewers find it difficult to work on multiple tasks simultaneously. Working on multiple tasks such as performing reviews and fixing a bug is common for developers. *Context switching* from one problem space to another appears to be challenging. More importantly, when the patch undergoes several revisions, reviewers have to keep the context between revisions of the patch to make sure all their concerns with the proposed changes are addressed by the submitter. From the submitter's point of view, keeping track of the comments from the reviewer or other peers can be difficult. This becomes a challenge when a patch writer is working on a large bug or feature that involves a substantial discussion on the best way to resolve or implement it.

RQ4: The key challenges are twofold. Technical challenges are associated with gaining familiarity with the code, coping with the code complexity, and having suitable tool support. While personal challenges are related to time management, technical skills, and context switching.

3) Tools

The majority of respondents perform code review tasks inside Bugzilla that provides a very basic and limited set of code review related features — it allows side-by-side viewing of the patch and code, as well as adding comments to the patch `diff`. The most commonly requested feature is a built-in `lint`-like tool that should provide automatic static analysis, automatic format and style checking, and automatic spell-checking. Moreover, many developers stated that such a tool should not only automatically check the code, but also automatically fix it (where it is possible). Such a feature would allow them to focus on a bigger picture rather spending time on small problems: *"I should be paying more attention to the architecture and the problem solving mechanics of the patch, rather than whether or not the braces are in the right position"* (R21).

Developers also expressed interests in having *better development environment* that offers the ability to easily get the patch from the issue tracking system into the local editor for analysis. Another feature is autolanding (i.e., incorporation into the code base) of patches once they are reviewed. Finally, the developers expressed a desire for direct access to the indexed source code from inside the issue tracker to better understand how the code that being changed is used, as well as for the ability to get the change history of that code.

Almost every code review involves "before-after" comparison of the code. Therefore, it is not surprising that developers want improved support for *diff tools*. The most desired features here are the ability to see the diff *"in the context of the entire file"* (R57) and compare the difference between the original code and code with multiple consecutive patches applied.

5.4 Discussion

We now discuss several research directions that have emerged from this work and can help researchers and practitioners to plan their next research projects.

Reviewer recommender system. Mozilla developers need to control an overwhelming flow of information including bug reports, review requests, updates on their patches, etc. [12]. One way to help developers manage the increasing flow of information is to provide them with tools that can assist them with specific tasks. For example, for code review tasks, a reviewer recommender system could be able to help both reviewers and patch writers determine the right person to review a code change at hand considering reviewer code/module expertise, his or her current review loads and availability (schedule). For example, R8 asks for a tool to be able to "automatically identify potential reviewers based on similar types of code change (perhaps in other modules)". While there is a large body of research that addresses this problem of expertise recommendation and offers a variety of techniques [5, 6, 84, 123], most existing solutions are research tools that do not scale well or would otherwise be impractical to deploy within an industrial environment.

Next generation code review tool. Many reviewers expressed concerns with current support for tasks related to code review. Code review is an essential part of the development process at Mozilla; yet respondents complained about the lack of a good code review tool. We found that developers expressed interests in having an online code review tool that supports automatic static analysis, automatic format and style checker, as well as automatic spell checker. These features can help developers with their time-management by allowing them to focus on the code change and how they fit into the bigger picture rather than paying attention to the formatting and style nits. The next generation review tool should also support code indexing and navigation for reviewers to be able to better understand code modifications and their interactions with other areas of the code base. Another desired feature is related to developing better `diff` tools to enable the comparison of different versions of the code or tracking of individual code changes. Reviewers also commented on the importance of having the ability to compare code on "file-by-file" rather then "line-by-line" level and to determine the differences between multiple consecutive patches.

Reshaping OSS. While we only attracted and surveyed Mozilla developers, the results of our study can be applied to other OSS projects such as Linux, Apache, Red Hat, etc. Most recent studies were either conducted at Microsoft [8] or focused on the pull-based development model [52]. While pull-based development (e.g., via GitHub) is gaining popularity among distributed software development community, the need to continue studying and supporting the evolution of large long-lived OSS projects remains as important as

ever. We noticed that some developers are interested in borrowing emerging technologies (e.g., GitHub) and bringing them to their own working environments. OSS projects are constantly reshaping themselves [3], and researchers can facilitate their growth by helping them address their practical needs and overcome the obstacles they face. Having said that, our study adds to the existing body of knowledge on code review.

5.5 Threats and Limitations

The first limitation lies in the validity of our findings from the qualitative study. While we carefully designed our survey questions to ensure their clarity, as with all exploratory studies, there is a chance we may have introduced the researcher bias when applying coding to the open ended questions. We tried to minimize this by coding the 20% of the card sorts extracted from each question independently, measuring the coder reliability on the next 20% and reporting these values in the chapter (see Table 5.1).

As with any survey method, to control for sampling bias can be challenging. We targeted the core developers of the Mozilla community who actively participate in code review tasks either by evaluating patches of their peer developers or submitting their own code changes to reviewers for quality assessment.

We only survey developers from one large open source community, yet we targeted Mozilla's core developers who are full-time employees. While our findings might not generalize outside of Mozilla, we believe any medium and large open source project employ similar code review practices. Nevertheless, further research studies are needed to be able to provide greater insight into code review quality and develop an empirical body of knowledge on this topic. To encourage replication of our study, we documented our survey questions and card sort results in a technical report that is made available online [69]. We also made anonymized survey responses publicly available.

5.6 Summary

Code review is a vital element of any long-lived software development project. A high-quality execution of this process is essential to ensuring the ongoing quality of project's code base. This work explores the code review practices of a large, open source project and aims to understand the developers' perception of code review quality. To accomplish this, we surveyed 88 core contributors to the Mozilla project. The qualitative analysis of

the survey responses provides insights into the factors that affect the time and decision of a review, the perceived review quality, and the challenges developers face when conducting code review tasks. Our findings suggest that the review quality is mainly associated with the thoroughness of the feedback, the reviewer's familiarity with the code, and the perceived quality of the code itself. We also found that developers often struggle with managing their personal priorities, maintaining their technical skill set, and mitigating context switching.

Chapter 6

Code Review in Pull-Based Development

In the previous chapter we were focused on the review process as it happens in more traditional approaches to evaluating code contributions. However, recently pull-based development has become a popular choice for developing distributed projects, such as those hosted on GitHub. In this model, contributions are pulled from forked repositories, modified, and then later merged back into the main repository. In this chapter, we report on two empirical studies that investigate pull request (PR) merges of Active Merchant, a commercial project developed by Shopify Inc. In the first study, we apply data mining techniques on the project's GitHub repository to explore the nature of merges, and we conduct a manual inspection of pull requests; we also investigate what factors contribute to PR merge time and outcome. In the second study, we perform a qualitative analysis of the results of a survey of developers who contributed to Active Merchant. The study addresses the topic of PR review quality and developers' perception of it.

Chapter Organization. Section 6.1 highlights the direction of the research in this chapter. Section 6.2 describes the methodology we followed in our study. In Section 6.3, we present the results of our qualitative and quantitative analyses. Section 6.4 discusses our findings, while Section 6.5 addresses threats to validity. Finally, Section 6.6 summarizes our results.

Related publication. The work described in this chapter has been presented in the following paper:

- Oleksii Kononenko, Tresa Rose, Olga Baysal, and Michael W. Godfrey. Studying Pull Request Merges: A Case Study of Shopify's Active Merchant. *Under review.*

6.1 Introduction

Pull-based software development has gained substantial popularity in recent years. Hosting services that support this style of development, such as GitHub and Bitbucket, have attracted a huge number of new and existing projects: GitHub alone is estimated to have more than 19.4 million active repositories[1]. Unlike more traditional approaches to evaluating code contributions, in the pull-based model developers make changes to an isolated copy of the project's repository, and then later submit a *pull request* (PR) to the owners of the project to incorporate their changes into the main codebase; of course, the project owners need to evaluate the proposed changes, and decide whether to merge the new code into the project in the main repository.

Many previous studies, both qualitative and quantitative in nature, have mined data from GitHub to investigate a variety of research questions. Since the vast majority of public software projects on GitHub are open source software systems (OSSs), the findings from those studies likely reflect the kinds of processes typically used in open source development. However, there are also commercial projects on GitHub that allow their repositories to be viewed by the public, while at the same time maintaining strict ownership and tight control over their ongoing evolution. Such repositories give researchers a unique opportunity to study industrial software development processes. We decided to perform an in-depth study on one such project.

We had three criteria for selecting an appropriate project to study. First, while many projects host their codebase on GitHub, it is also common practice to perform day-to-day development activities elsewhere using private external repositories, with only occassional large updates being made to the public repository [63]. Therefore, one criterion was that the selected project should be developed completely on the GitHub platform so we would be confident that the granularity of observed activities was that of day-to-day development. A second criterion was that the selected project should be commercially successful, since successful projects are more likely to systematically employ similar practices that supported their success. Finally, we wanted to study a project that had been developed in physical proximity to our labs, in case we wanted to interview members of the main development team. Based on these criteria, we decided to study the Active Merchant project developed by Shopify Inc. which has offices in Ottawa, Toronto, Montreal, Waterloo, and San Francisco.

Shopify is an e-commerce company that provides a platform for online stores. As of February 2017, more than 377,500 merchants use this platform to sell commercial

[1] https://octoverse.github.com

goods [112]. Active Merchant is a part of that platform; it is a payment abstraction library that handles and unifies access to a variety of payment gateways with different internal APIs. The development of that project is done completely on GitHub. All PRs must pass a code review process and get approval before being merged into the main codebase. Although this project is owned by Shopify, Spreedly Inc. has recently become an active contributor as well.

Our investigations are built around a quantitative analysis of the Active Merchant project repository, as well as an exploratory survey that we conducted with Shopify developers. Our goal is to answer the following research questions.

RQ1 *Which merge strategies are used by developers? Do they affect the pull request review time?*

A submitted PR may consist of multiple commits; a developer can add yet more commits to address reviewers' comments. If the PR is accepted, developers can incorporate the commits in several ways [63]. We study how developers merge PRs, as well as fill the gap in the existing research by analyzing the effect the PR merge type has on the merge time.

RQ2 *What factors affect the PR review time and decision?*

Previous studies have looked into effect of a variety of factors that concern the time needed to reach a decision about a PR, as well as the effect on that decision itself [50, 121]. We investigate what factors play a role in the studied commercial project.

RQ3 *How do developers perform and assess the PR review process?*

We believe that the data extracted from project's code repositories would tell us only part of the full story. To get a more comprehensive view of the pull-based development process used by Active Merchant, we conducted a survey among Shopify developers. The analysis of the survey results provided us with a better understanding of the developers' perception of PRs and PR review quality.

This chapter makes the following contributions:

- An in-depth study of new contributions in a pull-based software development model within a successful commercial software project.
- A survey with professional full-time developers that offers insight into their perception of the process of assessing of new PRs.

- A publicly shared dataset[2] that includes mined data with manually verified and labeled merged PRs, the survey questions used, and the anonymized survey responses.

6.2 Methodology

To answer our research questions, we performed a combination of quantitative and qualitative analyses. Our quantitative study consisted of mining software artifacts from Active Merchant's GitHub repository [111], pre-processing, and analyzing the extracted data. For our qualitative study, we surveyed the developers involved in the project development.

6.2.1 Data Mining

Data collection. Active Merchant is a library that provides a unified API that allows communication with many different payment gateways. The project is hosted on GitHub and employs a pull-based mechanism for submitting and accepting code contributions, as well as performing review of those contributions. GitHub provides APIs that allow users to access its data [47]. We used an official library developed by GitHub to make API calls [46]. For our study, we looked at the contributions made to Active Merchant between January 1, 2012 and October 1, 2016; we extracted a total of 1,657 pull requests (PRs) that were submitted during this period. During the extraction process, we tracked a variety of information about each PR, including the unique ID of its author, the date the PR was added to the repository, the date the PR was closed, whether the PR was merged and (if so) the date of the merge, the natural language description of the PR, and its size statistics.[3] For each PR, we also collected both PR-wide comments and in-code comments left by the developers.

GitHub user accounts of many Active Merchant contributors do not contain affiliation information or email address. To recover missing email addresses, we extracted the actual commits from the repository. For each commit, we analyzed commit author information recorded by GitHub (represented by unique user ID) and commit author information recorded in the header of the commit by Git (Git identifies users using their email address, so this field is always present). If GitHub user data was missing, we used name (if available) and the email address from the commit header. While it is possible that some of these email addresses are inaccurate, using this approach we were able to reduce the number

[2] `http://swag.cs.uwaterloo.ca/~okononen/shopify`

[3] We did not calculate the size statistics ourselves; instead, we relied on the values provided by GitHub.

of anonymous contributors. To recover developers' affiliation information, we parsed the email addresses and set it based on the domain name of the emails (except for "public" emails such as Gmail, Yahoo, etc.).

PR merge types. Kalliamvakou et al. noted that GitHub data is not always reliable regarding whether a PR has been merged or not [63]. The discrepancy between the recorded merge information and the actual merge status exists because developers can merge a PR using several different approaches. We used the heuristics proposed by Kalliamvakou et al. [63] to recover "missing" merge flags. These heuristics are based on commits in the master branch of a repository as well as on the content of the last comments left on a PR. For example, according to one heuristic, a PR was merged if there is a commit in the repository's master branch and that commit closed a PR using a specially formatted message appended to the commit message. By applying these heuristics, we were able to mark 798 "not merged" PRs (as reported by GitHub) as "merged" ones. Kalliamvakou et al. also warned that their heuristics may result in a considerable number of false positives. To reduce this risk, we performed a manual inspection of the merged PRs; the inspection also afforded us the opportunity to label each PR according the merge type labels suggested by Kalliamvakou et al.:

- *GitHub merge* — a merge performed using GitHub facility (i.e., using the "merge" button in the UI).
- *Cherry-pick merge* — a merge when a developer selects a subset of commits from a PR and adds it to the repository without any changes.
- *Commit squashing* — a situation when a developer creates a new commit that contains all commits from a PR, makes additional changes if needed, and adds this commit to the repository.

Manual inspection of pull requests. Researchers Kononenko and Rose performed the inspection of all PRs marked as "merged" by the stated heuristics. To ensure that the researchers had the same understanding of the merge types, we selected 20 random PRs and performed independent labelling of each PR. We then compared the assigned labels and calculated intercoder reliability score (i.e., percent agreement). The researchers achieved high agreement (93%) between themselves: they differed only in two PRs. After that, the remaining PRs (778) were split in two sets; two researchers separately inspected and labelled one of these sets. As a result of this inspection, we found seven PRs that were incorrectly marked as "merged".

Data pre-processing. To minimize any potential noise in the collected data, we tried to eliminate outliers by applying three filters:

- We removed 5% of the PRs with the longest review time to account for PRs that struggled to catch developers' attention. Several PRs took an extremely long time to get reviewed; for example, the longest review took 637 days, while the median for review time is 3 days.
- Some PRs are unusually large in terms of added/removed lines of code (LOC) — the biggest PR is nearly 1 million LOC, while the median is only 35 LOC — and to account for such PRs we removed the largest 5% of all PRs.
- Since we were interested only in studying those PRs that the developers had decided on, we removed all PRs that were not marked as "closed" (26 PRs in total).

After applying these filters at the same time, our dataset was reduced to 1,475 PRs.

6.2.2 Explanatory Factors

Previous research suggests a set of different metrics that can affect code review time and outcome [15,50]. Table 6.1 lists the explanatory factors we considered in our study. The selection of each factor was governed by our ability to accurately calculate its values from the mined data (i.e., we did not include a factor if we could not collect the data corresponding to that factor or if we needed to apply some heuristic to compute its value).

Although GitHub allows a PR to be assigned to a specific developer, we found that this feature was rarely used within the Active Merchant repository. Therefore, one of the challenges we faced was determining the exact time boundaries of a review period. We considered the PR submission date to be the date that the review process started. Since a PR cannot be merged before it has passed the review, we considered the date a PR was closed as the date the review process ended. Thus, the time between these two dates (i.e., start and end dates of review) is defined as review process length.

Another challenge we had to overcome was the lack of standardized flags or labels in GitHub to indicate the outcome of a PR review. In the studied repository, some reviewers add a textual comment, some use emoticons, while others include images of boats (meaning "ship it"). We marked all closed and merged PRs as the ones that successfully passed the review, and all closed but not merged PRs as the ones that received a negative review. Similar assumptions were made in [50, 121].

Table 6.1: Overview of the factors studied.

Explanatory Factor	Description
PR size	Sum of added and removed LOC
# files	# files changed by a PR
# commits	# commits in a PR
PR author experience	# prior PRs submitted by PR author
# comments	# comments left on a PR
# author comments	# comments left by the PR author
# commenting developers	# devs participating in discussion
# in-code comments	# comments left on source code
# author in-code comments	# comments left on source code by author
# in-code commenting devs	# devs who commented on source code
PR author's affiliation	An org that a PR author affiliates with

6.2.3 Data Analysis

To understand the effect of the selected factors on review time and review outcome, we built Multiple Linear Regression (MLR) and Logistic Regression Models respectively. These models try to capture the relationship between the explanatory variables — in our case, the factors described in Table 6.1 — and a response variable — i.e., the PR review time and review outcome [27]. Our goal of understanding the relationship between explanatory and dependent variables, as well as our model construction process are similar to the ones in the previously published studies [24, 71, 80, 85].

Variable transformation. Empirical evidence suggests that software engineering data is rarely normally distributed [81]. To minimize any possible skewness in the data, we applied a log transformation *log(x+1)* to all continuous variables (e.g., size, author experience, comments, etc.). Because categorical variables (e.g., `affiliation`) cannot be used directly in regression models, we employed a dummy coding technique to transform a categorical variable into a set of dichotomous variables that capture the same information.

Controlling Multicollinearity. Multicollinearity is defined as a high correlation among two or more explanatory variables in a regression model. We checked the models for multicollinearity using the variance inflation factor (VIF). A VIF score of each variable represents how much its variance is explained by the collinearity with other variables. We used `vif` function from the `R car` package to calculate VIF scores [43]. As recommended in [42], the threshold for VIF score was set to 5. Through an iterative process, we checked that our models contain only variables with VIF scores lower than the threshold; at each

iteration, if there was a variable with a VIF score higher than the threshold, we removed that variable and recalculated the VIF scores.

Model Evaluation. To evaluate our models, we considered R^2 values. For our MLR model, we used *Adjusted* R^2 value [55]; unlike R^2, this value is affected by the extra variables with low explanatory power in the model: the more such variables, the lower the value. To reduce the number of "useless" variables in the MLR model, we used a bidirectional stepwise selection technique [41], a process of adding and removing independent variables for finding a best subset of such variables. There is no R^2 value for Logistic Regression Models; instead several statistics, so called "pseudo R^2", have been proposed. We are using the one proposed by Tjur — Tjur's D [120]. This statistic is closely related to the definition of R^2 in MLR models, and it is designed for dichotomous dependent variables. We used R `binomTools` package to calculate Tjur's D [25].

6.2.4 Survey Design and Participants

To understand developers' work practices and their vision of the pull request review process established in the project, we decided to conduct a survey with them. Similarly to previous studies [50, 70], we designed a survey containing three groups of questions: nine questions related to the demographic information about participants and their work practices, three Likert-scale questions focused on PR review, and four open-ended questions asked participants to provide more information concerning their responses to the Likert-scale questions. The questions from the survey are presented in Appendix B. Participants were informed that our survey would take 10–15 minutes to complete.

Since Active Merchant is a product of Shopify, we targeted Shopify developers because they own the product and remain its main contributors. Our dataset included 88 developers who were affiliated with Shopify. To recruit participants for our survey, we sent out 78 personalized emails inviting developers to participate in the survey; 10 developers in our database had missing email addresses. For five of those emails we received an automated response saying that the email address had been deactivated; we speculate that these developers are no longer with Shopify. The survey was open for two weeks — from February 27, 2017 to March 13, 2017 — and we received 16 responses. While the number of responses may seem small, the response rate of 22% (16/73) is higher than the suggested minimum response rate of 10% [54].

6.2.5 Card Sorting

We employed a grounded theory approach for analyzing the developers' responses to open-ended questions in the survey. Before our analysis we had no preconceived ideas or theories about the survey responses, so we used an open coding approach to create categories and themes, and to group the data into them [83].

Researcher Kononenko split 16 survey responses into 181 isolated quotes (cards), i.e., each quote represents a single statement that differs from other statements in a particular answer to a question. After that, two researchers, serving as coders, went through the cards grouping them into themes, and later grouping themes into broader categories. The coders used the following protocol on all but one of the open-ended questions:[4]

- The coders took the first 25% of cards that correspond to a particular question and — independently of each other — organized them into several groups. Once they were done with the first round of card sorting, they compared and discussed the emerged groups and the cards in them.
- During the next round, the coders took another 25% of cards and — again, independently of each other — sorted the cards into the groups created in the previous step. If a coder believed that a card did not match any of the existing groups, they were allowed to create a new group for that card. To ensure the integrity of the process, we computed the intercoder reliability at this step.
- During the final round, the coders sorted the rest of the cards (i.e., the remaining 50%) together.

We opted for percent agreement as our intercoder reliability coefficient as it is one of the most popular reliability metrics. We used ReCal2 to compute percent agreement values for each question [44]. The values of the reliability coefficient were different among questions and ranged between 91.9% and 100%, with the average of 96.4%.

6.3 Results

In this section we present the results of our quantitative and qualitative studies, and we answer our research questions.

[4]One of the questions received only few responses making it impractical for us to apply the described procedure.

6.3.1 RQ1: Merge types and their effect on review time.

In a more traditional development model, if a contribution (e.g., a patch) is approved, it will be added to the repository. If such a contribution has undergone a series of revisions before being approved, only its final version will be incorporated into the codebase. Pull-based development, on the other hand, has created a new way of dealing with incoming contributions to a software project. When dealing with PRs, developers have more flexibility: they can choose to incorporate as little as they deem beneficial to the project or they can decide to merge a complete PR. If they decide to incorporate the full PR, they have another choice: they can either leave the commits from a PR untouched — preserving more historical information this way — or "squash" the commits into a single commit and add it into the repository — to have a cleaner commit history on the master branch. While researchers have studied different aspects of pull-based development, we did not find any studies that analyzed the merge approaches used by developers. Thus, first we decided to take an exploratory look at the PR merge strategies.

As described in Section 6.2.1, we applied several heuristics to determine merged PRs; later, we also manually inspected and labelled all merged PRs in our dataset. Table 6.2 reports the results of this manual classification. While we were surprised to see that only a small number of PRs — about 25% — were merged using the native GitHub UI, it might be due to the fact that the merge via "merge" button is possible only if (a) there are no further changes required, and (b) GitHub can automatically resolve a merge conflict if it occurs. In fact, during the manual inspection we noticed that a lot of merged PRs had additional changes made by a merger. When developers were merging a PR, they often updated `changelog`, `readme`, and/or `contributors` files to reflect the new change. Squashing was the most popular merge type used by developers. Although some historical information is lost during such a merge, there are some advantages too. The main benefit here is that the squashing merge helps developers to keep the commit history in the master branch simple and relatively clean: each commit represents a single PR. With such an organization of the branch, developers can ensure that the commits are accompanied by a descriptive log message; also, it is easier for developers to revert a merged PR if there is a need. Moreover, squashing naturally supports both the small additional changes the mergers make and the automatic closure of a PR.

To study whether merge types have an effect on the time a PR stays open — i.e., until it is ultimately rejected or merged — we used two non-parametric statistical tests: Kruskal-Wallis analysis of variance [73], and a post-hoc Mann-Whitney U (MWW) test [74]. The Kruskal-Wallis test revealed that merge type has a statistically significant effect on time ($\chi^2(3)$=89.02, $p < 0.001$). Since this test does not show where the significance occurs,

Table 6.2: Classification of PRs by a merge type.

Merge type	**Count**	**Percent**	**Median review time (in min)**
Not merged	372	25.2%	10,111.5
GitHub	367	24.9%	1,107.8
Squashing	612	41.5%	4,241.8
Cherry-picking	124	8.4%	3,177.6

we followed up with pairwise comparison using MWW test with Bonferroni correction. The test showed significant difference among all pairs except one: the difference between *cherry-picking* and *squashing* was not statistically significant, although the median time for the former is smaller than the median time for the latter. We report the median time for each merge type in Table 6.2. The huge difference in median time between unmerged PRs and merged PRs might indicate that if a PR is going to be accepted, it will be accepted quickly; otherwise, it will be shelved and "forgotten".

> *RQ1: While most developers merge pull request via squashing (41.5%), GitHub and cherry-pick PR merge types are also a common practice. Merge type has a statistically significant effect on PR merge time; cherry-picking and squashing merges take more time than the merges done via the GitHub UI.*

6.3.2 RQ2: Factors affecting merge time and decision.

To investigate which factors influence the time developers take to make a decision about a PR, as well as the factors that affect the review decision (to merge or not to merge), we built two statistical models: Multiple Linear Regression Model for PR review time, and Logistic Regression Model for PR review decision. The models used factors from Table 6.1 as independent variables. We removed `merge type` factor from the decision model because it implicitly reflects (i.e., is correlated with) the dependent variable. Table 6.3 reports the regression coefficients for each of the studied factors. In addition to the qualitative analysis, in our survey, we asked developers which factors they experienced to be influential to time and decision. We used 4-point Likert-scale questions, as well as open-ended questions to obtain developer insights.

Merge time. The MLR model indicates that the PR size has a statistically significant effect on review time. The positive value of the regression coefficient means that the larger a PR is, the longer it takes for developers to review it. PR size was also seen as influential

Table 6.3: Models for fitting data.

	PR review time	**PR review outcome**
	Adjusted R^2: 0.28	Tjur's D: 0.14
Size (LOC)	0.110*	-0.187***
Number of files	‡	‡
Number of commits	·	‡
Writer experience	-0.285***	0.217***
# of comments	†	†
# of commenting devs	1.021***	-0.786***
# of author comments	‡	·
# of in-code comments	†	†
# of in-code commenting devs	0.389*	0.357*
# of in-code author comments	‡	‡
Affiliation with Shopify	-1.721***	1.160***
Affiliation with Spreedly	-1.980***	1.046***
Cherry-pick merge	0.654*	n/a
GitHub merge	-0.610**	n/a
Squashing merge	0.804***	n/a

†Removed during VIF analysis.
‡Removed during stepwise selection.
Stat. significance codes: *** $<$ 0.001 $<$ ** $<$ 0.01 $<$ * $<$ 0.05 $<$ ·

by almost all developers who participated in the survey (Figure 6.1). These findings are similar to the previous research studies [15, 62, 70, 124].

Writer experience appears to have a statistically significant effect on review time. Negative regression coefficient for this factor demonstrates that more experienced developers tend to have quicker turnaround time for their PRs. One possible explanation for this finding is that experienced developers might be more familiar with the codebase and project culture, and thus are likely to submit PRs that fit better into the project. While we did not consider reviewer experience as a factor — because it was not feasible to calculate it accurately — it was one of the factors we asked developers about. Developers believe that both author and reviewer experience are important contributors to PR review time (81% and 87% of positive answers respectively).

Out of six discussion-related factors only two made it to the final model: the number of developers who left comments at a PR level and at a source code level. Both of these metrics have positive regression coefficients, indicating that each new developer participating in a

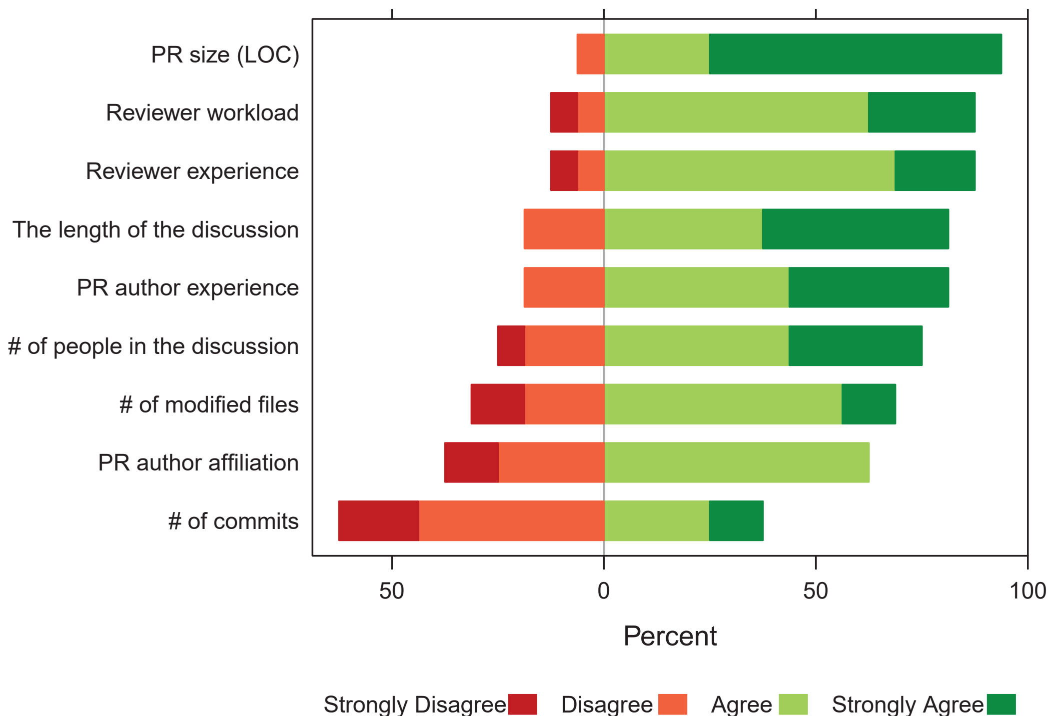

Figure 6.1: Factors influencing PR review time.

discussion delays the decision on a PR. Discussion of new contributions is vital to the health of software project, thus a thorough discussion should likely be welcomed although it can cause a delay. At the same time, we were surprised to see that the PR author comments factor was not present in the model. It is author's job to explain a proposed change and address reviewer comments; our intuition is that the lack of such comments would only delay the final decision. Contrary to the results of the model, developers indicated that the length of a discussion is a critical factor (81% positive responses).

The model shows that the PR author's affiliation influences review time as well. Pull requests submitted by Shopify developers (owners of the project) or Spreedly developers (who work very closely with Shopify on this project) receive faster reviews on their PRs than PRs submitted for review by external developers. This finding is somewhat similar to the one made by Baysal et al. [13] who studied code review of Mozilla project. Surprisingly, few developers agreed with the statement that author affiliation affects PR review time. Perhaps, these developers did not participate in a review of external PRs, or because they believe that the established process is impartial.

Merge type was also included in the final model; the findings here are similar to the ones presented in Section 6.3.1. A merge that is performed using GitHub UI is correlated with shorter review time, while the other two merge types are associated with longer review time.

The open-ended questions of the survey provided developers an opportunity to discuss any other factors not covered by the Likert-scale questions. The open coding analysis of the open-ended questions also revealed several additional factors that developers believe have an impact on the PR review time. The biggest theme identified in the responses is *PR quality*, which includes *PR description* and *PR complexity* categories. As explained by D9, *"bad descriptions are the biggest factor; someone submitting a 1,000 LOC PR with a good description is much better than someone submitting a 100LOC PR with only 'Added XXX integration"'*. Several developers believe that *type of change* (e.g., *"new feature, refactor, bugfix, etc."*) and where the change affects code's *architecture/design* (e.g., *"big refactoring"* (D14)) are also important factors affecting review time. *Human factors* such as *"trust you have in author"* (D6) and reviewers' familiarity with *"that part of the codebase"* (D2), PR discussions that may take place across multiple channels *"in a GitHub issue, face-to-face, etc."* (D10), the *"set up of the tophat"* (i.e., *testing*) are all considered by developers to contribute to the PR review time.

Merge decision. Similar to the previous model, the PR size metric is included in the final model for review outcome (i.e., merge decision). Its negative regression coefficient indicates that larger PRs are more likely to receive a negative merge decision (i.e., a PR is not merged) than the smaller ones. However, when we asked developers what they think about the influence of this factor, the answers were split: only 56% of developers agreed that the size affects the review outcome (Figure 6.2). A possible explanation for this is that a larger PR has a higher chance of containing more than one "atomic" change, which is against PR submission policies in many software projects. While developers may not mind accepting a large PR that is coherent and single-purpose, they may feel more negatively about a large PR that is a collection of loosely related changes.

The PR author experience has a statistically significant effect on merge decision. The more PRs a developer has submitted in the past, the more likely their PR will be accepted again. Developers actively contributing to the project are often well known to other developers, and therefore the acceptance of a PR might be affected by their reputation [19] or interpersonal relationship with other developers [70]. Surprisingly, when we asked developers about impact of experience on merge decision, they believed that neither author nor reviewer experience affects it (63% of negative responses for each factor).

Similar to the model on time, the only discussion-related factors in the model are the number of people leaving comments and the number of people commenting on the

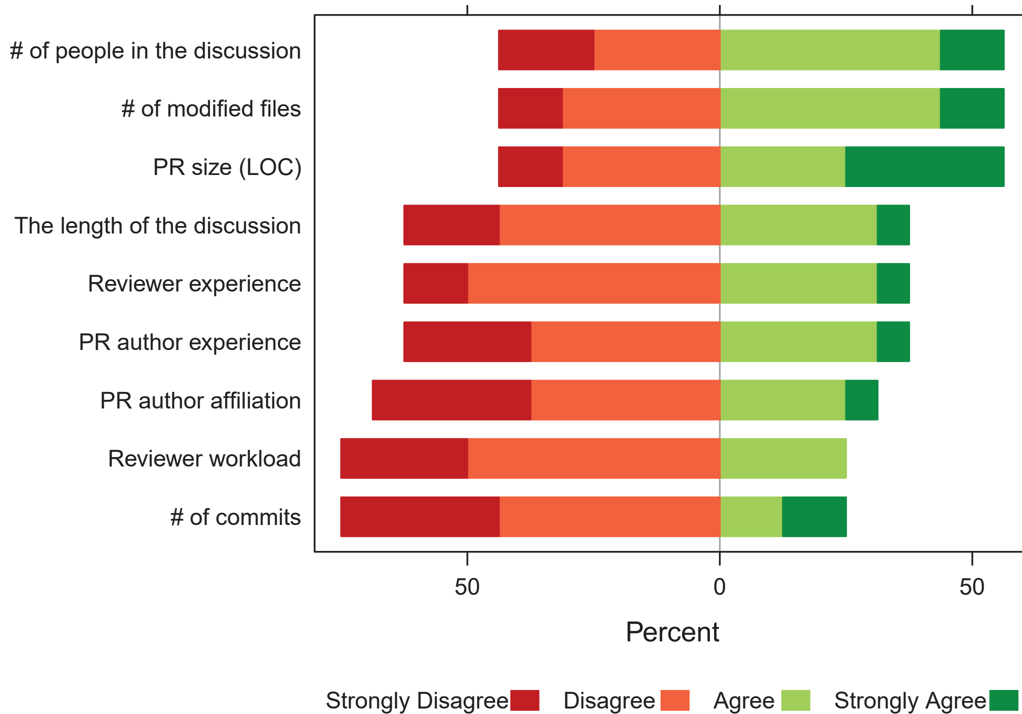

Figure 6.2: Factors influencing PR review outcome.

source code. However, in this model, these factors have the opposite effect on the merge decision: the higher number of developers commenting on a PR leads to lower chances of it being approved, while the number of developers leaving comments on the source code is likely to increase the chance of a PR being accepted and merged. We speculate that this happens in the situations when PR comments are likely to be more 'high level' (e.g., the need for such a change, alignment with project's goals, etc.) while the comments on the source code, by definition, are more 'low level' (e.g., implementation, API choice, etc.). At the same time, the discussions regarding high-level issues are likely to be controversial; therefore, bringing more developers to such discussions might prevent them from reaching an agreement. Developers agreed that the number of people involved in a PR discussion affects its acceptance (56% of positive responses), however, they disagreed the length of that discussion plays a role (63% of negative responses).

The affiliation of the PR author has a statistically significant impact on the merge decision. Both Shopify's and Spreedly's regression coefficients are positive, indicating that the PRs that come from the developers of these two organizations have a higher chance of being accepted.

In analyzing developer responses to the open-ended survey question related to review decision, we observed that several categories emerged. The highest impact on the PR review decision is perceived to be *PR quality* including its complexity, *"can we instead generalize this feature? can you do instead something simpler using the current functionality of the code?"* (D12). Developers also argue that PR author's responsiveness and workloads (we put these under *human factors*) affect reviewer's decision on whether to merge the PR. As D6 explains, *"mostly if people have time, most PRs seem to eventually get merged as long as the author has time to fix things."* Some developers find that overall project schedule such as *"release plan"*, *"code freeze"* (D14) may also impact merge decisions.

RQ2: The statistical models revealed that both PR review time and merge decision are affected by a PR size, the discussion, as well as author experience and affiliation. Developers believe that PR quality, type of change, and responsiveness of an author are also important factors.

6.3.3 RQ3: How developers perform and assess PR review process

During the open coding process, 22 key categories (including "irrelevant") emerged; Table 6.4 presents these categories in detail reporting the number of quotes, the number of respondents, the question numbers, and the totals for each question.

PR review process of Active Merchant. Since each organization adopts its own code review guidelines and practices, we first wanted to understand how developers conduct reviews of the Active Merchant PRs. In particular, in our survey we asked developers about the steps they typically follow when they are asked to review a PR.

The analysis of the survey answers shows that developer see *testing* as the key feature of the PR review process: new code must pass automated tests and be peer reviewed. Thus, one of the first steps is to "tophat" (i.e., test the code locally). To do so, reviewers check whether the PR author has tested the code, *"did you tophat the change? what steps did you follow?"* (D10). D10 further explains that Shopify requires *"separate tophats (testing) from someone other than the PR author before submission"*. Therefore, reviewers *"look for tests and what kind of tests and where they cover"* (D1) and *"make sure that test cover all changes"* (D10).

The next key step of the process is to *understanding the scope of the change*. First, reviewers read PR title and description and *skim through*, i.e., *"an initial pass to get a*

Table 6.4: The categories created during open coding.

Category	Q9		Q11		Q13		Q15		Q16	
	#Q	#R	#Q	#R	#Q	#R	#Q	#R	#Q	#R
Understanding context/rationale /scope	19%	60%	-	-	25%	50%	16%	33%	7%	20%
Code inspection	15%	70%	-	-	-	-	-	-	-	-
Touchy code	4%	20%	-	-	-	-	-	-	-	-
Testing	22%	70%	5%	10%	8%	17%	6%	20%	14%	33%
PR complexity / granularity	4%	20%	30%	50%	25%	33%	12%	33%	-	-
PR description	-	-	25%	30%	-	-	20%	47%	-	-
Skimming through	4%	20%	-	-	-	-	-	-	-	-
Catching bugs	2%	10%	-	-	-	-	-	-	5%	13%
Refactoring	2%	10%	-	-	-	-	-	-	-	-
Comments/discussion	11%	40%	10%	10%	-	-	4%	13%	7%	20%
Code quality	2%	10%	-	-	-	-	18%	47%	7%	20%
Architecture/design	9%	10%	10%	20%	-	-	-	-	5%	13%
Type of a change	-	-	10%	20%	-	-	-	-	-	-
Familiarity/knowledge of code-base	-	-	5%	10%	-	-	-	-	-	-
Release schedule	-	-	-	-	17%	17%	-	-	-	-
Human factors (e.g., experience, trust)	-	-	5%	10%	17%	33%	2%	7%	2%	7%
Time	-	-	-	-	-	-	2%	7%	11%	33%
Feedback	-	-	-	-	-	-	-	-	32%	60%
Conformance to project goals	-	-	-	-	-	-	6%	13%	9%	13%
Performance	-	-	-	-	-	-	2%	7%	-	-
Revertability	-	-	-	-	-	-	6%	13%	-	-
Irrelevant	7%	30%	-	-	8%	17%	8%	27%	2%	7%
Total	54	10	20	10	12	6	51	15	44	15

Notes: #Q: the number of quotes, #R: the number of respondents, Q9: PR review process, Q11: factors affecting time, Q13: factors affecting decision, Q15: characteristics of PR quality, Q16: characteristics of PR review quality.

sense of what it's about" (D6). And next, they try to *"understand the full extent of the change, not simply the changes in the* ***diff****"* (D10). D1 reports that *"I read the what, how and why you are trying to do with your PR."* To understand why a change was made, sometimes developers need to *"jump to different parts of the code as necessary to reference other changes"* (D8). At the end of this step, reviewers pay close attention to *"touchy code"* (D1), i.e., *"things that look weird"* (D6).

Code inspection is an integral part of the PR review. Reviewers look at the code and evaluate *code quality* according to code guidelines. For example, D10 checks if PR author

"named everything correctly and in an intuitive way: variables, tests". Some reviewers apply different code inspection strategies depending on the type of change they review. D8 reflects on his approach: *"If it's a bugfix or feature, start reading through the changes in one window (top to bottom). If it's a refactor, open the code in two windows side-by-side, so additions and deletions can be browsed independently"*.

Reviewers typically provide their feedback — in a form of comments or questions — to a PR author to *"discuss the solution and the approach, not just the implementation"* (D10). Such *discussions* are critical as they are seen as communication mechanisms between PR author and reviewer. Reviewers check whether the PR author has addressed their questions, as D8 elaborates *"I circle back on questions and comments to see if they've been answered by code later in the PR"*.

Apart from code quality, reviewers also check for any violations related to *architecture and design*. Such inspections can be performed on code itself (*"are any abstractions leaked into code?"*), tests (*"are the tests tightly coupled making refactoring harder in the future?"*) or use cases (*"is any complexity added from tyring to anticipate future use-cases of the code?"*). Also, reviewers check whether PR author has used the *"best methods"* of implementing a piece of functionality.

Developer perception of a PR quality. Enforcing quality standards is a key aspect of code review; standards of both code and project development must be met by proposed PRs. With this research question, we explore how developers define and view PR quality.

One of the main attributes for developers when evaluating PR quality is its *description*. Developers believe that the PR description should be thorough, explaining *"what it's solving and why"* (D8), describing *"happy/unhappy paths"* and *"possible errors/problems that are not fully solved"* (D11). Some respondents also said they wanted PR descriptions to include *"potential alternatives"* to the solution and *"decision why current solution was chosen"* (D12). Useful commit messages can help reviewers to decide whether *"PR is large [and needs to] be broken down into smaller bits"* (D8).

Code quality is another top property that Shopify reviewers look for. PR must follow *"coding guidelines"* (D14), *"clean and in a mergeable (not draft) state (no commented out lines, syntax follows convention, etc.)"*, *"annotated if necessary (why certain things are changed; foreseeing where a reviewer might have questions)"* (D8).

PR complexity is also an important indicator of quality. Developers check whether a PR is *"too large for people to review in one go?"* (D2) and *"can be split into smaller PRs"* (D4). Reviewers want a PR to be *"small and easy to understand"* (D1) and to *"solve only one issue, and make the smallest change possible (doesn't get carried away doing non-targeted changes"* (D8).

Developers also checked PRs for their *revertability*, where PR is *"self-contained"* (D2), and *"can be reverted (in most cases) with no side effects"* (D4). *Conformance to project goals* is seen as an important property of a PR. For example, D13 justify that *"a good PR either adds features as per the goals of the project or rectifies errors and inconsistencies in existing code and documentation"*. Other reviewers look at whether PR's functionality satisfies performance requirements, i.e., it *"satisfies SLA's the code is going to be run under"* (D6).

Several respondents suggested that *testing* is a good indicator of the PR quality. *"Does PR have tests?"* (D6), *"can [PR] be tested on its own?"* (D10), *"how good PR in terms of tests coverage?"* (D14) are some of the questions that developers try to answer when reviewing PRs.

"Good discussions going on" (D4), author experience submitting PRs, and time it takes to review a PR as explained by D5 *"the minimum time I can spend reviewing it, the better is the PR"* are also attributed to the PR quality.

Perception of the PR review quality. We now offer insights related to understanding developer perception of the main characteristics contributing to a PR review quality.

The responses to the relevant Likert-scale question in the survey are shown on Figure 6.3. For the majority of factors we asked developers about, there was no strong prevalence of either positive or negative responses. The only factor that received the overwhelming support support is reviewer experience (94% of positive responses). Many developers are also agree that response time is strong indicator of PR review quality. Surprisingly, only one size-related metric (PR size) received a considerably large number of positive responses (81%). The vast majority of the developers disagreed with the statement that the affiliation of the PR author affects the quality of the review process. This might indicate that developers are confident that the established practices are fair, and do not differentiate based where a contribution came from.

Applying the manual coding analysis to the open-ended question, we found that the majority of survey respondents indicated that *constructive feedback* remains the key attribute of a high quality review. Developers expect from reviewers to 1) maintain *"a good balance between asking questions and being clear about what you, as a reviewer, want changed"* (D2); 2) offer *"actionable comments"* (D2) for PR author so it's easy for them to address these comments; 3) express their feedback in an appropriate manner, *"strong opinions weakly held — a weak opinion is useless"* (D4); 4) take time to *"educate when you point out a mistake"* (D6); 5) ask detailed questions to PR authors to get them *"thinking about what they are trying to accomplish"* (D10). When PR is "rejected", developers want reviewers to offer *"the list of what is missing of the user story/task functionality, coding style, architectural approaches"* (D15).

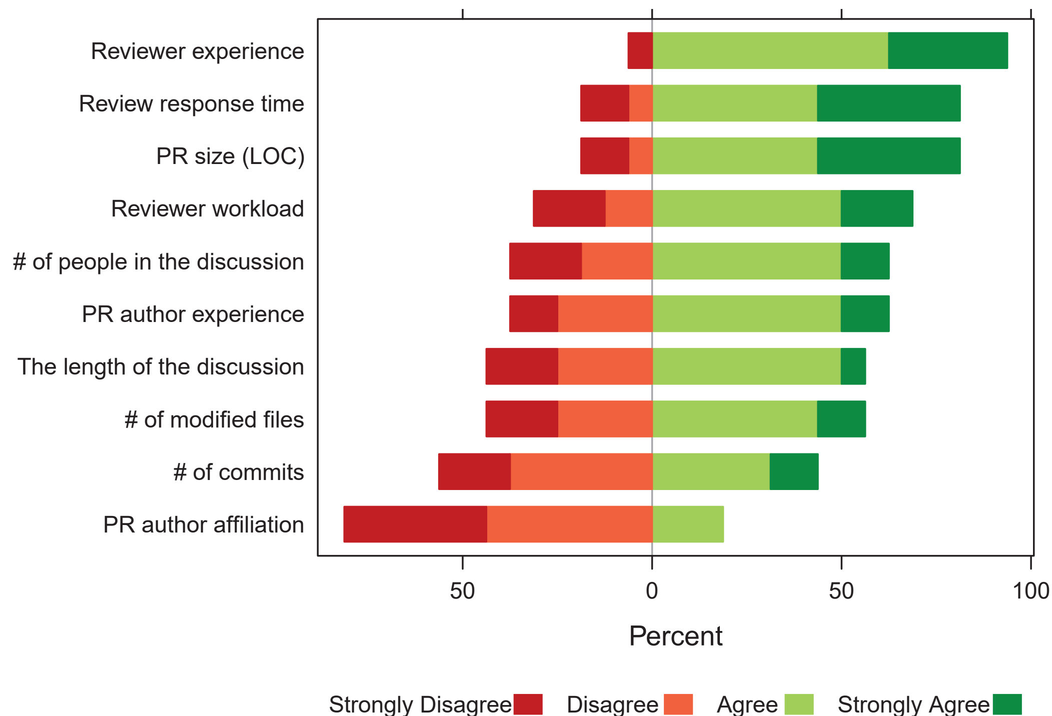

Figure 6.3: Factors influencing PR review quality.

Testing is viewed as a process that helps reviewers to conduct PR reviews. Reviewers typically need to pull a PR into their local repository and test it to make sure it does not cause any issues. Thus, reviewers check for presence of *"automated tests and the quality of the tests"* (D1). Other factors that developers attribute to review quality if *time* taken to review PRs, reviewers are advised to *"take time to read it all"* (D6) and not *"rush through"* (D4).

Conformance to project goals, once again, was also important property. Developers noted that review quality is associated with whether reviewers understand that *"we are all on the same team"*, as D2 suggests *"let's keep things moving, as long as they are moving in the right direction"*. Reviewers are expected to evaluated PRs based on project priority and *"charter"*, as well as how they improve *"maintainability of the project"* (D13).

PR reviews are also assessed by how well developers *understand context/rationale/scope* of the change and whether *"its impact is well acknowledged"* (D7), proper syntax/language (D8), and whether the PR review facilitated a good *discussion*, e.g., such as *"generated from important stakeholders"* (D7).

RQ3: Since Active Merchant uses a pull-based development model, PR reviews are performed to evaluate changes. Developer perception of PR quality is defined by its description, complexity, and revertability. PR review quality is seen as a function of constructive feedback, quality of tophats (tests), generated discussion between reviewer and PR author, and moving the project forward.

6.4 Discussion

Merge types. One of our findings is that developers use squashing more often than any other merge technique. While this approach has the largest median merge time associated with it, developers clearly see benefits in using it. Squashing also leads to some loss of historical information, and hence it has been suggested that it is a "bad" development practice[5]. We argue that further research is needed to better understand the considerations that developers make when deciding how to integrate a pull request. With such an understanding, researchers can better address the shortcomings of current merge techniques, which in turn may improve the quality of future research of git-stored data.

Merge time. Recent work by Gousios et al. [50] has also studied merge time and the factors affecting it. The authors analyzed similar factors to the ones studied in this chapter; however, their findings are somewhat different. Both studies showed that developer experience/reputation impacts merge time. However, in Gousios et al.'s study the project-level metrics were shown to be influential while we did not investigate those factors. Furthermore, the factors that were significant in our model were not significant in their study. This suggests that there is likely no unified model that will work across the disparate landscape of all GitHub projects, and that each project/domain should be studied individually.

Merge decision. The factors that affect the merge decision were also studied by Gousios et al. [50] and Tsay et al. [121]. Our findings align well with those of Tsay et al.: both sets of studies showed similar effects of PR size, discussion (although via different metrics: number of commits vs. number of people involved), and affiliation on decision to merge a PR. Gousios et al.'s study showed the effect of the number of files changed as well as some project-level metrics; in our work, those metrics were either insignificant or not studied. Here, the claim we made for the differences in findings regarding merge time appears to be unjustified. However, it is worth pointing out that both Tsay et al. and we relied on the same statistical model — logistic regression — while Gousios et al. used a

[5]Certainly, software engineering researchers may be expected to be unhappy with the loss of information.

random forest classifier. We wonder to what degree the choice of statistical model affects the results, which in turn might warrant a call for unifying the way in which statistical models are used by software engineering researchers.

6.5 Threats to Validity

Threats to *internal validity* concerns the quality of study design and rigorousness of its execution. In our study, these threats are related to data mining, model construction, as well as survey design and analysis. We extracted data using GitHub's own API; this ensured that we had the most up-to-date dataset at our disposal. To limit the number of inaccurate records, we performed a manual inspection of the merged PRs. In addition, we filtered out the obvious outliers from the dataset. While the choice of factors selected for the models might be a threat, we relied on the metrics previously used by the research community. In designing our survey, we tried to ensure that our questions were clear and easy to answer. We might have introduced some research bias during the analysis of open-ended questions; however, we also tried to minimized any such bias by following a strict protocol described in Section 6.2 and reporting the intercoder reliability scores.

External validity concerns the generalizability of the findings. We focused on a single software project and the developers working on it. While Active Merchant is a successful commercial project and the survey respondents are highly experienced full-time employees, it might not be possible to generalize the findings across all projects hosted on GitHub. At the same time, we do believe that any medium-sized software project with similar structure (i.e., a commercial project that is open to external contributions) is likely to exhibit many similar features of pull-based development. Nevertheless, further research is required to enhance the understanding of pull-based software development model and to create a unified body of empirical knowledge about it. To allow replication of this work, we made anonymized dataset, classification of PRs by merge type, the survey design, and the anonymized survey responses publicly available.

6.6 Conclusions

Pull-based software development is a popular model of modern distributed software development. In this work we provide an in-depth study of a commercial software project that employs the pull-based model. First, we studied PR merge types used by the project. We manually classified merged PRs and found that the most of PRs were merged using

squashing and that PR merge type and PR merge time have a statistically significant relationship. We then built statistical models to investigate the effect of a variety of factors on the PR review time and PR merge decision. We found that PR size, the number of people involved in the discussion of a PR, author experience and his or her affiliation were significant factors in both models. Developers also believe that PR description and complexity, type of change, and responsiveness of an author are influential factors as well. Finally, we surveyed Shopify developers to understand their perception of the PR quality and PR review process. The analysis of the survey responses showed that the developers associate the quality of a PR with the quality of its description, its complexity and revertability, while the quality of the review process is linked to the feedback quality, tests quality, and the discussion among developers. The quantitative findings obtained in this chapter (i.e., the factors affecting PR review time and acceptance) are similar the findings in Chapter 3. The developers perception of PR quality and PR review quality is somewhat similar to the perception of Mozilla developers we studied in Chapter 5.

Chapter 7

Conclusions

In a large, long-lived project, an effective code review process is key to ensuring the long-term quality of the code base. Code review is considered to be one of the most effective QA practices in software development. While it is relatively expensive in terms of time and effort, it delivers benefits of identifying defects in code modifications before they are committed into the project's code base [39]. Reviewers play a vital role in the code review process not only by shaping and evaluating individual contributions but also by ensuring the high quality of the project's master code repository. Moreover, reviewers serve as a human face of a project, they are the ones developers interact with to discuss their contributions.

We view the quality of code review as a complex function of contributors' interests (e.g., the timely acceptance of their contributions) and reviewers' interests (e.g., acceptance of only good contributions). We claim that by mining and analyzing software artifacts and by studying developers' day-to-day experience we can learn the specifics of the quality function of a particular software project, which in turn will help the stakeholders to better understand and view the established code review process from the unified perspective.

In Chapter 3, we looked into the code review process from the point of view that might be associated with the one of contributors to a software project, namely we investigated what factors might affect code review time and the likelihood of patch acceptance. By mining issue tracker systems and applying statistical analysis to the gathered data, we were able to show that the size of a patch, the source code area it changes, submitter's affiliation and experience, as well as reviewer's experience and workload influence the time it takes to review that patch. We also discovered that patch acceptance is affected by many of the same factors as well. These findings presented in this chapter might help both

contributors and project owners — contributors might want to submit smaller patches as well as be more "visible" to the core developers while the owners might want to establish a process for assigning the reviewers to incoming patches based on the author's background and reviewer's current tasks.

In Chapter 4, we applied data mining to a source code repository and an issue tracking system to investigate the quality of contributions that passed the code review process and to explore the relationships between the reviewers' code inspections and a variety of factors that might affect the quality of such inspections. We showed that 54% of reviewed patches contained defects that required future bug fixes. We also built statistical models that captured the effect of different factors on the likelihood of a reviewer missing or introducing a bug during the inspection of a patch. The findings from this chapter identify the problem that the stakeholders might even be aware of. To address such a problem, they can alter the review process to be more vigilant around big patches, ensure that reviewers are not overloaded with tasks, encourage more discussion, and assign more experienced reviewers to specific patches.

In Chapter 5, we turned to developers themselves and asked them about their opinion on the issues we studied in the previous two chapters. We surveyed 88 professional developers (who were also full-time Mozilla employees) to understand their perception of code review quality, what factors contribute to how they evaluate submitted code, and what challenges they face when they perform code review activities. With respect to review time and patch acceptance, the developers' responses somewhat aligned with the findings we obtained in Chapter 3. Surprisingly, developers did not identify "catching bugs" as the main characteristic of high quality review. In fact, for developers, code review quality is more about thorough and timely feedback provided by a reviewer with exceptional knowledge of the codebase.

The studies in the previous chapters were based on open source software systems that used more traditional model of evaluation of contributions. Nowadays, pull-based software development model have become popular among many software projects. For our final study, in Chapter 6, we applied data mining techniques and surveyed the developers of an industrial software project hosted on GitHub. The data mining showed that pull request review time and the likelihood of pull request acceptance affected by the similar factors we found to be influential in Chapter 3. At the same time, the developers we surveyed did not have strong opinions about the effect of any factor on the acceptance of pull requests. This might indicate that there are differences between code review processes in OSS and industrial projects. Surprisingly, the developers identified the same main characteristic of a well-done pull request review — feedback quality — that was identified by the Mozilla developers in Chapter 5.

Overall, the studies presented in this thesis show how we can better understand the established code review processes through data mining of existing software artifacts as well as direct communication with the involved developers. With better understanding of these processes, the stakeholders can specifically target aspects of their review process that would benefit the project. For instance, OSS projects, such as Mozilla, can modify the code review process in way that prioritize contributions from outsiders to encourage them to participate more. Industrial projects, such as Active Merchant, can make the project requirements to new code more clear so the overall code review process will be more impartial.

7.1 Code Review Process Recommendations

We now consider how the findings of this dissertation may provide constructive feedback for developers, reviewers, and managers in the goal of improving code review quality. As we mentioned in Chapter 1, code review quality can be evaluated in many ways using a variety of metrics. Naturally, different stakeholders will care about some of those metrics more than about the others. As such, we structure our recommendations from the perspective of three major groups — developers, reviewers, and managers.

Recommendations for developers:

- Submit patches that are small and localized: fewer lines of code overall, fewer changed files, and less fragmentation ("spread") of a change across the software system. Our studies showed that size metrics affect many aspects of code review — smaller changes are reviewed faster and are accepted more often. Additionally, reviewers indicated that they prefer small and isolated patches. Do not address several different things in a single patch.

- Provide a clear description and rationale for each patch. Reviewers want to see not only what is being change but also why it is being changed. In the descriptions that developers write, reviewers are also looking for the signs that developers thought about other solutions as well as potential problems with the proposed one.

- Write patches that are simple and easy to understand. Complex code is more difficult to maintain, and it is easier for bugs to stay unnoticed in such code. Moreover, reviewers believe that an overly complex patch is a sign that patch writer did poor job solving a problem.

- Write tests for new code. Update the existing tests if the proposed change affects them. Although it might sound redundant, include the results of test execution if the code review tool used by the project does not perform auto testing.
- Familiarize yourself with the project. Make sure that the changes you write follow the same style used in the project, do not violate the established architecture and design, and fit nicely with the existing code base.
- Work with a reviewer as a team. Do not abandon your changes while they are under the review; respond to the reviewer in a timely manner, and address their concerns.

Recommendations for reviewers:

- Provide clear and detailed feedback. It cannot be stressed enough how important feedback quality is — it provides the unique opportunity for knowledge transfer between a reviewer and a developer, it is the only source of information that can help a developer understand what was done wrong and what can be improved. While detailed feedback is important, do not concentrate only on the little things such as style or spelling errors. Developers want reviewers to focus on "bigger picture", highlight weak points, and suggest a way of fixing them. Provide actionable comments, so it would be easier for developers to address them.
- Be mindful of how you communicate with developers, especially your tone. Phrase your feedback constructively; be supportive, patient, and tactful.
- Larger patches require more attention, as we found that they are more likely to have defects go unnoticed.
- Strive to perform reviews promptly; however, do not allow deadline pressures or other stress to affect the review quality. Out studies show that reviewers with higher workloads tend to miss more defects.

Recommendations for managers:

- Ensure that project policies and goals are defined and known to both developers and reviewers. Developers must know what the requirements are for the changes they submit (e.g., code style, use of tests, review process steps, etc.). Reviewers will be able to use those requirements as a baseline in their reviews.

- If possible, setup the code review process in a way that incoming changes are reviewed by multiple reviewers. Our findings, as well as finding of other researchers indicate that the presence of "a second pair of eyes" greatly improves the review quality.

- If your project is open source, check repeatedly that the established review process is impartial — check that there is no much difference in review time and acceptance for patches submitted by "core" team and "external" developers.

- Design and implement an explicit process for assigning incoming changes to reviewers. This process should take into account reviewers' current workload, their experience with code reviews, and their familiarity with code that the proposed change affects.

Most of our recommendations are behavioral in nature, and thus they are likely not to provide immediate results. Instead, they will pay off over time, and in our opinion, should be viewed as a long-term investment in the quality of the code review process.

7.2 Future Work

Based on the findings presented in this thesis, we identified several possible directions for future research.

- **Understanding the effect of testing on code review quality.**
  Software testing is undoubtedly an important technique that aims to improve the quality of software. The majority of software projects use testing, although they differ in test coverage, test quality, and the policies around writing and maintaining of the tests. From our surveys with the developers (Chapter 5 and Chapter 6), we learned that they view tests as an important and integral piece of the code review process. While a lot of research has been done on the effect of testing and code review on software quality, there are no studies on the effect of testing on code review quality. Does testing help reviewers? Do they overly rely on the presence and/or results of tests in their reviews? Can we decouple testing from the code review process? These are some questions that might be worth investigating.

- **Investigating the bugs caught during the review.**
  In Chapter 4, we studied the factors that might affect reviewers' ability to identify defects during the review. It would be interesting to explicitly study these bugs in more detail: How numerous are they? Are they demonstrably different from other

bugs (e.g., criticality, files affected, time to fix once identified)? Our intuition is that if reviewers are swamped with non-critical defects (e.g., spelling, indentation, etc.), they might deal with these issues first and after that feel as if they have done a proper inspection while critical defects are left unnoticed. If this is indeed the case, we need to study whether identification of such non-critical defects can be automated (for example, through the use of static analysis tools) so the developers could focus on more important questions regarding the code in front of them.

- **Designing a next-generation code review tool.**
  Both open source and industrial developers stated that the quality and thoroughness of the feedback is the main characteristic of a well-done code review. To "make" developers leave thorough feedback, we need to find ways to motivate them to do so. This might be achieved through the gamification (the process of adding game elements to non-game processes) of the code review process. Introducing elements like votes for patches and reviews, badges, "thank you's" for the feedback might add some "healthy competitiveness" to the review process which in turn may increase its quality.

7.3 Summary of Contributions

Here we highlight the main contributions presented in this thesis:

- We identified three sets of factors — technical, personal, and organizational — that influence the duration of code review as well as the outcome of that review in two big OSS projects [15].

- We showed that a large percentage of code changes that successfully passed the review process still contain defects [71].

- We investigated which aspects contribute to poor code review quality and found that size of a patch, the number of affected files, the presence of a second reviewer, reviewer workload and experience, as well as developers participation in the discussion of a patch are the factors that affect the effectiveness of code review [71].

- We surveyed professional Mozilla developers and found that they believe that factors such as the experience of developers, the choice of a reviewer, size of a patch, its quality and rationale affect the time needed for review; while bug severity, code

quality and its rationale, presence and quality of tests, and developer personality impact review decisions [70].

- We explored how developers perceive the quality of the code review process, and the problems developers face during code review and found that developer perception of code review quality is shaped by their experience and defined as a function of clear and thorough feedback provided in a timely manner by a peer with a supreme knowledge of the code base, strong personal and inter-personal qualities [70].
- We investigated the factors that affect the timeliness and outcome of pull request reviews and found that a pull request size, the discussion, as well as author experience and affiliation influence both review time and review decision [72].
- We explored how developers perceive pull request quality and found that their perception is defined by pull request description, complexity, and revertability. Pull request review quality is seen as a function of constructive feedback, quality of tests, and generated discussion between author and reviewer [72].

Overall, our work presents an in-depth analysis of code review quality in a variety of settings — open source software and industrial projects, traditional and pull-based distributed development models. We observed similar patterns in the execution of code review that the stakeholder should be aware of to maintain the long-term health of the projects. Our findings are based on the evidence found in the studied software systems as well as on the communication with the developers and reviewers involved in those systems.

References

[1] Steve Adolph, Wendy Hall, and Philippe Kruchten. A methodological leg to stand on: lessons learned using grounded theory to study software development. In *Proceedings of the 2008 Conference of the Center for Advanced Studies on Collaborative Research: Meeting of Minds*, page 13. ACM, 2008.

[2] Steve Adolph, Wendy Hall, and Philippe Kruchten. Using grounded theory to study the experience of software development. *Empirical Software Engineering*, 16(4):487–513, 2011.

[3] K.S. Amant and P. Zemliansky. *Internet-based Workplace Communications: Industry & Academic Applications*. Information Science Pub., 2005.

[4] J. Anvik and G.C. Murphy. Determining implementation expertise from bug reports. In *Proceedings of the 4th International Workshop Mining Software Repositories*, pages 2–2, May 2007.

[5] John Anvik. Automating bug report assignment. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 937–940, New York, NY, USA, 2006. ACM.

[6] John Anvik, Lyndon Hiew, and Gail C. Murphy. Who should fix this bug? In *Proceedings of the 28th International Conference on Software Engineering*, pages 361–370, 2006.

[7] Lerina Aversano, Luigi Cerulo, and Concettina Del Grosso. Learning from bug-introducing changes to prevent fault prone code. In *Proceedings of the 9th International Workshop on Principles of Software Evolution*, pages 19–26. ACM, 2007.

[8] Alberto Bacchelli and Christian Bird. Expectations, outcomes, and challenges of modern code review. In *Proceedings of the 35th International Conference on Software Engineering*, ICSE '13, pages 712–721. IEEE Press, 2013.

[9] Gabriele Bavota, Bernardino De Carluccio, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Orazio Strollo. When does a refactoring induce bugs? an empirical study. In *Proceedings of the 12th International Working Conference on Source Code Analysis and Manipulation*, SCAM '12, pages 104–113. IEEE Computer Society, 2012.

[10] Olga Baysal, Michael W. Godfrey, and Robin Cohen. A bug you like: A framework for automated assignment of bugs. In *Proceedings of the 17th International Conference on Program Comprehension*, pages 297–298. IEEE, May 2009.

[11] Olga Baysal and Reid Holmes. A Qualitative Study of Mozilla's Process Management Practices. Technical Report CS-2012-10, David R. Cheriton School of Computer Science, University of Waterloo, Waterloo, Canada, June 2012.

[12] Olga Baysal, Reid Holmes, and Michael W. Godfrey. No issue left behind: Reducing information overload in issue tracking. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 666–677, New York, NY, USA, 2014. ACM.

[13] Olga Baysal, Oleksii Kononenko, Reid Holmes, and Michael W Godfrey. The secret life of patches: A firefox case study. In *Proceedings of the 19th Working Conference on Reverse Engineering*, pages 447–455. IEEE, 2012.

[14] Olga Baysal, Oleksii Kononenko, Reid Holmes, and Michael W. Godfrey. The influence of non-technical factors on code review. In *Proceedings of the 20th Working Conference on Reverse Engineering*, pages 122–131. IEEE, 2013.

[15] Olga Baysal, Oleksii Kononenko, Reid Holmes, and Michael W. Godfrey. Investigating technical and non-technical factors influencing modern code review. *Empirical Software Engineering*, 21(3):932–959, 2016.

[16] Andrew Begel and Thomas Zimmermann. Analyze this! 145 questions for data scientists in software engineering. In *Proceedings of the 36th International Conference on Software Engineering*, pages 12–23, 2014.

[17] Moritz Beller, Alberto Bacchelli, Andy Zaidman, and Elmar Juergens. Modern code reviews in open-source projects: Which problems do they fix? In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 202–211, New York, NY, USA, 2014. ACM.

[18] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. What makes a good bug report? In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 308–318. ACM, 2008.

[19] Christian Bird, Alex Gourley, Prem Devanbu, Anand Swaminathan, and Greta Hsu. Open borders? immigration in open source projects. In *Proceedings of the 4th International Workshop on Mining Software Repositories*, page 6. IEEE Computer Society, 2007.

[20] Christian Bird, Nachiappan Nagappan, Brendan Murphy, Harald Gall, and Premkumar Devanbu. Don't touch my code!: Examining the effects of ownership on software quality. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 4–14, New York, NY, USA, 2011. ACM.

[21] Christopher M Bishop et al. *Pattern recognition and machine learning*, volume 4. Springer New York, 2006.

[22] Gargi Bougie, Christoph Treude, Daniel M German, and M Storey. A comparative exploration of freebsd bug lifetimes. In *Proceedings of the 7th Working Conference on Mining Software Repositories*, pages 106–109. IEEE, 2010.

[23] Gerardo Canfora and Luigi Cerulo. How software repositories can help in resolving a new change request. In *Proceedings of the the Workshop on Empirical Studies in Reverse Engineering*, 2005.

[24] Marcelo Cataldo, Audris Mockus, Jeffrey A Roberts, and James D Herbsleb. Software dependencies, work dependencies, and their impact on failures. *IEEE Transactions on Software Engineering*, 35(6):864–878, 2009.

[25] Rune Haubo B Christensen and Merete K Hansen. *binomTools: Performing diagnostics on binomial regression models*, 2011. R package version 1.0-1.

[26] J. Cohen. *Best Kept Secrets of Peer Code Review*. Smart Bear Inc., Austin, TX, USA, 2006.

[27] J. Cohen, P. Cohen, S.G. West, and L.S. Aiken. *Applied Multiple Regression/Correlation Analysis for the Behavioral Sciences*. L. Erlbaum Associates, Inc., 2003.

[28] Gerry Coleman and Rory OConnor. Using grounded theory to understand software process improvement: A study of irish software product companies. *Information and Software Technology*, 49(6):654–667, 2007.

[29] M. Conway. How do committees invent? *Datamation Journal*, pages 28–31, April 1968.

[30] Juliet Corbin and Anselm Strauss. *Basics of qualitative research: Techniques and procedures for developing grounded theory*. Sage publications, 2014.

[31] John W Creswell. *Research design: Qualitative, quantitative, and mixed methods approaches*. Sage publications, 2013.

[32] Davor Cubranic. Automatic bug triage using text categorization. In *Proceedings of the 16th International Conference on Software Engineering & Knowledge Engineering*, pages 92–97. KSI Press, 2004.

[33] Barthélémy Dagenais, Harold Ossher, Rachel KE Bellamy, Martin P Robillard, and Jacqueline P De Vries. Moving into a new software project landscape. In *Proceedings of the 32nd International Conference on Software Engineering*, pages 275–284. ACM, 2010.

[34] Cleidson RB de Souza and David F Redmiles. An empirical study of software developers' management of dependencies and changes. In *Proceedings of the 30th International Conference on Software Engineering*, pages 241–250. ACM, 2008.

[35] E. P. Doolan. Experience with fagan's inspection method. *Software Practice & Experience*, 22(2):173–182, February 1992.

[36] Tore Dybå, Vigdis By Kampenes, and Dag IK Sjøberg. A systematic review of statistical power in software engineering experiments. *Information and Software Technology*, 48(8):745–755, 2006.

[37] Raymond E. *The cathedral and the bazaar. Musings on Linux and Open Source by an accidental revolutionary.* O'Reilly & Associates, Cambridge, 1999.

[38] Jon Eyolfson, Lin Tan, and Patrick Lam. Do time of day and developer experience affect commit bugginess? In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, pages 153–162. ACM, 2011.

[39] Michael Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211, 1976.

[40] Michael Fischer, Martin Pinzger, and Harald Gall. Populating a release history database from version control and bug tracking systems. In *Proceedings of the International Conference on Software Maintenance*, pages 23–32, Amsterdam Netherlands, September 2003. IEEE Computer Society Press.

[41] J. Fox. *Applied Regression Analysis, Linear Models, and Related Methods.* SAGE Publications, 1997.

[42] J. Fox. *Applied Regression Analysis and Generalized Linear Models.* SAGE Publications, 2008.

[43] John Fox and Sanford Weisberg. *An R Companion to Applied Regression.* Sage, Thousand Oaks CA, second edition, 2011.

[44] Deen Freelon. ReCal2: Reliability for 2 coders. `http://dfreelon.org/utils/recalfront/recal2/`.

[45] Harald Gall, Mehdi Jazayeri, and Jacek Krajewski. Cvs release history data for detecting logical couplings. In *Proceedings of the 6th International Workshop on Principles of Software Evolution*, pages 13–23, Washington DC USA, 2003. IEEE Computer Society.

[46] GitHub. GitHub API Client Library. `https://github.com/octokit/octokit.net`.

[47] GitHub. GitHub API v3. `https://developer.github.com/v3/`.

[48] GitHub. GitHub Octoverse. `https://octoverse.github.com/`.

[49] Barney G Glaser and Anselm L Strauss. *The discovery of grounded theory: Strategies for qualitative research.* Transaction Publishers, 2009.

[50] Georgios Gousios, Martin Pinzger, and Arie van Deursen. An exploratory study of the pull-based software development model. In *Proceedings of the 36th International Conference on Software Engineering*, pages 345–355. ACM, 2014.

[51] Georgios Gousios, Margaret-Anne Storey, and Alberto Bacchelli. Work practices and challenges in pull-based development: The contributor's perspective. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 285–296, New York, NY, USA, 2016. ACM.

[52] Georgios Gousios, Andy Zaidman, Margaret-Anne Storey, and Arie van Deursen. Work practices and challenges in pull-based development: The integrator's perspective. In *Proceedings of the 37th International Conference on Software Engineering*, pages 358–368, 2015.

[53] Todd L. Graves, Alan F. Karr, J. S. Marron, and Harvey Siy. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*, 26(7):653–661, July 2000.

[54] R.M. Groves, F.J. Fowler, M.P. Couper, J.M. Lepkowski, E. Singer, and R. Tourangeau. *Survey Methodology*. Wiley, 2 edition, 2009.

[55] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The elements of statistical learning: data mining, inference and prediction*. Springer, 2 edition, 2009.

[56] Les Hatton. Testing the value of checklists in code inspections. *IEEE Software*, 25(4):82–88, 2008.

[57] Ottar Hellevik. Linear versus logistic regression when the dependent variable is a dichotomy. *Quality & Quantity*, 43(1):59–74, 2009.

[58] Israel Herraiz, Daniel M. German, Jesus M. Gonzalez-Barahona, and Gregorio Robles. Towards a simplification of the bug report form in eclipse. In *Proceedings of the 5th Working Conference on Mining Software Repositories*, pages 145–148, 2008.

[59] Rashina Hoda, James Noble, and Stuart Marshall. Using grounded theory to study the human aspects of software engineering. In *Human Aspects of Software Engineering*, page 5. ACM, 2010.

[60] Hadi Hosseini, Raymond Nguyen, and Michael W Godfrey. A market-based bug allocation mechanism using predictive bug lifetimes. In *Proceedings of the 16th European Conference on Software Maintenance and Reengineering*, pages 149–158. IEEE, 2012.

[61] Yue Jiang, Bojan Cuki, Tim Menzies, and Nick Bartlow. Comparing design and code metrics for software quality prediction. In *Proceedings of the 4th International Workshop on Predictor Models in Software Engineering*, PROMISE '08, pages 11–18. ACM, 2008.

[62] Yujuan Jiang, Bram Adams, and Daniel M. German. Will my patch make it? and how fast?: Case study on the linux kernel. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 101–110. IEEE Press, 2013.

[63] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. The promises and perils of mining github. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 92–101. ACM, 2014.

[64] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E. Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering*, 39(6):757–773, 2013.

[65] Chris F. Kemerer and Mark C. Paulk. The impact of design and code reviews on software quality: An empirical study based on psp data. *IEEE Transactions on Software Engineering*, 35(4):534–550, July 2009.

[66] Sunghun Kim, E. James Whitehead, Jr., and Yi Zhang. Classifying software changes: Clean or buggy? *IEEE Transactions on Software Engineering*, 34(2):181–196, March 2008.

[67] Sunghun Kim, Thomas Zimmermann, Kai Pan, and E. James Jr. Whitehead. Automatic identification of bug-introducing changes. In *Proceedings of the 21st IEEE International Conference on Automated Software Engineering*, pages 81–90. IEEE Computer Society, 2006.

[68] Andrew J. Ko, Brad A. Myers, and Duen Horng Chau. A linguistic analysis of how people describe software problems. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 127–134, 2006.

[69] Oleksii Kononenko and Olga Baysal. A Qualitative Exploratory Study of How OSS Developers Define Code Review Quality. Technical Report CS-2015-14, University of Waterloo, Waterloo, Canada, August 2015.

[70] Oleksii Kononenko, Olga Baysal, and Michael W Godfrey. Code review quality: how developers see it. In *Proceedings of the 38th International Conference on Software Engineering*, pages 1028–1038. ACM, 2016.

[71] Oleksii Kononenko, Olga Baysal, Latifa Guerrouj, Yaxin Cao, and Michael 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. IEEE, 2015.

[72] Oleksii Kononenko, Tresa Rose, Olga Baysal, and Michael W Godfrey. Studying pull request merges: A case study of shopify's active merchant. Under review.

[73] William H. Kruskal and W. Allen Wallis. Use of ranks in one-criterion variance analysis. *Journal of the American Statistical Association*, 47(260):pp. 583–621, 1952.

[74] E.L. Lehmann and H.J.M. D'Abrera. *Nonparametrics: statistical methods based on ranks*. Springer, 2006.

[75] Mika V. Mantyla and Casper Lassenius. What types of defects are really discovered in code reviews? *IEEE Transactions on Software Engineering*, 35(3):430–448, May 2009.

[76] Jennifer Marlow, Laura Dabbish, and Jim Herbsleb. Impression formation in online peer production: Activity traces and personal profiles in github. In *Proceedings of the 2013 Conference on Computer Supported Cooperative Work*, CSCW '13, pages 117–128. ACM, 2013.

[77] Patricia Yancey Martin and Barry A Turner. Grounded theory and organizational research. *The Journal of Applied Behavioral Science*, 22(2):141–157, 1986.

[78] Frank Jr. Massey. The kolmogorov-smirnov test for goodness of fit. *Journal of the American Statistical Association*, 46(253):pp. 68–78, 1951.

[79] Shinsuke Matsumoto, Yasutaka Kamei, Akito Monden, Ken-ichi Matsumoto, and Masahide Nakamura. An analysis of developer metrics for fault prediction. In *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, PROMISE '10, pages 18:1–18:9. ACM, 2010.

[80] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 192–201. ACM, 2014.

[81] Shane Mcintosh, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering*, 21(5):2146–2189, October 2016.

[82] Thilo Mende and Rainer Koschke. Effort-aware defect prediction models. In *Proceedings of the 14th European Conference on Software Maintenance and Reengineering*, CSMR '10, pages 107–116. IEEE Computer Society, 2010.

[83] Matthew B. Miles and A. Michael Huberman. *Qualitative data analysis: An expanded sourcebook*. Sage, 1994.

[84] A. Mockus and J.D. Herbsleb. Expertise browser: a quantitative approach to identifying expertise. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 503–512, May 2002.

[85] Audris Mockus. Organizational volatility and its effects on software defects. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 117–126. ACM, 2010.

[86] Audris Mockus, Roy T. Fielding, and James D. Herbsleb. Two case studies of open source software development: Apache and mozilla. *ACM Transactions on Software Engineering and Methodology*, 11(3):309–346, 2002.

[87] Audris Mockus and Lawrence Votta. Identifying reasons for software change using historic databases. In *Proceedings of the 16th International Conference on Software Maintenance*, pages 120–130, October 2000.

[88] Mariano Angel Montoni and Ana Regina Rocha. Applying grounded theory to understand software process improvement implementation. In *Proceedings of the 7th International Conference on the Quality of Information and Communications Technology*, pages 25–34. IEEE Computer Society, 2010.

[89] Rodrigo Morales, Shane McIntosh, and Foutse Khomh. Do code review practices impact design quality? a case study of the qt, vtk, and itk projects. In *Proceedings of the 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering*, pages 171–180. IEEE, 2015.

[90] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 181–190. ACM, 2008.

[91] Mozilla. BMO/ElasticSearch. `https://wiki.mozilla.org/BMO/ElasticSearch`.

[92] Mozilla. Code Review FAQ. `https://developer.mozilla.org/en/Code_Review_FAQ`, February 2015.

[93] Mozilla. Code-Review Policy. `http://www.mozilla.org/hacking/reviewers.html`, February 2015.

[94] MozillaWiki. Code Review. `https://wiki.mozilla.org/Firefox/Code_Review`, February 2015.

[95] MozillaWiki. Modules. https://wiki.mozilla.org/Modules, February 2015.

[96] Nachiappan Nagappan, Brendan Murphy, and Victor Basili. The influence of organizational structure on software quality: An empirical case study. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 521–530. ACM, 2008.

[97] Mary Natrella. *NIST/SEMATECH e-Handbook of Statistical Methods*. NIST/SEMATECH, July 2010.

[98] Lucas D Panjer. Predicting eclipse bug lifetimes. In *Proceedings of the 4th International Workshop on Mining Software Repositories*, page 29. IEEE Computer Society, 2007.

[99] Weka Machine Learning Project. Weka. http://www.cs.waikato.ac.nz/ml/weka/.

[100] Emil Protalinski. Opera confirms it will follow google and ditch webkit for blink, as part of its commitment to chromium. https://goo.gl/MyiEBu, April 2013.

[101] Foyzur Rahman and Premkumar Devanbu. Ownership, experience and defects: A fine-grained study of authorship. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 491–500, New York, NY, USA, 2011. ACM.

[102] Foyzur Rahman and Premkumar Devanbu. How, and why, process metrics are better. In *Proceedings of the 35th International Conference on Software Engineering*, ICSE '13, pages 432–441, Piscataway, NJ, USA, 2013. IEEE Press.

[103] Peter C. Rigby and Christian Bird. Convergent contemporary software peer review practices. In *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering*, pages 202–212, 2013.

[104] Peter C. Rigby and Daniel M. German. A preliminary examination of code review processes in open source projects. Technical Report DCS-305-IR, University of Victoria, January 2006.

[105] Peter C Rigby, Daniel M German, and Margaret-Anne Storey. Open source software peer review practices: A case study of the apache server. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 541–550. ACM, ACM, 2008.

[106] Peter C. Rigby and Margaret-Anne Storey. Understanding broadcast based peer review on open source software projects. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 541–550. ACM, 2011.

[107] Jarrett Rosenberg. Statistical methods and measurement. In Forrest Shull, Janice Singer, and DagI.K. Sjberg, editors, *Guide to Advanced Empirical Software Engineering*, pages 155–184. Springer London, 2008.

[108] Glen W. Russell. Experience with inspection in ultralarge-scale development. *IEEE software*, 8(1):25–31, 1991.

[109] Carolyn Seaman and Forrest Shull. Inspecting the history of inspections: An example of evidence-based technology diffusion. *IEEE Software*, 25:88–90, 2008.

[110] Emad Shihab, Zhen Ming Jiang, Walid M Ibrahim, Bram Adams, and Ahmed E Hassan. Understanding the impact of code and process metrics on post-release defects: a case study on the eclipse project. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, page 4. ACM, 2010.

[111] Shopify. Active Merchant. `https://github.com/activemerchant/active_merchant/`.

[112] Shopify. Shopify Announces Fourth-Quarter and Full Year 2016 Financial Results. `https://investors.shopify.com/Investor-News-Details/2017/Shopify-Announces-Fourth-Quarter-and-Full-Year-2016-Financial-Results/default.aspx`.

[113] Leif Singer, Fernando Figueira Filho, and Margaret-Anne Storey. Software engineering at the speed of light: How developers stay current using twitter. In *Proceedings of the 36th International Conference on Software Engineering*, pages 211–221. ACM, 2014.

[114] R. Mark Sirkin. *Statistics for the social sciences*. Sage Publications, 1995.

[115] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '05, pages 1–5. ACM, 2005.

[116] Anselm Strauss and Juliet M Corbin. *Basics of qualitative research: Grounded theory procedures and techniques.* Sage Publications, Inc, 1990.

[117] Jerome D Thayer. Using multiple regression with dichotomous dependent variables. 1986.

[118] 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 Working Conference on Mining Software Repositories*, pages 168–179, May 2015.

[119] Patanamon Thongtanunam, Shane McIntosh, Ahmed E. Hassan, and Hajimu Iida. Review participation in modern code review. *Empirical Software Engineering*, 22(2):768–817, 2017.

[120] Tue Tjur. Coefficients of determination in logistic regression modelsa new proposal: The coefficient of discrimination. *The American Statistician*, 63(4):366–372, 2009.

[121] Jason Tsay, Laura Dabbish, and James Herbsleb. Influence of social and technical factors for evaluating contribution in github. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 356–366, New York, NY, USA, 2014. ACM.

[122] Davor Čubranić and Gail C. Murphy. Hipikat: Recommending pertinent software development artifacts. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 408–418, Washington, DC, USA, 2003. IEEE Computer Society.

[123] C. Weiss, R. Premraj, T. Zimmermann, and A. Zeller. How long will it take to fix this bug? In *Proceedings of the 4th International Workshop on Mining Software Repositories*, May 2007.

[124] Peter Weissgerber, Daniel Neu, and Stephan Diehl. Small patches get in! In *Proceedings of the 5th Working Conference on Mining Software Repositories*, pages 67–76, 2008.

[125] Mozilla Wiki. MozReview. `https://wiki.mozilla.org/Auto-tools/Projects/MozReview`, August 2015.

[126] Chadd Williams and Jaime Spacco. Szz revisited: Verifying when changes induce fixes. In *Proceedings of the 2008 Workshop on Defects in Large Software Systems*, DEFECTS '08, pages 32–36, New York, NY, USA, 2008. ACM.

[127] Chadd C. Williams and Jaime W. Spacco. Branching and merging in the repository. In *Proceedings of the 5th Working Conference on Mining Software Repositories*, MSR '08, pages 19–22, New York, NY, USA, 2008. ACM.

[128] Ian H. Witten, Eibe Frank, and Mark A. Hall. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2011.

[129] Jifeng Xuan, He Jiang, Zhilei Ren, Jun Yan, and Zhongxuan Luo. Automatic bug triage using semi-supervised text classification. In *Proceedings of the 22nd International Conference on Software Engineering & Knowledge Engineering*, pages 209–214, 2010.

[130] Shahed Zaman, Bram Adams, and Ahmed E. Hassan. Security versus performance bugs: a case study on Firefox. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, pages 93–102, 2011.

[131] Thomas Zimmermann, Nachiappan Nagappan, Harald Gall, Emanuel Giger, and Brendan Murphy. Cross-project defect prediction: A large scale experiment on data vs. domain vs. process. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 91–100. ACM, 2009.

APPENDICES

Appendix A

Survey of Mozilla Developers

1. How would you describe your role on the project(s)?

   ☐ Software Developer/Engineer
   ☐ Project Manager/Lead
   ☐ QA/Testing Engineer
   ☐ Other: ______________

2. How many years of experience do you have in software development?

   ○ < 1
   ○ 1 to 2
   ○ 3 to 6
   ○ 7 to 10
   ○ 10+

3. You work for:

   ○ Mozilla
   ○ Other: ______________

4. How are you involved in code review?

☐ Writing patches

☐ Reviewing patches

☐ Discussing patches/bugs

☐ Other: ______________

5. On average, how many patches do you submit for a review every week?

   ○ < 5

   ○ 6 to 10

   ○ 11 to 20

   ○ 21+

   ○ I do not submit

6. How long have you been reviewing patches?

   ○ less than 6 months

   ○ 6 to 12 months

   ○ 1 to 2 years

   ○ 3 to 4 years

   ○ 5+ years

   ○ I do not review

7. On average, how many patches do you review every week?

   ○ < 5

   ○ 6 to 10

   ○ 11 to 20

   ○ 21+

   ○ I do not review

8. In what environment do you typically conduct code review?

   ○ Issue tracking (e.g., Bugzilla)

○ Copy a patch locally into editor/IDE

○ Other: _______________

9. Where do you discuss patches?

   ☐ Issue tracking

   ☐ Email

   ☐ IRC

   ☐ Skype/Hangouts

   ☐ Face-to-face discussions

   ☐ Other: _______________

10. The following factors influence code review DECISIONS (i.e., whether you accept or reject the patch itself after having reviewed it):

	Strongly disagree	Disagree	Neither agree nor disagree	Agree	Strongly agree
Patch size (LOC)	○	○	○	○	○
Code chunks	○	○	○	○	○
Number of modified files	○	○	○	○	○
Module	○	○	○	○	○
Priority of a bug	○	○	○	○	○
Severity of a bug	○	○	○	○	○
Number of previous patches (resubmits)	○	○	○	○	○
Review queue (aka load)	○	○	○	○	○
Reviewer experience	○	○	○	○	○
Patch writer experience	○	○	○	○	○
Number of people involved in the discussion of a patch	○	○	○	○	○
The length of the discussion of a patch	○	○	○	○	○

11. In your opinion, what other factors affect code review DECISIONS?

____

____

____

12. The following factors influence code review TIME (duration):

	Strongly disagree	Disagree	Neither agree nor disagree	Agree	Strongly agree
Patch size (LOC)	○	○	○	○	○
Code chunks	○	○	○	○	○
Number of modified files	○	○	○	○	○
Module	○	○	○	○	○
Priority of a bug	○	○	○	○	○
Severity of a bug	○	○	○	○	○
Number of previous patches (resubmits)	○	○	○	○	○
Review queue (aka load)	○	○	○	○	○
Reviewer experience	○	○	○	○	○
Patch writer experience	○	○	○	○	○
Number of people involved in the discussion of a patch	○	○	○	○	○
The length of the discussion of a patch	○	○	○	○	○

13. In your opinion, what other factors affect code review TIME?

____

____

____

14. How do you assess the quality of a patch?

____

____

____

15. In your opinion, what characteristics do contribute to a well-done code review?

    ____

    ____

    ____

16. The following factors influence code review QUALITY (e.g., the likelihood of detecting problems with a patch):

	Strongly disagree	Disagree	Neither agree nor disagree	Agree	Strongly agree
Patch size (LOC)	○	○	○	○	○
Code chunks	○	○	○	○	○
Number of modified files	○	○	○	○	○
Module	○	○	○	○	○
Priority of a bug	○	○	○	○	○
Severity of a bug	○	○	○	○	○
Number of previous patches (resubmits)	○	○	○	○	○
Review queue (aka load)	○	○	○	○	○
Reviewer experience	○	○	○	○	○
Patch writer experience	○	○	○	○	○
Number of people involved in the discussion of a patch	○	○	○	○	○
The length of the discussion of a patch	○	○	○	○	○
Review response time	○	○	○	○	○

17. In your opinion, what other factors affect code review QUALITY?

____

____

____

18. What is your biggest challenge in performing code review tasks?

    ____

    ____

    ____

19. What tools would you like to have to assist you with code review activities?

    ____

    ____

    ____

Appendix B

Survey of Shopify Developers

1. How would you describe your role on the project(s)?

   ☐ Software Developer/Engineer
   ☐ Project Manager/Lead
   ☐ QA/Testing Engineer
   ☐ Other: _______________

2. How many years of experience do you have in software development?

   ○ < 1
   ○ 1 to 2
   ○ 3 to 6
   ○ 7 to 10
   ○ 10+

3. On average, how many pull requests do you submit every week?

   ○ 1 to 2 pull requests
   ○ 3 to 4 pull requests
   ○ 5 to 10 pull requests
   ○ 10 to 20 pull requests

- ○ 20+ pull requests
- ○ I do not submit

4. On average, how many pull requests do you review every week?

   - ○ 1 to 2 pull requests
   - ○ 3 to 4 pull requests
   - ○ 5 to 10 pull requests
   - ○ 10 to 20 pull requests
   - ○ 20+ pull requests
   - ○ I do not review

5. How long have you been involved in pull request reviews?

   - ○ less than 6 months
   - ○ 6 to 12 months
   - ○ 1 to 2 years
   - ○ 3 to 4 years
   - ○ 5+ years
   - ○ I do not review

6. As a developer, what types of merges do you do?

   - ☐ GitHub merge (using GitHub UI)
   - ☐ Cherrypick merge (e.g., git cherrypick)
   - ☐ Squash merge
   - ☐ Other: _______________

7. What type of merge do you use most often?

   - ○ GitHub merge (using GitHub UI)
   - ○ Cherrypick merge (e.g., git cherrypick)
   - ○ Squash merge

○ Other: _______________

8. Where do you discuss pull requests?

   ☐ GitHub
   ☐ Email
   ☐ VoIP/online chat (e.g., Skype, Hangout)
   ☐ Face to face discussions
   ☐ Other: _______________

9. Please, briefly explain the steps you typically follow when you are asked to review a pull request.

   ____
   ____
   ____

10. In your opinion, which of the following factors affect TIME of pull request review?

	Strongly disagree	Disagree	Agree	Strongly agree
Pull request size (LOC)	○	○	○	○
Pull request size (# of commits)	○	○	○	○
File count	○	○	○	○
Pull request author experience	○	○	○	○
Pull request reviewer experience	○	○	○	○
Length of the discussion	○	○	○	○
Number of people involved in the discussion	○	○	○	○
Reviewer work load	○	○	○	○
Affiliation of a pull request author (e.g., Shopify dev, Spreedly dev, etc.)	○	○	○	○

11. In your opinion, what are the other factors that may influence TIME for pull request review?

    ____
    ____
    ____

12. Which of the following factors do you think affect pull request review DECISIONS (i.e., whether you merge/not merge the pull request)?

	Strongly disagree	Disagree	Agree	Strongly agree
Pull request size (LOC)	○	○	○	○
Pull request size (# of commits)	○	○	○	○
File count	○	○	○	○
Pull request author experience	○	○	○	○
Pull request reviewer experience	○	○	○	○
Length of the discussion	○	○	○	○
Number of people involved in the discussion	○	○	○	○
Reviewer work load	○	○	○	○
Affiliation of a pull request author (e.g., Shopify dev, Spreedly dev, etc.)	○	○	○	○

13. In your opinion, what are the other factors that may influence DECISION of pull request review?

    ____
    ____
    ____

14. Which of the following factors do you think affect the QUALITY of pull request review (based on your definition of quality)?

	Strongly disagree	Disagree	Agree	Strongly agree
Pull request size (LOC)	○	○	○	○
Pull request size (# of commits)	○	○	○	○
File count	○	○	○	○
Pull request author experience	○	○	○	○
Pull request reviewer experience	○	○	○	○
Length of the discussion	○	○	○	○
Number of people involved in the discussion	○	○	○	○
Reviewer work load	○	○	○	○
Affiliation of a pull request author (e.g., Shopify dev, Spreedly dev, etc.)	○	○	○	○
Reviewer time	○	○	○	○

15. How do you evaluate the quality of a pull request?

____
____
____

16. How would you define a “pull request review of high quality”? And what factors do you think affect review quality?

____
____
____