Université de Montréal

Investigating the Impact of Personal, Temporal and Participation Factors on Code Review Quality

par
Yaxin Cao

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Mémoire présenté à la Faculté des études supérieures
en vue de l'obtention du grade de Maître ès sciences (M.Sc.)
en Computer Science

April, 2015

Université de Montréal
Faculté des études supérieures

Ce mémoire intitulé:

Investigating the Impact of Personal, Temporal and Participation Factors on Code Review Quality

présenté par:

Yaxin Cao

a été évalué par un jury composé des personnes suivantes:

Eugene Syriani, président-rapporteur
Olga Baysal, directeur de recherche

Mémoire accepté le: . . . . . .

RÉSUMÉ

La *révision du code* est un procédé essentiel quelque soit la maturité d'un projet ; elle cherche à évaluer la contribution apportée par le code soumis par les développeurs. En principe, la *révision du code* améliore la qualité des changements de code (*patches*) avant qu'ils ne soient validés dans le repertoire maître du projet. En pratique, l'exécution de ce procédé n'exclu pas la possibilité que certains bugs passent inaperçus.

Dans ce document, nous présentons une étude empirique enquétant la révision du code d'un grand projet open source. Nous investissons les relations entre les inspections des *reviewers* et les facteurs, sur les plans personnel et temporel, qui pourraient affecter la qualité de telles inspections.

Premiérement, nous relatons une étude quantitative dans laquelle nous utilisons l'algorithme SSZ pour détecter les modifications et les changements de code favorisant la création de bogues (bug-inducing changes) que nous avons lié avec l'information contenue dans les révisions de code (code review information) extraites du systéme de traçage des erreurs (issue tracking system). Nous avons découvert que les raisons pour lesquelles les réviseurs manquent certains bogues était corrélées autant à leurs caractéristiques personnelles qu'aux propriétés techniques des corrections en cours de revue. Ensuite, nous relatons une étude qualitative invitant les développeurs de chez Mozilla à nous donner leur opinion concernant les attributs favorables à la bonne formulation d'une révision de code. Les résultats de notre sondage suggèrent que les développeurs considèrent les aspects techniques (taille de la correction, nombre de chunks et de modules) autant que les caractéristiques personnelles (l'expérience et review queue) comme des facteurs influant fortement la qualité des revues de code.

Mots clés: Code review, code review quality, bug-inducing changes, mining software repositories, Mozilla, empirical study, quantitative analysis, qualitative study, survey

ABSTRACT

Code review is an essential element of any mature software development project; it aims at evaluating code contributions submitted by developers. In principle, code review should improve the quality of code changes (patches) before they are committed to the project's master repository. In practice, the execution of this process can allow bugs to get in unnoticed.

In this thesis, we present an empirical study investigating code review of a large open source project. We explore the relationship between reviewers' code inspections and personal, temporal and participation factors that might affect the quality of such inspections. We first report a quantitative study in which we applied the SZZ algorithm to detect bug-inducing changes that were then linked to the code review information extracted from the issue tracking system. We found that the reasons why reviewers miss bugs are related to both their personal characteristics, as well as the technical properties of the patches under review. We then report a qualitative study that aims at soliciting opinions from Mozilla developers on their perception of the attributes associated with a well-done code review. The results of our survey suggest that developers find both technical (patch size, number of chunks, and module) and personal factors (reviewer's experience and review queue) to be strong contributors to the review quality.

Keywords: Code review, code review quality, bug-inducing changes, mining software repositories, Mozilla, empirical study, quantitative analysis, qualitative study, survey

CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF APPENDICES

LIST OF ABBREVIATIONS

LOC	Patch size
PWE	Patch writer experience
PIDP	Number of people involved in the discussion of a patch
LDP	The length of the discussion of a patch

To my parents and grandparents

To my friends

ACKNOWLEDGMENTS

There are too many people that I would like to thank. I was a perfect stranger to the academic research when I arrived at this university. Once I found myself in a dilemma for not knowing the direction. The valuable guidance of many people saved me from losing myself.

First of all, I would like to thank my supervisor Dr. Olga Baysal for the continued support and guidance from day one. I even cannot find a word to describe how lucky I am to meet her. It is a great honour and enjoyable for me to work together. Without her, nothing can be possible here.

Then, I would like to sincerely thank Oleksii Kononenko and Dr. Latifa Guerrouj for their help during our projects. It has always been a pleasure to collaborate with them and their assistance has been valuable to me during this process.

Special thanks go to Prof. Houari A. Sahraoui and Prof. Philippe Langlais, their help and encouragement is greatly appreciated.

Great thanks go to Prof. Guy Lapalme, Prof. Jian-Yun Nie, Prof. Claude Frasson and Prof. Pascal Vincent. If my study is a wonderful journey in my life, the prettiest scenery is your precious knowledge and I will cherish a lifetime benefit from it.

I would like to express my gratitude to all members in the software engineering group (GEODES) of the University of Montreal for the help they offered. You are my family and you are the best!

Also, I also would like to thank the developers from Mozilla for their participation in our research study.

And last, but not least, to my parents, my family and my friends, thank you for all the supporting and encouragement to help me achieve my dreams. I love you!

REMERCIEMENTS

Il y a trop de gens dont je voudrais remercier la générosité. J'étais un parfait étranger au domaine de la recherche lors de mon arrivée à l'université. Une fois, je me suis retrouvé face à une impasse, sans connaître de direction à suivre. Les conseils précieux qui m'ont été dispensés par mon entourage m'ont sauvé de la perdition.

Tout d'abord, je tiens à remercier ma directrice, Mme Olga Baysal pour son support sans faille et les conseils qu'elle m'a prodigué dès le premier jour. Sans son soutient, je n'aurais pu faire quoi que ce soit.

Ensuite, je voudrais remercier sincèrement le docteur Oleksii Kononenko et la doctoresse Latifa Guerrouj pour leur aide durant nos projets. Ce fut un plaisir sans cesse renouveler que de collaborer avec eux, et leur assistance a été inestimable (durant tout le processus de recherche).

Un remerciement particulier pour les Professeurs Houari A. Sahraoui et Philippe Langlais. Leur aide et leurs encouragements ont été des plus précieux.

Un grand merci au Professeurs Guy Lapalme, Jian-Yun Nie, Claude Frasson et Pascal Vincent. Si mon étude est un moment (important/magnifique) de ma vie, le plus beau paysage fut votre précieux savoir. C'est un bagage dont j'emporterai avec moi les bénéfices le reste de mon existence.

Je voudrais exprimer ma gratitude à tous les membres du groupe de recherche en génie logiciel de l'Université de Montréal (GEODES) pour l'aide qu'il m'ont offert. Vous êtes ma famille et vous êtes les meilleurs.

Aussi, je tiens à remercier les intervenants de chez Mozilla pour leur collaboration dans notre recherche.

Et enfin, mais non des moindres : à mes parents, à ma famille et à mes amis, je tiens à adresser le plus sincère des remerciements pour leur support sans faille, leurs encouragements en toutes circonstances et tous ces petits riens qui, mis bout à bout, m'ont permis aujourd'hui de déposer cette thèse. C'est un rêve qui se réalise, un espoir minutieusement apprécié qui enfin prend forme. Et ce, grâce à vous. Love you!

CHAPTER 1

INTRODUCTION

Code review is an essential element of any mature software development project; it aims at evaluating code contributions submitted by developers. Code review is often thought of as one of best practices of a software project. Code inspections have been proven to be an effective way of identifying defects in the code changes before they are committed into the project's code base [14]. Reviewers, the gatekeepers of a project's master repository, must carefully validate the design and implementation of patches to ensure they meet the expected quality standards.

In principle, code review should improve the quality of code changes (patches) before they are committed to the project's master repository. In fact, one of the main motivations of modern code review is to improve a change to the software prior or after integration with the software system [1]. However, in practice, the execution of this process can allow bugs to get in unnoticed.

In this thesis, we studied code review of a large open source system, Mozilla project. For Mozilla, code review is an important and vital part of their code development since contributions may come not only from Mozilla developers but also from the greater user community. Thus, Mozilla community embraces code review to help them maintain a level of consistency in design and implementation practices among many hackers and various modules of Mozilla [39]. In addition, they employ code review to increase code quality, promote best practices, and reduce regressions [38].

Mozilla reviews every patch; each patch has to be evaluated by at least one reviewer. According to the Mozilla's code review policy [41], reviewers should grant a review if 1) they believe that the patch does no harm and 2) the patch has test coverage appropriate to the change. If reviewers feel that they are incapable of providing a careful review on a certain patch (e.g., due to the lack of time or expertise on a module), they need to re-direct the patch to other reviewers who have better expertise in the area and are able to review code in a timely manner. However, software defects are found after the changes

have been reviewed and committed to the version control repository. These post-release defects raise red flags over the quality of code reviews. Poor reviews that let bugs sneak in unnoticed can introduce stability, reliability, and security problems, affecting user experience and satisfaction with the product.

While existing research on code review studies its various aspects (e.g., the relation between code coverage/participation and software/design quality [32, 35]), the topic related to the quality of code review remains unexplored. In this thesis, 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 [50] to detect bug-inducing changes that are then linked to the code review data extracted from the issue tracking system.

The goal of this thesis is to investigate the impact of the variety of factors on code review quality; to accomplish this, we formualte the following research questions:

RQ1: **Do code reviewers miss 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 [7, 31, 46]. A recent study demonstrated that low review participation has a negative impact on software quality [32].

RQ3: **Does participation in code review influence its quality?**

A recent study demonstrated that low review participation has a negative impact on software quality [32].

RQ4: **Do temporal factors (i.e., related to time) influence the quality of the review process?**

Most studies conclude that day of the week the developers submit their patch correlates with the volume of introduced defects [19, 50].

RQ5: **What factors do developers perceive as most important in contributing to code review quality?**

Soliciting opinions from developers on what factors they find can affect the qual-

ity of their review tasks is important. Qualitative study can allow us to gain such insights into the developer attitudes and perceptions.

The results of our quantitative and qualitative investigation demonstrate that reviewers do miss bugs when performing reviews always. Developer-related characteristics such as their review experience and review loads, we well as the technical properties of the patch such as its size and the number of files it affects are main contributors to the quality of code review.

The work described in this thesis was peer-reviewed and published at the International Conference on Software Maintenance (ICSME-2015) [29].

Thesis organization. The rest of the thesis is organized as follows. Chapter 2 presents relevant research efforts related to code review, software quality, and code review quality. Chapter 3 presents our quantitative study, describing the methodology (data extraction, linkage of issue tracking and version control repositories, etc.), the analysis and the results of our first four research questions. Chapter 4 reports our qualitative study and the results of the survey that was conducted with Mozilla developers on their perception of the main characteristics and factors that contribute to a high quality code review. Chapter 5 concludes this thesis by highlighting our main findings. And finally, Chapter 6 discusses possible future directions we can follow to improve this work.

CHAPTER 2

RELATED WORK

To the best of our knowledge, the topic of code review quality has not been well investigated by our research community. Thus, in this chapter we present most close and relevant research to this topic, including: code review, software quality, as well as code review and software quality.

2.1 Code Review

A large body of work has attempted to assess modern code review by large software systems. Rigby and German presented a first investigation of code review processes in open-source projects. They compared the code review processes of four-open source projects namely, 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 [48]. Later, Rigby and Storey have investigated the mechanisms and behaviours adapted by developers to identify code changes they are competent to review. They explore 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 non-technical issues such as feature, scope, or process issues [49].

This finding is inline with those by Baysal et al. who have shown that review positivity, i.e., the proportion of accepted patches, is also influenced by non-technical factors [5]. The authors have also investigated organizational (the company) and personal dimensions (reviewer load and activity, patch writer experience) on code review response time and outcome. They find that organizational and personal factors influence review timeliness, as well as the likelihood of a patch being accepted.

Prior to this work researchers have found that organizational structure can influence

software quality. 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 [45]. Both Baysal et al. and Nagappan et al. findings agrees with Conway's law [11], which assume that a software system's design reflects the structure of the organization that develops it.

Jiang et al. empirically 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 have shown that developer experience, patch maturity, and priori 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 [23].

A recent qualitative study at Microsoft has been performed to explore the motivations, challenges, and outcomes of tool based code reviews. This investigation reveals that while finding defects remains the main motivation for review, other motivations exist such as knowledge sharing among team members [1].

2.2 Software Quality

Researchers have studied the impact of design and code review on software quality. For example, Kemerer et al. have investigated the effect of review rate on the effectiveness of defect removal and the quality of software products, while controlling for a number of potential confounding factors. This study have empirically shown that allowing sufficient preparation time for reviews and inspections can produce better performance [26]. A more recent work by Kamaie [24] empirically evaluated a "Just-In-Time Quality Assurance" approach to identify in real-time software risky changes. Their study extensively evaluated change-level prediction through the analysis of six open-source and five commercial projects. Their finding reveal that process metrics outperform product metrics when software quality assurance effort is considered [24]. Kim et al. [28]

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.

Śliwerski et al. investigated risky changes, they analyzed CVS archives for problematic fix-inducing changes. They suggested a technique called, SZZ, to automatically locate fix-inducing changes by linking a version archive to a bug database [50]. In this study we applied SZZ to detect bug-inducing changes which we later link to code review data. SZZ was successfully applied to understand whether refactorings inducing bug-fixes [2], as well as to build prediction models that focus on identifying defect-prone software changes [24]. The investigation of Sliwerski 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. Recently, Eyolfson et al. [13] 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.

Several other metrics have been used to predict defects. For example, Graves et al. [18] 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. Nagappan and Ball [44] suggest the use relative code churn metrics, which measure the amount of code change, to predict defect density at the file level. Jiang et al. [22] 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. In a study performed by Moser et al. [36], the authors have shown that process metrics perform similarly to code metrics when predicting defect-prone files in the Eclipse project. Hassan [20] has demonstrated that scattered changes can be used to determine defect-prone files. He analyzed 14 distinct factors leveraged from code changes to predict whether or not a change is buggy. Zimmermann et al. [52] have focused on investigating defect prediction from one project to another using seven commercial projects and four-open source projects. They find that there is no single factor that produce accurate predictions.

Rahman and Devanbu suggested the use of defect prediction models to compare

the impact of product and process metrics [47]. The results of their research suggest that code metrics are generally less useful than process metrics for prediction. In their other work, the authors find that lines of code implicated in a bug fix are more strongly associated with single developer contributions. This finding suggests that code review is an essential part of the software quality assurance [46]. Mende and Koschke [33] 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 to more realistic performance evaluations.

Recent works 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 [7]. 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 [31].

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 people resources can help managers enhance quality outcomes. We hypothesize that a modern code review process can neglect buggy changes and that this may be due to several factors technical, personal, and organizational.

2.3 Code Review and Software Quality

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

Recently, McIntosh et al. empirically investigated the relationship between software quality and code review coverage and code review participation. They find that both

code review coverage and participation significantly impact on software quality. Low code review coverage and participation are estimated to produce components with up to two and five additional post-release defects respectively. These results confirm that poor code review negatively affect software quality [32].

Beller et al. have empirically explored the problems fixed through modern code review in OSS systems. They find that the types of changes due to the modern code review process in OSS are similar to those in the industry and academic systems from literature, featuring a similar ratio of maintainability-related to functional problems [6].

Mäntylä and Lassenius classified the types of defects found in review on university and three industrial software systems [30] 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 [21] found relevant differences in defect finding capabilities among code reviewers.

CHAPTER 3

QUANTITATIVE STUDY

To address our first four research questions we followed a data mining process shown in Figure 3.1 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).

3.1 Background

For the Mozilla projects, the code review could be responsible for its maintaining a level of consistency and compatibility in design and implementation practices. The advantage of this mechanism is to guarantee the quality, security and continuous integration across the many submissions of patch and among the several code changes.

Mozilla employs a two-tier code review process for assessing submitted patches — *review* and *super review* [39]. A *review* is performed by a module owner or peers of the module; a reviewer is someone who has domain expertise in a problem area. *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 [40] for all Mozilla modules, 162 reviewers for Core module [42], 25 reviewers for Firefox [43], and 11 reviewers for Firefox for Android (aka Fennec). However, any person with level three commit access — core product access to the Mercurial version control system — Currently, there are 30 super-reviewers [40] for all Mozilla modules, 162 reviewers for Core module [42], 25 reviewers for Firefox [43], and 11 reviewers for Firefox for Android (i.e., Fennec). However,

any person who is not on the list of designated reviewers but has level three commit access — core product access to the Mercurial version control system — can review a patch.

Mozilla reviews every patch. Bugzilla [1] issue tracking system records and stores all the information related to code review tasks. Developers submit a patch containing their code changes to Bugzilla and request a review from a designated reviewer of the module where the code will be checked in. Reviewers annotate the patch either positively or negatively reflecting their opinion of the code under review. For highly-impactful patches super reviews may be requested and performed. Once the reviewers approve a patch, code changes are committed to the Mozilla's source code repository.

A typical patch review process consists of the following steps:

1. Once the patch is ready and needs to be reviewed, the owner of the patch (writer) requests a review from a reviewer (i.e., a module owner or a peer). The review flag is set to "*review?*". If the owner of the patch decides to request a super review, he or she may also do so and the flag is set to "*super-review?*".
2. When the patch passes a review or a super review, the flag is set to "*review+*" or "*super-review+*" respectively. If the patch fails review, the reviewer sets the flag to "*review–*" or "*super-review–*" and provides explanation on a review by adding comments to a bug in Bugzilla.
3. If the patch is rejected, the patch owner may resubmit a new version of the patch that will undergo a review process from the beginning. If the patch is approved, it will be checked into the project's official codebase.

1. `http://bugzilla.mozilla.org`

3.2 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` [2] 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 work, 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*.

Core module contains components that are used by all other modules, e.g., Gecko layout engine is located here and used by every Mozilla's web browser. contains the source code for the desktop version of Firefox web browser. Firefox for Android module implements a mobile version of Firefox web browser.

Table 3.I describes the main characteristics of these systems; the numbers represent "clean" datasets that we obtained after performing the steps described in Sections 3.3, 3.4, and 3.5.

We report the number of commits, reviews, writers, and reviewers for our Mozilla-all, Core, Firefox, and Firefox for Android datasets.

2. `http://hg.mozilla.org/mozilla-central`

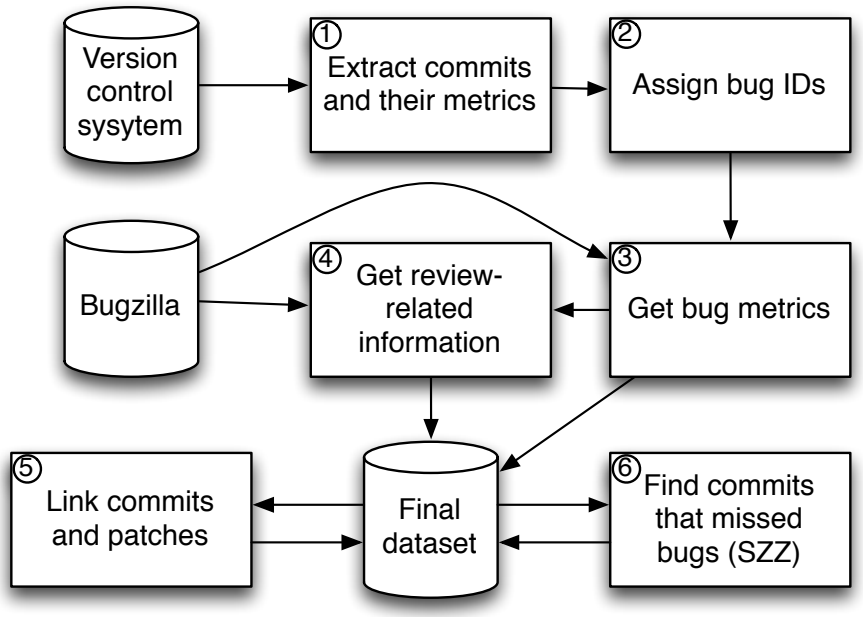

Figure 3.1 – Process overview.

Table 3.I – 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

3.3 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`.

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. [28], 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

did find 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.

3.4 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 the environment where developers constantly make commits to the repository independently from one another. For example, 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, 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, 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,

applying such techniques was beyond the scope of our work.

In our approach, we decided to opt 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 [3, 5]. 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 can not 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.

3.5 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 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).

Table 3.II – A taxonomy of considered technical and personal metrics used.

Type	Metric	Description	Rationale
Technical	Size (LOC)	The total number of added and removed lines of code.	Large commits are more likely to be bug-prone [51]; 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.	Similar, reviews are more likely to be prone to bugs if the change spread across multiple files.
	Module	The name of the Mozilla module (e.g., Firefox).	Reviews of changes within certain modules are more likely to be prone to bugs.
	Priority	Assigned urgency of resolving a 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	Assigned extend to which a bug may affect the system.	We think that changes with higher levels of severity introduce less 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, 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 [5] 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.
	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.

3.6 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

Table 3.III – A taxonomy of considered participation and temporal metrics used.

Type	Metric	Description	Rationale
Participation	Number of developers on CC	The number of developers on the CC list at the moment of review decision.	Linus's law states that "given enough eyeballs, all bugs are shallow" [12].
	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 [32].
	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 [32].
	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?
Temporal	Review time	Time in minutes taken to review a patch.	Quickly reviewed code changes are more likely to be bug-prone.
	Review request week day	Day of week (Mon, Tue, Wed, Thu, Fri, Sat, Sun) the review was requested on.	Are reviews requested on Mondays more likely to introduce bugs than reviews asked for during other weekdays or over the weekend?
	Review week day	Day of week (Mon, Tue, Wed, Thu, Fri, Sat, Sun) the review was submitted on.	Are reviews performed on Mondays more likely to introduce bugs than reviews done during other weekdays or over the weekend?
	Review request month day	Day of month (0-31) the review was requested on.	Are review requests earlier in the month more likely to introduce bugs than requests towards the end of the month?
	Review month day	Day of month (0-31) the review was submitted on.	Are reviews earlier in the month more likely to introduce bugs than reviews at the end of the month?

algorithm proposed by Śliwerski et al. [50] to identify the list of bug-inducing changes.

Śliwerski et al. [50] analyzed CVS archives to identify fix inducing changes — changes that lead to problems caused by fixes. They developed an algorithm named SZZ (Sliwerski, Zimmermann and Zeller) to automatically locate these fix inducing changes by linking a version archive (such as CVS) to a bug database. Many researchers use and apply the SZZ algorithm since it provides a very practical and convenient approach to identify bug-introducing changes from bug-fix changes.

To provide some background on how SZZ works, we now summarize the workflow of the SZZ algorithm descibed in following three main steps:

1. We search for bug fix changes through retrieving bug identifiers or relevant keywords in change logs, or tracking accurately a recorded linkage between a bug tracking system and a specific commit from the version control system.
2. We apply the `diff` function to check out the modification during the bug fixing process. Normally, the diff tool provides a list of regions that differ between the two files. According the original authors' point of view, each region of the code change is called a "hunk". The deleted or modified source code in each hunk is the location of a bug.
3. We continue tracking the origins of the deleted or modified source code in the hunks through using built-in `annotate` feature of the version control systems. For each hunk in the source code, we search and catch the most recent revision that the hunk was modified in, and the developer who was in charged of this change. As a result, we are able to identify the origins discovered as bug-introducing changes.

While the SZZ algorithm allows us to link the bug and its bug introducing changes, based on our experiments on Mercurial version control system, the SZZ algorithm has some limitation. Main flaws can be described as follows:

1. Not all changes can be identified as bug fixing ones. Some modifications caused by bug fix work are not actual bugs (false positives), for example, new comments added, formatting changes and blank lines. Therefore, we can not assume that all hunks in the modifications are relevant to bug fixes.
2. The Mercurial's `annotate` command is not consistent with `annotate` in other version control systems. While it is similar to `blame` in Git, it does not provide the information sufficient to identify bug-introducing changes.
3. Mercurial does not provide a feature to directly track the parent of each line of code in each revision.

To overcome these limitations of the SZZ algorithm, we perfomed a number of modifications to the logic of the initial algorithm.

For each commit that is a bug fix, our implemented 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 with our self-made script tool 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 [27]. 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 rest 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 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 our implemented 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 data set were marked as bug-inducing if they were present in this list; otherwise, they were marked as bug-free commits.

Figure 3.2 presents an illustrative example of how we link a bug reported to the issue tracking system (Bugzilla) to the code changes committed to the version control system and how we identify the change that introduced this bug in the first place. The example also shows how we then link bug-inducing changes with the code review process and the individual reviewer who missed this bug.

1. We extract the list of fixed/resolved bugs (in our example, the bug with bugID=892926) from Bugzilla;
2. We identify the latest revision for bugID 892926 as revision #179722;
3. For each revision (in this example, for revision #179722), we identify all the files changed. Here, we consider the `tabbrowser.xml` as the file modified in this revision (there are typically many files modified in one single revision);
4. We then determine the lines changed for this file (using `hg diff -git`) and identify the parent revision;
5. For each of the lines removed, we determine revision when the bug was introduced ("blamed" revision). Here, we got revision #138289 by executing `hg annotate`;
6. By linking the change back to Bugzilla, we determine the person who reviewed this change (i.e., patch). In this example, the reviewer username is determined

as “dao” who reviewed this patch that was commmitted in revision #138289 and introduced bug #892926.

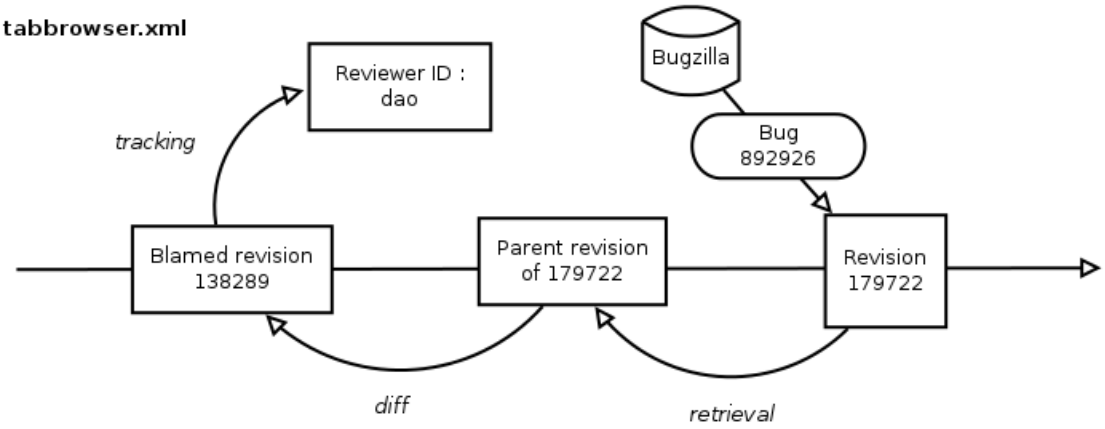

Figure 3.2 – Example of applying the SZZ algorithm to the bug with bugID=892926 for the file `tabbrowser.xml`.

3.7 Determining Explanatory Factors

Previous work demonstrated that various types of metrics have relationship with code review time and outcome [5]. Similarly, we grouped our metrics into four types: *technical*, *personal*, *participation* and *temporal*.

In this section, we describe our factors which practitioners may use to predict buggy code reviews. Additionally, we explore the rational behind using various attributes related to these factors.

Table 3.II and Table 3.III describe the metrics used in our study and provides rationale for their selection. Once we finished building the datasets, we had all the required data for technical and temporal factors. The metrics for technical factors and temporal 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 [37]. 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 four types presented in this section served as explanatory variables for building our models that we describe next.

3.8 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 [9]. 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 3.II.

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. 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 [5, 8, 32, 34].

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 [16], 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 [15], 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.

3.9 Results

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

3.9.1 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 [1]. 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.

Table 3.IV – 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 %
Firefox4Android	2,160	1,210	56.0 %

As indicated in Table 3.IV, 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 3.I), the proportion of "buggy" code reviews in these modules is almost identical.

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. [28] 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 [50] found 42% of bug-inducing changes

for Mozilla and 11% for Eclipse projects (the dataset contained changes and bugs before 2005).

3.9.2 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 3.II and Table 3.III to our MLR model.

Table 3.V 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. 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" [4] 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 [13, 46] 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 re-submitted by a developer seem to have a positive effect on review quality for one of

Table 3.V – Model statistics for fitting data. Values represent regression coefficients associated with **technical** and **personal** factors.

	Mozilla	**Core**	**Firefox**	**FF4A**
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***	·	·	†

†Disregarded during VIF analysis (VIF coefficient > 5).
⋆ "It's complicated": for categorical variables see explanation of the results in Section 3.9.
FF4A = Firefox for Android.
‡Disregarded during stepwise selection.
Statistical significance: '***' $p < 0.001$; '**' $p < 0.01$; '*' $p < 0.05$; '·' $p \geq 0.05$.

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.

3.9.3 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 quality of software systems [32]. 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 3.III to our models.

Table 3.VI 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” [4].

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 3.VI 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 [4]. 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

Table 3.VI – Model statistics for fitting data. Values represent regression coefficients associated with **technical** and **participation** metrics.

	Mozilla	**Core**	**Firefox**	**FF4A**
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).
⋆ "It's complicated": for categorical variables see explanation of the results in Section 3.9.
FF4A = Firefox for Android.
‡Disregarded during stepwise selection.
Statistical significance: '***' $p < 0.001$; '**' $p < 0.01$; '*' $p < 0.05$; '·' $p \geq 0.05$.

review is a complex process involving personal and social aspects [10].

Table 3.VI 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" [4] 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.

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.

3.9.4 RQ4: Do temporal factors influence the quality of the review process?

Previous research found that time-related factors do contribute to the quality of software systems. A known fact is that developers should not commit their patches on Fridays as they are more likely to introduce bugs on this day of the week [50]. Consequently, builds in the first half of a month and builds on Wednesday are more likely to pass certification (and not being buggy) [19]. Therefore, we want to investigate whether temporal metrics such as review time, the day of the week/month when the review was requested or done, influence the results of a code review process. The intuition here is that the more time reviewers spends on a patch, the better inspection and feedback they can provide. To investigate whether code review quality is affected by the time-related attributes, we add the temporal metrics described in Table 3.III to our models.

We first examine the review time and rate. Table 3.VIII reports the median and mean values of review time (in minutes) and rate (lines per hour) for all four systems. An average review takes around 43 minutes, and review rate is 136 lines of code per hour (i.e., mean values for Mozilla all). This shows that reviewers allocate an appropriate amount of time to inspect code changes. According to Kemerer and Paulk [25], best practices suggest that code should not be reviewed at a rate faster than 200 lines per hour.

Table 3.VII shows that `review time` has a statistically significant impact on review bugginess only for Firefox for Android system; and its estimated coefficient is positive, indicating that an increase in time tends to make reviewers miss problems with code. This is surprising because a common belief among reviewers is that "reviews are only as good as the time you spend on them" [3]. In the models for other three systems, `review time` is removed due to the lack of explanatory power.

For Mozilla all and Core, regression coefficients for the categorical variable `review request week day` = Friday are 0.035 ($p < 0.05$) and 0.05 ($p < 0.01$) respectively. This demonstrates that if code reviews are requested on Friday, reviewers are more likely

3. http://benjamin.smedbergs.us/blog/2014-10-22/how-i-do-code-reviews-at-mozilla/

Table 3.VII – Model statistics for fitting data. Values represent regression coefficients associated with **technical** and **temporal** factors.

	Mozilla	**Core**	**Firefox**	**FF4A**
Adjusted R^2	0.127	0.125	0.167	0.152
Size	0.208***	0.202***	0.218***	0.167***
Chunks	†	†	†	0.175***
Number of files	0.115***	0.118***	0.152***	‡
Module	⋆	n/a	n/a	n/a
Priority	‡	⋆	‡	⋆
Severity	‡	‡	·	‡
Super review	-0.148***	-0.169***	·	n/a
Review time	·	‡	‡	0.009*
Review request week day	⋆	⋆	‡	‡
Review week day	†	†	†	†
Review request month day	†	†	†	†
Review month day	‡	⋆	‡	⋆

†Disregarded during VIF analysis (VIF coefficient > 5).
⋆ "It's complicated": for categorical variables see explanation of the results in Section 3.9.
FF4A = Firefox for Android.
‡Disregarded during stepwise selection.
Statistical significance: '***' $p < 0.001$; '**' $p < 0.01$; '*' $p < 0.05$; '·' $p \geq 0.05$.

Table 3.VIII – Review time (in minutes) and rate (lines per hour).

System	Review Time		Review Rate	
	Median	Mean	Median	Mean
Mozilla all	6.4	43.1	5.2	136.3
Core	7.2	47.5	4.9	142.3
Firefox	5.7	40.5	5.7	120.7
Firefox for Android	3.7	21.0	6.2	92.7

to neglect bugs in these code changes. This confirms previous findings by Śliwerski et al. that patches submitted on Fridays are more likely to induce fixes [50]. Developers, examining these results, can derive simple rules of thumb to follow, such as the fact that picking a Friday to ship your patches for review is not a good bet.

When examining the result for `review month day`, we see that for Core module developers reviewing contributions on day=7 are less likely to miss bugs (regression coefficient = 0.06, $p < 0.05$), while for the Firefox for Android module, reviews done in the second half of the month are more likely to become rigorous in nature. For contributors, these findings suggest to submit their patches for review later in the month if they are not quite sure whether their code changes are truly flawless.

From Table 3.VII we can see that while temporal metrics might have an effect on review quality, technical attributes remain better predictors, in particular the size of the changes and the number of files affected by these changes. 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 four 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).

3.10 Threats to Validity

In this section, we discuss the possible threats to the validity of our quantitative study.

External validity. Threats to external validity concern the possibility of generalizing our results. To make our results as generalizable as possible, we analyzed various modules of Mozilla including Core, Firefox, Firefox for Android. We believe the number of the analyzed modules is sufficient enough to generalize our results. However, our findings cannot be generalized across all open source projects. Further research is 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.

In Bugzilla, bug reports per se actually serve several purposed: they can be bug-fix requests, or requests for adding new functionality, or documentation-related changes, etc. Since Bugzilla does not provide mechanisms of distinguishing between "true" bugs and new feature requests, we treat all changes as bug fixes. Also, Bugzilla stores all times and dates in the UTC timezone. Thus, day of the week and day of the month metrics might not reflect the actual timezone of developers and times/dates of the patches or reviews they submit.

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.

CHAPTER 4

QUALITATIVE STUDY

A survey is an efficient way that can promote understanding and communication between researchers and developers. We decided to further explore how developers perform code review, what criteria affect their judgement on accepting or rejecting patches and what challenges they face when performing these tasks. We conducted a qualitative study with Mozilla developers to study the reasons of what factors they perceive as most important in contributing to code review quality. In this chapter, we describe our survey, its design, analysis, and results.

4.1 Methodology

In this section, we present how we design our survey, who our participants are and how we analyze the feedback we received.

Our methodology of conducting the survey is similar to the method used by Gousios et al. [17]. Our survey generally consists of three kinds of questions: the multiple choice questions with fixed option, the Likert-scale questions and free answer questions. The survey starts with general investigation and ends with specific questions we were interested in studying. We solicited responses from the Mozilla developers. Our survey was design to have developers spend between 5 to 10 minutes to complete the questionnaire and was published to two Mozilla forums such as "Mozilla developer platform" and "Mozilla code review" forums, we also sent out an email with the link to the survey to individual developers. Developers could browse our questionnaire and contribute their opinion by answering 20 questions.

To better understand $developers'$ working background and context, we prepare some question for investigating their demographic characteristics. We suspect that the developer's working experience, occupational habits and professional preference are relevant to code review quality. For example, their role in the project(s), experience, the type of

code review activity they are involved in (writing, reviewing, discussing or other), etc. This information, we believe, can help us to get a clear picture of their personal working background. On the other hand, some working context information, such as, the number of patches that developers submit for a review every week, the period that developers review patches, the environment and tools they typically conduct code review with and the number of patches they review every week can satisfy our curiosity for exploring the details about their code review practices. During this part of the survey, some questions are provided with simple selection of answers, and the open-ended questions are intermixed with multiple choice ones.

For the purpose of investigating the influence and incidence of different factors on the decisions made by code reviewers and the time code review processes take, we also design the Likert-scale questions, that is the most widely used approach to scaling responses in survey research by asking people to respond to a series of statements about a topic, in terms of the extent to which they agree with them, and so getting insights into the cognitive components of attitudes and perception.

In our survey design, the Likert-type or frequency scales is used to fixed choice response formats and is designed to measure developers' perception of the different levels of impact from different factors on their code review tasks. These ordinal scales measure levels of developers' agreement/disagreement to those different factors that we present to them. Here, we assume, in our Likert-type scale approach, the concurrence/non-concurrence of feeling is linear, which means a Likert-scale from strongly agree to strongly disagree, and makes the assumption that attitudes can be measured here. In our survey, participants were offered a choice of five pre-coded responses with the neutral point (Strongly disagree, Disagree, Neither agree nor disagree, Agree, Strongly agree). In the survey, developers should follow their interior feelings of choosing the right answer from our Likert-scale metrics.

Although our main purpose of this research is to investigate the Influence of various dimensions on code review quality, we expect that developers can provide useful and rich information to validate our research. Therefore, we also design some open ended question, which means there are no fixed choices and participants could answer the ques-

tion according to their experience and beliefs. For example, we ask them to comment on the other potential factors that could affect code review timeliness and outcome, as well as to describe characteristics that could contribute to a well-done code review. Free-form answers can facilitate unexpected results, yet they allow us to solicit interesting information. On the other hand, we are also curious about what challenges developer face when conducting code review and how they assess the quality of submitted patches. In addition, we also ask participants about what tools they would like to have available to them to assist their code review activities.

4.2 Analysis and Results

In this section, we present our findings and results of the survey.

In a survey, the background of participants and their working habits is an important and interesting indicator to an investigator. To gather background and demographic information, we carefully design several questions related to the developers' working context.

Figure 4.1 – Developer role on the project(s).

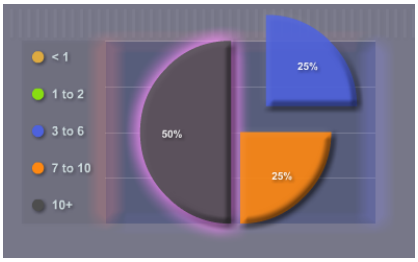

Figure 4.2 – Developer experience (years).

From Figure 4.1, we find that the majority of the participants (88%) associate themselves as software developers or engineers. Only 14.3 % of the participants perform the role of a project manager or QA engineer; and the rest of them (approximately 13%)

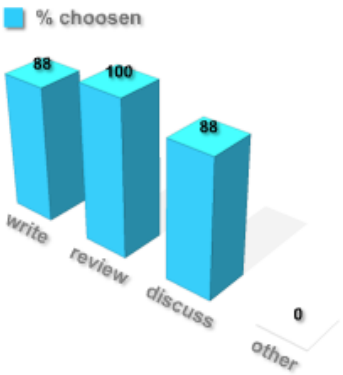

Figure 4.3 – Developer participation in code review activities.

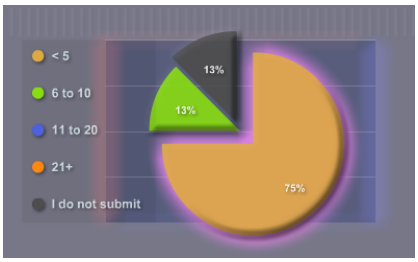

Figure 4.4 – Weekly patch submission.

work as release managers. We believe that this distribution of roles is a typical one for a mature development and code review team. We should note, however, that some developers play multiple roles on the same project. For example, a developer can be a team lead and at the same time she can also perform tasks of a code reviewer; some responsibilities such as code reviewer are given based on the developer expertise and experience on the project.

Table 4.2 demonstrates the distribution of the developers' experience (number of years). We measured developer experience by their time being on a project and defined 5 levels: less than 1 year, 1 to 2 years, 3 to 5 years, 7 to 10 years, more than 10 years. According to this table, it is clear that all the participants have substantial experience of software development. One half of the participants have over 10 years of experience accounted for the greatest proportion of whole population (50%). The group of "7 to 10 years" represented 25% of the population and the rest had at least 3 years of software development. These results supported our confidence in the construct validity of our survey.

Table 4.3 demonstrates how developers are involved in code review tasks. The results show that all participants have done patch reviews and the majority of them (88%) participate in patch writing and discussions of patches and bugs. The majority of developers recognize the value of code review and enthusiastically participate in code review

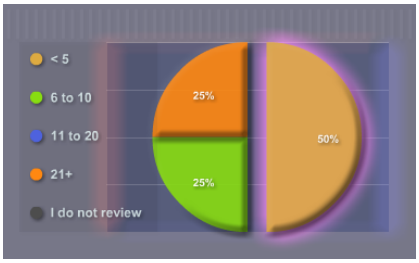

Figure 4.5 – Weekly patch reviews.

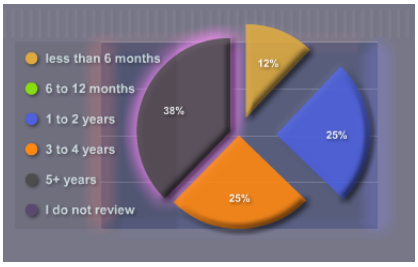

Figure 4.6 – Developer code reviewing experience.

to safeguard of project quality.

Table 4.4 and Table 4.5 provide information about how many patches developers submit for a review and how many patches they review every week. Most developers, about 75%, submit less than 5 patches every week, while there are 25% of developers review more than 21 patches. The rest of them (approximately 13%) review 6 to 10 patch every week. According to our previous questions, developers who review more than 21 patches per week have experience in software development (more than 10 years). By contrast, when it comes to weekly review loads, 1/2 of the participants choose less than 5, 25% of them review 6 to 10 patches per week and 25% of them review more than 21 patches per week. It is clear that some developers at Mozilla work primarily on fixing bugs, while others are primarily involved in conducting patch reviews. We speculate that developers who take management positions in the project, like a team lead or a release manager, always review a great amount of patches. While developers may be involved in other duties on the project (meetings, educational activities, etc.), our findings suggest that code review are more time consuming that bug fixing.

Table 4.6 demonstrates how long our participants have been reviewing patches, which reflects their working experience as code reviewers. The majority of out participants (38%) have a substantial experience reviewing patches ("more than 5 years"). 25% of developers conducted code review for "3 to 4 years" and 25% of them have experience of "1 to 2 years". The beginner type makes up the rest of them (about 12%). Thus, our participants have a wide range of reviewing experience levels.

Table 4.7 presents the environments developers typically conduct code review in.

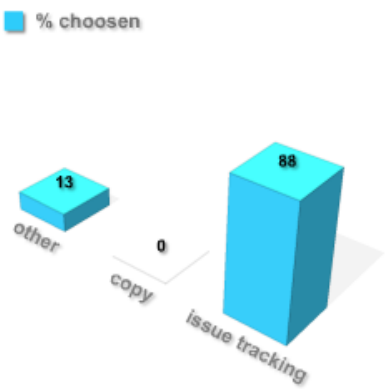

Figure 4.7 – Environments for conducting code review.

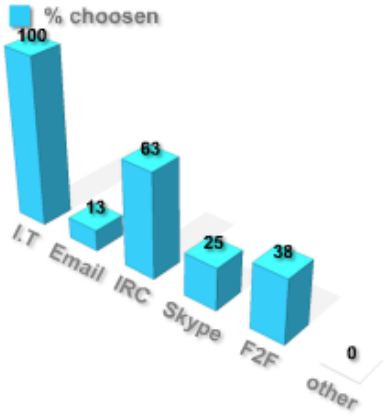

Figure 4.8 – Venues for discussing patches.

The results demonstrate that the issue tracking system remains the most popular choice for performing code review tasks. However, some developers review code using Mozilla's code review tool called MozReview, which offers each individual developer a ReviewBoard.

Figure 4.8 shows where developers prefer to discuss patches. The issue tracking system is the most common channel for patch or bug discussions (100%), compared to 13% of discussions that happen via email, 63% in the IRC, 25% via Skype/Hangout and 38% via face-to-face interactions. According to our result, issue tracking system, such as Bugzilla, serves as a central environment for Mozilla developers to perform code review as it integrates many important features, such as bug reporting, review assignment, patch submission and status monitoring, etc., to assist developers with all their project communication and tracking their daily activities.

Figure 4.9 illustrates the degree of recognition and the likelihood of detecting problems with a patch and demonstrates developer perceptions about the influence of different factors on the quality of code review. We can see that the reviewer experience is strongly agreed by our participants to have a great impact on code review. According to developers' opinions, the patch size (LOC), code chunks, module, review queue (i.e., review load) are strong contributors to the quality of code review. Most of developers believe that review response time, the length of the discussion of a patch, the

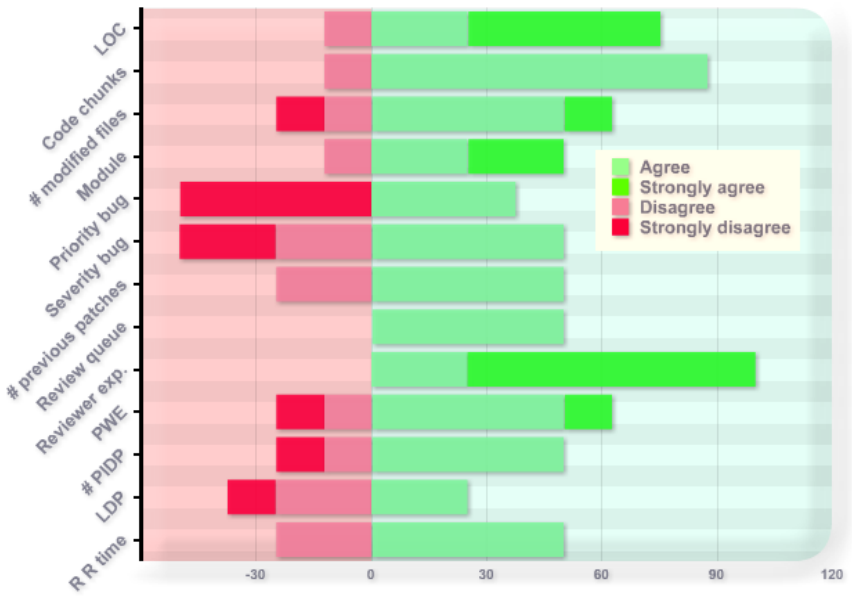

Figure 4.9 – Factors influencing code review quality.

number of people involved in the discussion of a patch, priority of a bug do not have a significant impact on review quality. Moreover, factors such as review response time, priority/severity of a bug stimulated a controversy among the expressed opinions.

Now, we would like to report the analysis of two open questions related to the quality of code review.

For the question related to the characteristics that may contribute to a well-done code review, we received various answers. Some developers say that a good code review should "understand the problem it's trying to solve and at the same time to understand the code it's modifying", as well as to "verify that the change solves the problem and verify that the fix is as clear, simple, and straightforward as it can be consider performance implications". Developers also suggest that it is quite important to give a constructive, clear feedback to the author of the patch, "we should avoid nitpicking insignificant details, and make a clear distinction between "nits" and "important feedback", thorough understanding of the problem and solution (both by writer and reviewer)". One of the

Mozilla team manager believes that "it really does matter every party knowing what the patch is for and exactly what problem it solves" and that it is essential that each party involved in code review has a good understanding of how to test/run/write.

One developer mentions that a good reviewer should pay attention to security elements such as memory management, "no leaks", "no unsafe memory usage"; while other developers feel that tooling is important, "good diffs, as well as interdiffs of past submitted patches". Most participants agree that a patch writer has to take the coding style into consideration "to avoid loosing time on nitpicking", while reviewers should provide pointers to the patch writer about how to solve particular review feedback points.

Some developers report that the code and annotation should be thorough: a high quality review is the one "done while attentive (as opposed to tired)" and "not done under stress like when there's a deadline".

Finally, for the question about other factors that may affect code review quality, one developer suggests that considering "how tricky/complex the code being modified is" may be important. While others believe that reviewer's style of making reviews may influence the outcome of the review. For example, "some reviewers expect more followup bugs to be filed and fixed, some want all the issues to be fixed in the original bug". Also, one developer believes that reviewer's expertise in a certain type of bugs/changes should be considered, "say, I'm experienced in memory management issues, so I tend to focus on those issues more than some other reviewers".

The results of the survey suggest that, based on the developers' opinions and feedback, both technical (such as patch size, number of chunks, and module) and personal factors (reviewer's experience and review queue) are strong contributors to the code review quality. These findings confirm the results obtained during our quantitative study (described in Chapter 3) which reported that both technical and personal factors are strong predictors of the quality measures.

4.3 Threats to Validity

In this section, we discuss the possible threats to the validity of our qualitative study.

External Validity: The results of our qualitative study can not be applied to all open source projects since all our survey participants were Mozilla developers. Other OSS projects might employ different code review practices, thus making the generalizability of our findings a possible threat.

Internal Validity: We carefully designed the survey to gain insight related to the code review practices and code review quality. We used both multiple choice questions, as well as open ended questions (other). As with any explanatory study, it is possible that we could introduce some bias when interpreting the open-ended answers.

CHAPTER 5

CONCLUSION

Code review is an essential and vital part of modern software development. 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 thesis, we have explored the topic of code review quality by investigating what factors might affect it. We tried to understand what aspects contribute to poor code review quality to help software development projects better understand their processes and practice. 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. Furthermore, we conduct a survey with the actual Mozilla developers to explore their attitudes and perception of code review and its quality, as well as other factors that may influence review quality.

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 6

FUTURE WORK

In our quantitative study, we mainly investigate the influence of three groups of factors such as personal, temporal and participation, that can have an impact on code review quality. However, the result of our qualitative study suggest that there are a number of other factors that were not considered in this work but may be important indicators of good or poor review quality. We plan to further explore the topic of code review quality by exploring other data sources and metrics such as code change complexity, reviewer's style of conducting review tasks, as well as his expertise in a certain type of bugs.

In this work we only investigated code review process of one open source project. While the Mozilla projects is a large OSS projects and is considered as a good representative, further research including other open source projects of various sizes is needed to confirm our findings. We plan to conduct a large-scale qualitative study with developers from various software development projects to validate our preliminary results on the developer perceptions of high quality code review.

While we received good feedback from our participants, we believe that in order to gain better insights into the quality of code reviews and how developers define quality in this context, we need to hear individual stories. For that, we plan to interview developers and seek their detailed opinions about code review practices, as well as to conduct observations of how they perform code review tasks to better understand, assess and assist developers with their daily activities and tasks surrounding code reviews.

BIBLIOGRAPHY

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

[2] 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 2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*, SCAM '12, pages 104–113, Washington, DC, USA, 2012. IEEE Computer Society.

[3] O. Baysal, O. Kononenko, R. Holmes, and M.W. Godfrey. The secret life of patches: A firefox case study. In *Proc. of the 19th Working Conference on Reverse Engineering*, pages 447–455, 2012.

[4] 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. URL `http://www.cs.uwaterloo.ca/research/tr/2012/CS-2012-10.pdf`.

[5] Olga Baysal, Oleksii Kononenko, Reid Holmes, and Michael W. Godfrey. The influence of non-technical factors on code review. In *WCRE*, pages 122–131. IEEE, 2013.

[6] 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.

[7] Christian Bird, Nachiappan Nagappan, Brendan Murphy, Harald Gall, and Premkumar Devanbu. Don't touch my code!: Examining the effects of ownership on soft-

ware 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.

[8] M. Cataldo, A. Mockus, J.A. Roberts, and J.D. Herbsleb. Software dependencies, work dependencies, and their impact on failures. *Software Engineering, IEEE Transactions on*, 35(6):864–878, Nov 2009.

[9] J. Cohen. *Applied Multiple Regression - Correlation Analysis for the Behavioral Sciences*. 2003.

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

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

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

[13] 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, New York, NY, USA, 2011. ACM.

[14] M. E. Fagan. Design and code inspections to reduce errors in program development. *IBM Syst. J.*, 15(3):182–211, September 1976.

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

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

[17] Georgios Gousios, Andy Zaidman, Margaret-Anne Storey, and Arie Van Deursen. Work practices and challenges in pull-based development: the integrator's perspective. Technical report, Delft University of Technology, Software Engineering Research Group, 2014.

[18] Todd L. Graves, Alan F. Karr, J. S. Marron, and Harvey Siy. Predicting fault incidence using software change history. *IEEE Trans. Softw. Eng.*, 26(7):653–661, July 2000.

[19] A.E. Hassan and K. Zhang. Using decision trees to predict the certification result of a build. In *Automated Software Engineering, 2006. ASE '06. 21st IEEE/ACM International Conference on*, pages 189–198, Sept 2006.

[20] Ahmed E. Hassan. Predicting faults using the complexity of code changes. In *ICSE*, pages 78–88. IEEE, 2009.

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

[22] 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, New York, NY, USA, 2008. ACM.

[23] 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, Piscataway, NJ, USA, 2013. IEEE Press.

[24] 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 Trans. Software Eng.*, 39(6):757–773, 2013.

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

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

[27] Sunghun Kim, Thomas Zimmermann, Kai Pan, and E. James Jr. Whitehead. Automatic identification of bug-introducing changes. In *ASE '06: Proceedings of the 21st IEEE International Conference on Automated Software Engineering (ASE'06)*, pages 81–90. IEEE Computer Society, 2006.

[28] Sunghun Kim, E. James Whitehead, Jr., and Yi Zhang. Classifying software changes: Clean or buggy? *IEEE Trans. Softw. Eng.*, 34(2):181–196, March 2008.

[29] Oleksii Kononenko, Olga Baysal, Latifa Guerrouj, Yaxin Cao, and Michael W. Godfrey. Investigating code review quality: Do people and participation matter? In *Proceedings of IEEE International Conference on Software Maintenance*, Bremen, Germany, Sept. 29–Oct. 01, 2015.

[30] Mika V. Mantyla and Casper Lassenius. What types of defects are really discovered in code reviews? *IEEE Trans. Softw. Eng.*, 35(3):430–448, May 2009.

[31] 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, New York, NY, USA, 2010. ACM.

[32] 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, New York, NY, USA, 2014. ACM.

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

[34] 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, New York, NY, USA, 2010. ACM.

[35] 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 *Proc. of the 22nd Int'l Conf. on Software Analysis, Evolution, and Reengineering (SANER)*, page To appear, 2015.

[36] 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, New York, NY, USA, 2008. ACM.

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

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

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

[40] Mozilla.org. Mozilla super-review. `http://www-archive.mozilla.org/hacking/reviewers.html`, February 2015.

[41] MozillaWiki. Code Review.

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

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

[44] Nachiappan Nagappan and Thomas Ball. Use of relative code churn measures to predict system defect density. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 284–292, New York, NY, USA, 2005. ACM.

[45] 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, New York, NY, USA, 2008. ACM.

[46] 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.

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

[48] 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.

[49] 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, New York, NY, USA, 2011. ACM.

[50] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? In *Proceedings of the 2005 International Workshop on Mining Software Repositories*, MSR '05, pages 1–5, New York, NY, USA, 2005. ACM.

[51] Peter Weissgerber, Daniel Neu, and Stephan Diehl. Small patches get in! In *Proc. of the 2008 Int. Working Conf. on Mining Soft. Repos.*, pages 67–76, 2008.

[52] 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, New York, NY, USA, 2009. ACM.

Appendix I

Qualitative Study's Supporting Materials (Survey Form)

A Study Investigating Code Review Quality

* Required

1. **1) How would you describe your role on the project(s)?** *
   Check all that apply.
   Check all that apply.

   - [ ] Software Developer/Engineer
   - [ ] Project Manager/Lead
   - [ ] QA/Testing Engineer
   - [ ] Other: .....

2. **2) How many years of experience do you have in software development?** *
   Mark only one oval.

   - ◯ < 1
   - ◯ 1 to 2
   - ◯ 3 to 6
   - ◯ 7 to 10
   - ◯ 10+

3. **3) You work for:** *
   Mark only one oval.

   - ◯ Mozilla
   - ◯ Red Hat
   - ◯ Other: .....

4. **4) How are you involved in code review?** *
   Check all that apply.

   - [ ] Writing patches
   - [ ] Reviewing patches
   - [ ] Discussing patches/bugs
   - [ ] Other: .....

5. **5) On average, how many patches do you submit for a review every week?** *
   Mark only one oval.

   - ◯ < 5
   - ◯ 6 to 10
   - ◯ 11 to 20
   - ◯ 21+
   - ◯ I do not submit

6. **6) How long have you been reviewing patches?** *
   Mark only one oval.

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

7. **7) On average, how many patches do you review every week?** *
   Mark only one oval.

   - ◯ < 5
   - ◯ 6 to 10
   - ◯ 11 to 20
   - ◯ 21+
   - ◯ I do not review

8. **8) In what environment do you typically conduct code review?**
   Mark only one oval.

   - ◯ Issue tracking (e.g., Bugzilla)
   - ◯ Copy a patch locally into editor/IDE
   - ◯ Other: .....

9. **9) Where do you discuss patches?** *
   Check all that apply.

   - [ ] Issue tracking
   - [ ] Email
   - [ ] IRC
   - [ ] Skype/Hangouts
   - [ ] Face-to-face discussions
   - [ ] Other: .....

10. **10) The following factors influence code review DECISIONS (Accept or Reject):** *
    Mark only one oval per row.

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

    .....

    .....

    .....

    .....

    .....

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

Mark only one oval per row.

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

.....

.....

.....

.....

.....

14. **14) How do you asses the quality of a patch?** *

.....

.....

.....

.....

.....

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

.....

.....

.....

.....

.....

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

Mark only one oval per row.

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

.....

.....

.....

.....

.....

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

    .....

    .....

    .....

    .....

    .....

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

    .....

    .....

    .....

    .....

    .....

20. **20) If you want us to keep you updated on the results of this study, please provide your email.**

    .....
