Supporting Modern Code Review

DongGyun Han

A dissertation submitted in fulfilment
of the requirements for the degree of

Doctor of Philosophy
of
University College London

Department of Computer Science
University College London

15 February 2019

Declarations

I, DongGyun Han, confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the thesis. The papers presented here are original work undertaken between April 2015 and February 2019 at University College London. They have been submitted for publication as listed below with a summary of my contributions to the papers:

1. D. Han, J. Krinke, M. Paixao, C. Ragkhitwetsagul, and G.Rosa, 'Pretty Patches: An Empirical Study of Coding Conventions During Code Review', submitted to 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), 2019 (under review).
   Contribution: I managed and lead the study, including the experiments and manual investigation. I designed the research questions and analysed data to derive answers to the research questions. In addition, I participated in the manual investigation as an investigator. This paper is presented in Chapter 4

2. M. Alonaizan, D. Han, J. Krinke, C. Ragkhitwetsagul, D. Schwartz-Narbonne, and B. Zhu, 'Recommending Related Code Reviews', major revision submitted to Transactions on Software Engineering (TSE), 2018 (major revision).
   Contribution: This study is based on the master's thesis of Manal Alonaizan, who is an author of the paper. During her master's program, Dr. Jens Krinke and I managed her master thesis. After her graduation, I extended the paper by conducting additional experiments, rewriting the contents, and adding an interview based on our technique during my Amazon internship. This paper is presented in Chapter 5.

In addition, I have co-authored the papers below during my PhD program. The following papers are not presented in this thesis:

1. M. Paixao, J. Krinke, D. Han, C. Ragkhitwetsagul, and M. Harman, 'The Impact of Code Review on Architectural Changes', IEEE Transactions on Software Engineering (TSE), 2018 (major revision).

2. M. Paixao, J. Krinke, D. Han, and M. Harman, 'CROP: Linking Code Reviews to Source Code Changes', in Proceedings of the 15th International Conference on Mining Software Repositories (MSR2018), 2018. (Data Showcase)

3. M. Paixao, J. Krinke, D. Han, C. Ragkhitwetsagul, and M. Harman, 'Are Developers Aware of the Architectural Impact of Their Changes?', in Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017), 2017.

4. T. Lee, J. Nam, D. Han, S. Kim and H. In, 'Developer Micro Interaction Metrics for Software Defect Prediction', IEEE Transactions on Software Engineering (TSE), 2016.

Date

Signature

Abstract

Modern code review is a lightweight and asynchronous process of auditing code changes that is done by a reviewer other than the author of the changes. Code review is widely used in both open source and industrial projects because of its diverse benefits, which include defect identification, code improvement, and knowledge transfer.

This thesis presents three research results on code review. First, we conduct a large-scale developer survey. The objective of the survey is to understand how developers conduct code review and what difficulties they face in the process. We also reproduce the survey questions from the previous studies to broaden the base of empirical knowledge in the code review research community. Second, we investigate in depth the coding conventions applied during code review. These coding conventions guide developers to write source code in a consistent format. We determine how many coding convention violations are introduced, removed, and addressed, based on comments left by reviewers. The results show that developers put a great deal of effort into checking for convention violations, although various convention checking tools are available. Third, we propose a technique that automatically recommends related code review requests. When a new patch is submitted for code review, our technique recommends previous code review requests that contain a patch similar to the new one. Developers can locate meaningful information and development context from our recommendations.

With two empirical studies and an automation technique for recommending related code reviews, this thesis broadens the empirical knowledge base for code review practitioners and provides a useful approach that supports developers in streamlining their review efforts.

Impact Statement

To improve the current practice, two steps are required: understanding the current issues and providing solutions for them. This thesis investigates the current issues in code review practice via two large-scale empirical studies and proposes an automated technique that recommends related code reviews. In addition, we conduct a thorough literature review to summarises state-of-the-art publications in code review research.

A large-scale developer survey targeting open source developers helped us to comprehend developers' expectations and reality. We extracted 16 questions in four categories from a preliminary interview with 12 developers. In addition, we reproduced five questions from two previous survey papers and compared our results with the published results to extend the body of empirical knowledge. Overall, our survey results provide a better understanding of open source developers' code review practices in terms of demographics, reviewer selection methodologies, expected and actual review time, practices in consulting previous reviews, motivation for code review, and review criteria.

We conducted a second empirical study to investigate coding convention issues during code review. We found that developers waste time checking simple coding convention violations, such as trailing whitespace, even though various automated convention checkers exist to support developers in these tasks. Based on our empirical studies, researchers and practitioners can better understand the current issues developers face and how to support them.

Lastly, we proposed a related review recommendation technique. Our approach recommends related reviews by computing similarities between newly submitted patches and previously reviewed patches. To the best of our knowledge, there is no existing technique to support developers by automatically locating related reviews; instead, developers need to manually track related reviews.

Acknowledgements

My PhD program and this dissertation could not have been finished without the sincere support of many people. I gratefully acknowledge the people who really supported and loved me here.

First of all, I thank my first supervisor, Dr Jens Krinke, for his dedicated supervision. I understand how much he has suffered because of the stubborn student (i.e. me) who is hard to persuade. However, he has always done his best to persuade me logically, to guide me to the correct way, and to derive the best research result with me. Without his kind and dedicated supervision, I cannot imagine my PhD could have been completed.

I also appreciate my co-second supervisors, Prof. Mark Harman and Dr Federica Sarro, who have provided much feedback to help me finish my PhD. In addition, I would like to recognise all members at CREST, UCL. My research was improved by discussions with talented CRESTies and by their kind help. I am especially grateful to Bobby Bruce, Carlos Gavidia, Matheus Paixao, and Chaiyong Ragkhitwetsagul.

Dr Alberto Bacchelli and Dr David Clark spent their valuable time reading my PhD thesis for my PhD viva as examiners. I appreciate their constructive comments to improve this thesis. I also appreciate Dr Dongsun Kim for his fruitful proofreading of this thesis.

Without sincere supervision and support from Dr Sunghun Kim and Prof. Hoyoung Kwak, I could not even have started my PhD. Dr Sunghun Kim taught me many research skills during my MPhil at Hong Kong University of Science and Technology (HKUST). Prof. Hoyoung Kwak provided diverse research experiences during my undergraduate time at Jeju National University.

Four internship periods at Amazon provided me with a range of research ideas. I could experience software engineering research in industry and watch how real developers work in practice. In particular, Daniel Schwartz-Narbonne supported me a great deal as a mentor during my four internship periods. In addition, I am grateful for the automated reasoning group (ARG) and developer tools at Amazon.

I value the friendship and support of all my Korean friends from my hometown, Jeju Island. I especially appreciate Changsoo Moon, Sunghwan Cho, Jihyun Kim, Hongyoon Oh, and Jae-eun Lee.

I also want to acknowledge Ryan at Kakao Friends and M&M's. Without their cheering, I might have dropped my PhD in the middle (seriously).

And last but not least, I would like to thank my family for their support all the way from the very beginning.

Contents

List of Figures

List of Tables

1 Introduction

Code review is the process by which one developer reviews the source code of another. Since Fagan [1976] proposed code inspection, which is a formal and synchronous code review technique, many researchers have shown various benefits of code review as a tool for detecting defects. As a result, many companies and organisations have adopted this technique in their development process.

However, the synchronous characteristic of code inspection hinders its adoption in practice. Code inspection requires an offline meeting with developers other than the author. This means that the developers who are needed to attend the code inspection meeting must stop their work to review their colleague's code. For example, if five developers attended a code inspection meeting for an hour, the company has spent five person-hours to review code snippets, which is very expensive and wasteful. In addition, scheduling a meeting and waiting for other colleagues also requires time. Because of their time-consuming and cumbersome characteristics, many practitioners try to avoid code inspection meetings, regardless of their benefits.

Recently, modern code review is becoming popular (Bacchelli and Bird [2013]), and many companies have adopted the technique in their development process. In contrast to the code inspection technique, modern code review is a tool-based, asynchronous, and informal code review process. Developers submit their code changes to an online code review tool whenever they are ready to integrate their changes without interrupting other team members. As soon as a developer submits a change to the code review tool, other team members are notified that there is a new code change waiting for review. The other team members can remain focused on their own work until they finish their current task. They do not need to wait for others to join the review session but can start their own review whenever they want. In the code review tool, they can see the changed code snippets, leave review comments, and discuss the change via comments. Therefore, developers save time and suffer from fewer interruptions while getting almost the same benefits achieved with code inspection.

In addition, many recent studies reveal various benefits of modern code review beyond detecting potential defects. Developers can find better solutions for bug fixing or feature enhancement while discussing the code changes, and this discussion leads to improved code quality. Another benefit is knowledge sharing among team members. It takes considerable human effort to train a new developer. If the novice developer joins code reviews, however, she will learn the code base by watching discussions between senior developers. Additionally, when she makes a code change, experienced developers can give her constructive comments via the code review tools.

1.1 Modern Code Review Process

Figure 1.1 shows the overall process of modern code review supported by tools. Unlike the code inspection technique, modern code reviews are conducted using a variety of code review tools now available.

A developer has at least one of two major roles in modern code review: *change author* or *reviewer*. The *change author* in the figure makes a change to the original source code, which has been checked out from the repository. She submits her change — also referred to as a patch in many studies — to the code review tool. Usually, the change is committed to a temporary branch of the source code repository and not directly to the master branch. At the moment of the change submission, a continuous integration (CI) tool automatically conducts a verification check. The verification criteria used can vary between companies and organisations. Basically, most CI tools check for build and test failures. If the change causes an error in verification, the change author must rework the change until no error occurs during the verification step.

If the change successfully passes the verification check, another developer — the *reviewer* in the figure — manually inspects the change. Note that there is

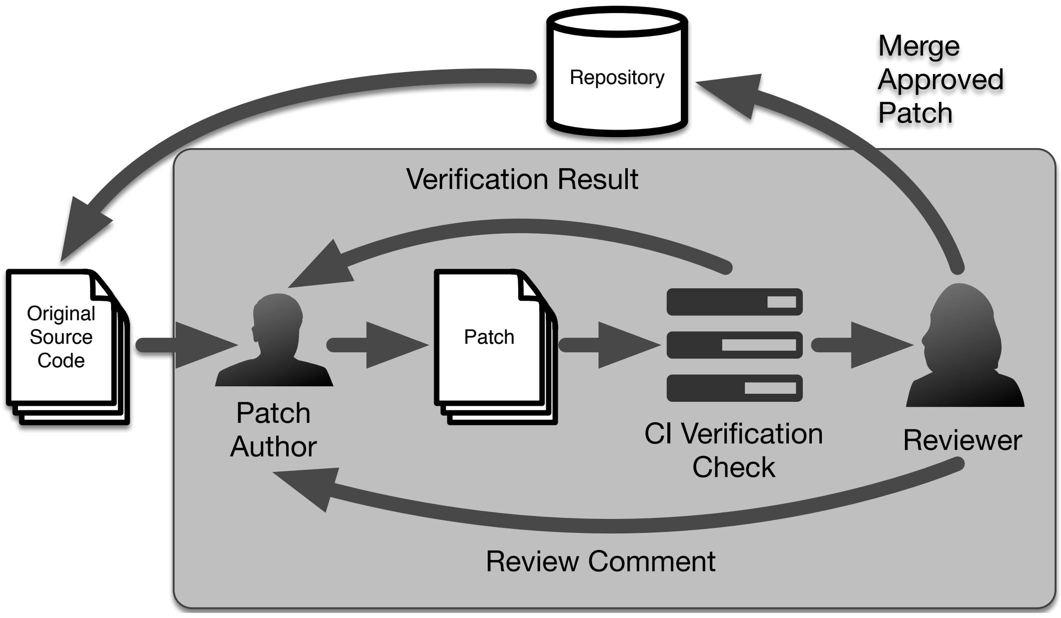


Figure 1.1: Modern code review process

usually more than one reviewer. These reviewers are generally experienced developers with more knowledge of the code change context than other developers. In the example, we assume that there is only one reviewer in the review process. The reviewer checks the change by considering functional and non-functional requirements (e.g. although the change may passed all the unit tests and CI checks, it can be rejected if it does not follow the coding conventions of the project). In addition, developers other than the reviewers and the change author can make additional comments during the review. If the change does not satisfy the requirements, the reviewer provides constructive comments about it. The change author then reworks it based on the reviewer's comment and repeats this process until the reviewer approves the change. Finally, the reviewer merges the approved change into the master branch of the repository.

1.2 Motivation

My PhD thesis has two ultimate goals:

First, the thesis seeks to understand developers' perspectives in code review. Comprehending what developers want and how they conduct code review is one of the most important objectives of empirical studies on code review. As code review becomes more popular, many empirical studies identify its benefits, but only a few studies are available that help us understand developers in a code review context.

The second goal is to determine how to support developers in code review. Although many empirical studies have reported on code review, not many techniques are available to support developers during the process. We must use the empirical knowledge gained from developers, to better support developers in code review.

1.3 List of Publications

During my PhD program, I have participated in and am currently working on the following papers as an author:

- DongGyun Han, Jens Krinke, Matheus Paixao, Chaiyong Ragkhitwetsagul, and Giovani Rosa, 'Pretty Patches: An Empirical Study of Coding Conventions During Code Review', submitted to 7th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), 2019 (under review; discussed in Chapter 4).
- Manal Alonaizan, DongGyun Han, Jens Krinke, Chaiyong, Ragkhitwetsagul, Daniel Schwartz-Narbonne, and Bill Zhu, 'Recommending Related Code Reviews', submitted to IEEE Transactions on Software Engineering (TSE), 2018 (major revision; discussed in Chapter 5).
- Matheus Paixao, Jens Krinke, DongGyun Han, and Mark Harman, 'CROP: Linking Code Reviews to Source Code Changes', in Proceedings of the 15th International Conference on Mining Software Repositories (MSR2018), Gothenburg, Sweden, 2018, Data Showcase.

- Matheus Paixao, Jens Krinke, DongGyun Han, Chaiyong Ragkhitwetsagul, and Mark Harman, 'Are Developers Aware of the Architectural Impact of Their Changes?', in Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017), Urban Champaign, Illinois, USA, 2017.

1.4 Expected Contributions

Improving current practice requires two steps: understanding the current issues and developing solutions for them. In this thesis, I present two empirical studies to improve our understanding of the current issues, and I propose an automated technique to improve the current practice.

One part of my work for my PhD thesis is a large-scale survey to understand how actual developers do code review in practice. The survey was conducted in open source communities (Eclipse and OpenStack). Since the survey reproduces some questions that appeared in previous publications, these results reveal differences between past and present. The survey results provide a better understanding of the code review practice in open source projects in terms of the developers' expectations and difficulties.

Another empirical study in this thesis highlights the coding convention issues during code review. The main goal of this second study is to learn what kinds of coding convention violations are detected during code review and how developers react to these violations. Although various tools support developers in detecting coding convention violations, we found that many patches are rejected because of such violations. This work extends our empirical knowledge by highlighting how developers handle coding conventions during code review and the difficulties they have in manually checking violations of those conventions.

Using the survey, I identified several difficulties developers face. To mitigate their pain during code review, I developed a technique to recommend related code reviews. The technique can provide more context for developers reviewing a change by recommending related previous code review requests.

1.5 Thesis Organisation

The remainder of the thesis is organised as follows. Chapter 2 surveys recent studies about code review. Chapter 3 presents results of the developer survey on code review. Chapter 4 reviews the empirical results on coding conventions used during code review. Chapter 5 describes the related review recommendation technique. Finally, Chapter 6 concludes the thesis and suggests the future work.

2 Literature Review

2.1 Surveys on Code Review

Bacchelli and Bird [2013] conducted a large-scale empirical study at Microsoft to learn the expectations, outcomes, and challenges of modern code review in practice. The authors interviewed developers to understand how they perform code review in practice. In addition, the authors manually inspected review comments extracted from CodeFlow, which is the internal code review tool at Microsoft. They conducted open card sorting to illustrate a mental model and derived categories from the interviews and code comment data without predefined categories. Based on the taxonomies extracted from open card sorting, the authors wrote a survey questionnaire. They sent the first survey, consisting of six questions, to 600 managers in the company; 165 (28%) responded. The authors also sent a second survey composed of 18 questions to 2,000 developers and received 873 (44%) answers. The survey results indicate that even though the primary motivation for code review was to find defects, developers thought code improvement, finding an alternative solutions, and knowledge sharing were also important motivations for conducting code review. The authors' comment analysis, however, shows most review comments are about code improvement, understanding, and social communication. Finally, the authors point out that knowledge transfer, increased team awareness, and creation of alternative solutions to problems are also expected outcomes of modern code review rather than simply the identification of defects.

Tao et al. [2014] investigated how to write acceptable patches to avoid review rejection and the need for resubmission. To this end, they manually analysed rejected patches from the Mozilla and Eclipse open source projects to derive review criteria used by developers in practice. They extracted 12 reasons for patch rejection. Based on this knowledge, they conducted a survey of Eclipse and Mozilla developers, asking how decisive the 12 patch rejection reasons were during code review. In addition, they asked how easy or difficult it was to identify the reasons for patch rejections. They reported that the most important reason for rejecting a patch was introducing new bugs; they also reported this was the most difficult problem to identify.

Kononenko et al. [2016] surveyed the ways in which developers perceive code review quality. They sent a questionnaire to 88 Mozilla developers. The authors observed that the review quality is associated with the thoroughness of the feedback, the reviewer's expertise with the code base, and the perceived quality of the code itself.

Bosu et al. [2017] conducted surveys of open source developers and developers from Microsoft to comprehend the process and the benefit of code review. The authors discovered that developers spent about 10 to 15 percent of their time in code reviews. In addition, developers spent more time in code review as they became more experienced. They reported that open source developers and Microsoft developers gave similar answers on their survey in general.

Gousios et al. [2015] investigated the integrator's role within a pull-based development. They surveyed 749 integrators to comprehend their practices and challenges. The authors argued that the integrators have difficulties in prioritising code changes to be merged.

Baum et al. [2017] studied the optimal order of reading source code changes. They conducted interviews and a survey and investigated the code review history of a company. Based on the diverse dimensions of study, the authors came up with six principles to organise source code changes. In particular, they argued that optimal grouping of the changed sections by relatedness is the most important.

2.2 Coding Conventions During Code Review

Panichella et al. [2015] investigated how developers handle static analysis results such as coding convention violations during code review. While their approach to the analysis is very similar to that we use in research presented in Chapter 4, their evaluation was limited and mainly focussed on quantitative analysis. While they manually investigated only a small sample of candidate reviews, we manually investigated all reviews. Moreover, they concluded that static analysis tools can be used to support developers during code reviews. In contrast, our analysis demonstrated that developers were not effective — and more importantly, not consistent — in detecting violations, suggesting that automated checking (and fixing) should be used to reduce the burden on reviewers and make the code review more rigorous in terms of catching violations that developers actually care about. We focused in detail on what violations are inserted and removed during the review, improving on the diversity of those already present.

Balachandran [2013] suggested the Review Bot, which is an extension of the Review Board and can recommend appropriate reviewers for a submitted review issue. The Review Bot uses line-level change history to determine the proper reviewer. The author evaluated his approach by using proprietary data from VMware. Since there were no previous studies on reviewer recommendation, the author compared the recommendation result with the result extracted via a different level of granularity (i.e. file-level change history). When the Review Bot recommended only one reviewer, its reported accuracy was 59.92%-61.17%. When it recommended five reviewers, it showed an 80.85%-92.31% accuracy rate.

Similar to Balachandran's work, Henley et al. [2018] integrated a CI tool, CloudBuild [Esfahani et al., 2016], that covers builds, test, and code analysis within a code review tool, CodeFlow. The authors showed that integrating static analysis tools within the code review leads to more communication between developers. In particular, their technique increased discussions of coding conventions by about 50%.

Singh et al. [2017] found that PMD can reduce the workload of code reviewers. Beller et al. [2016] empirically studied how developers leverage static analysis tools during code review, reporting that such tools are widely adopted in projects, but their use is not strictly enforced.

Czerwonka et al. [2015] reported on experience within Microsoft about benefits and costs of reviewing practices. They discussed the high cost of code reviews and the fact that reviews are not always used efficiently.

Vassallo et al. [2018] investigated developers in both industry and open source projects who use static analysis tools. They observed that developers configure static analysers at least once, and the configuration is rarely changed during a project. They also stated that developers assign different priorities to warnings generated by static analysers based on different contexts. Zampetti et al. [2017] empirically investigated the integration of static analysis tools and CIs. They found that a failure caused by a convention checker is one of the main reasons for build failure. Sarkar and Parnin [2017] argued that human efforts imply mental fatigue, which causes an increase in coding convention violations.

Smit et al. [2011] examined whether convention adherence is a proxy measurement for maintainability. They observed that adopting coding convention checking tools does not lead to a reduction in the number of violations. On the other hand, Java programmers find it difficult to comply with coding conventions, as a study by Elish and Offutt [2002] has demonstrated.

2.3 Recommendation Techniques for Code Review

Thongtanunam et al. [2014, 2015] introduced a reviewer recommendation system based on file location information. The authors asserted the importance of the reviewer recommendation problem by showing 4%–30% of reviews suffered a code reviewer assignment problem. Notably, large systems have more difficulties in finding appropriate reviewers than comparable small systems. They assume that files that have similar paths will be managed and reviewed by a similarly experienced developer. Based on this assumption, the authors proposed a file-path-based code reviewer recommendation system, RevFinder. They demonstrated that RevFinder correctly recommended 79% of reviewers (top-10 recommendation) and outperformed the Review Bot proposed by Balachandran [2013].

Ouni et al. [2016] proposed a search-based code reviewer recommendation technique called RevRec. Specifically, the authors adopted a genetic algorithm to find reviewers using previous reviews. They claimed that their approach could achieve 59% precision and 74% recall.

Zanjani et al. [2016] proposed another reviewer recommendation approach called cHRev based on code review history data. The authors argued that the specific information from previously resolved code reviews (i.e. the quantification of review comments and their recency) dramatically improved the recommendation performance.

Hannebauer et al. [2016] suggested a reviewer recommendation technique based on developers' expertise. They used six algorithms based on code change expertise and two algorithms based on code review expertise. The authors claimed that their technique could recommend at least one out of five reviewers correctly in 69%–75% of all cases.

2.4 Conclusions

In the 40 years since Fagan first proposed the code inspection technique, many code review studies have been conducted. Recently, the code review paradigm has shifted from formal code inspection to modern code review, which is a lightweight and asynchronous technique. With the emergence of modern code review, many developers and researchers became interested in studying its performance. Numerous empirical studies have been published, and diverse modern code review tools have been released. Through these advances, developers could gain many benefits. However, recent research has been heavily biased towards empirical studies of modern code review. Even though empirical studies are important and can provide valuable insights for practitioners, there has not been sufficient consideration for direct support of developers in modern code review. This calls for more work on code review automation and tool support.

3 How Do Developers Conduct Code Review? An Empirical Study of Code Review Practice in Open Source Projects[1]

3.1 Introduction

Fagan [1976] proposed the traditional code inspection technique to detect potential defects at an early stage. Fagan's code inspection technique is a formal review process based on an off-line meeting of developers in which they discuss whether the code is buggy or how to improve it. Since it was first proposed, a series of studies have observed the benefits of code inspection, including finding defects and reducing testing time [Fagan, 1986, Ackerman et al., 1989, Siy and Votta, 2001]. On the other hand, it has also been observed that this traditional approach to code review consumes significant development resources because of its formality and its synchronous nature [Votta, 1993].

Modern code review is differentiated from the traditional code inspection technique by its lightweight and asynchronous characteristics while preserving the benefits of the code inspection technique [Bacchelli and Bird, 2013]. Modern code review is conducted using a code review tool asynchronously (i.e. developers can review their colleagues' changes at their own convenience) rather than relying on an offline code inspection meeting. Because of

[1]This work has done with Jens Krinke and Federica Sarro at University College London and Daniel Schwartz-Narbonne at Amazon Web Services.

modern code review's benefits and efficiency, it has been widely adopted in both open source [Bosu and Carver, 2013, Tao et al., 2014] and proprietary projects [Feitelson et al., 2013, Bosu et al., 2015, Sadowski et al., 2018].

Given that the use of code review is increasing, we need to improve the current process to mitigate the burden it puts on developers. A three-step approach has been widely adopted by empirical researchers to improve a process or practice [Wohlin et al., 2003]. First, we need to understand the current practice. Based on a proper understanding of the current practice, we can derive potential improvement opportunities. Second, we must evaluate whether the new idea from the first step contributes to improving the current practice. Lastly, we can improve the current process by adopting the suggestions from the evaluation. In this chapter, we conduct a large-scale survey to evaluate developers' current code review practice and to better understand existing code review process.

Many researchers have conducted survey surveys to understand modern code review [Bacchelli and Bird, 2013, Kononenko et al., 2016, 2018, Bosu et al., 2017]. However, their findings are limited to the subjects they have investigated, and it is not clear whether the findings apply to other subjects. Different open source projects and proprietary projects can have different characteristics, and the conclusions from the previous studies may not be applicable to all projects. Therefore, reproducing existing studies can extend empirical knowledge and provide a better understanding by confirming the previous results with different subjects or by reporting different results from different subjects. We carried out a survey composed of 21 questions informed by developers and previous surveys on the same topic. To compare our new results to those of previous surveys, we need to get data from those earlier studies. With the original authors' support, we gained access to the survey results of Bacchelli and Bird [2013] and Tao et al. [2014] and reproduced the questions used in their studies.

This chapter presents the following contributions:

- A large-scale survey of code review within open source projects (Eclipse and OpenStack)
- A comparison between developers' perceptions and reality in terms of review time

- A reproduction of previous surveys in open source projects

The remainder of this chapter is organised as follows: Section 3.2 explains the previous survey studies we reproduced in this chapter. Section 3.3 describes our survey process. The results of the survey are presented in Section 3.4. We discuss these results in Section 3.5. Section 3.6 reviews the threats to the validity for our work, and Section 3.7 concludes the chapter.

3.2 Previous Surveys

In this chapter, we reproduce survey questions from two previous studies [Bacchelli and Bird, 2013, Tao et al., 2014]. Here We review these earlier studies for better understanding of the reproduced questions.

Bacchelli and Bird [2013] reported developers' expectations, outcomes, and challenges during code review via an empirical study within Microsoft. They conducted a survey to identify the motivations for code review are in the company and reported that developers are interested in finding defects and improving code. Moreover, the authors manually inspected code review comments stored in CodeFlow, Microsoft's internal code review tool. They concluded that code review has diverse benefits, including knowledge transfer and increased team awareness, although they found defect cases reported to be fewer than expected.

Tao et al. [2014] investigated how to write acceptable patches to avoid rejection and resubmission. To this end, they manually investigated rejected patches from the Mozilla and Eclipse open-source projects to derive review criteria that are used by developers in practice. Through a manual investigation, they extracted 12 reasons for patch rejection. Based on their results, they surveyed Eclipse and Mozilla developers, asking how decisive each of the 12 patch rejection reasons was during code review. They also asked how easy or difficult it was to determine the reason a patch was rejected. They reported that the fact that a patch was introducing new bugs was the most important reason for rejecting while also being the most difficult problem to identify in a patch.

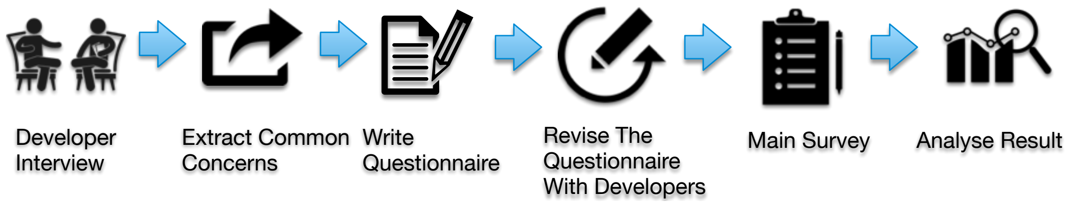


Figure 3.1: Overall process of the survey

3.3 Methodology

The survey is consisted of two parts. The first part aimed to understand developers' concerns with code review. Although many surveys have been conducted, as we stated in Section 3.1, we identified interesting questions about code review that have not yet been answered by researchers. The second part reproduced questions from surveys by Bacchelli and Bird [2013] and Tao et al. [2014].

Figure 3.1 illustrates the process we followed to design the survey. To extract the developers' concerns, we conducted semi-structured interviews. We did not use a specific question set, but allowed the questions to evolve through repeated interviews. We followed the interview process of Bacchelli and Bird [2013]. We interviewed 12 developers[2], and each whom spent 30 minutes with the author of this thesis. All interviews were recorded with the interviewee's consent. Once we had interviewed 10 developers, we reached the saturation point (i.e. no new issue was identified).

We performed open coding on the recorded interview transcript to extract common concerns. The author of this thesis split the recorded interview script into topics (i.e. single words or short sequences of words), then grouped similar topics into categories and assigned a representative name to each category. These topics and categories were reviewed by the collaborators and revised based on that review to mitigate subjective bias . The resulting questionnaire consisted of 16 questions under four categories: demographics, reviewer selection, review time, and consulting other reviews.

[2]Due to the company's policy, we can not present the details of the developers here.

Table 3.1: Questions in the questionnaire

Category	Answer Type	Question
Demographics	Multi-choice	Q1. Which best describes your primary work area?
	Open-ended	Q2. If your primary work areas are not listed above, please list them here
	Numeric	Q3. How many years of software development experience do you have?
	Numeric	Q4. How many years have you been a member of your current team / open source project?
	Numeric	Q5. How many members are there in your project (including yourself)
	Yes-No	Q6. Within your project, is it mandatory to perform code review?
	Multi-choice	Q7. Which of the following languages do you typically work with?
	Open-ended	Q8. If you use other language that are not listed above, please list them here
Reviewer Selection	Multi-choice	Q9. Whom do you prefer to send your code review?
	Open-ended	Q10. If you have another criteria for selecting a reviewer, please list them below
Review Time	Choice	Q11. Based on your expectation, what is the ideal time to review a review request?
	Numeric	Q12. Based on your expectation, a review request should be resolved (finally merged/abandoned) within how many days in general?
	Numeric	Q13. Based on your expectation, a review request should be resolved within how many revisions in general?
Consulting Other Reviews	Yes-No	Q14. Do you ever consult other/earlier reviews when doing a review?
	Open-ended	Q15. What are the reasons you consult other/earlier reviews?
	Likert Scale	Q16. In the context of carrying out code review, how much do you agree or disagree with the following statements?
Motivation [Bacchelli and Bird, 2013]	Top3	Q17. What are your motivations for code review? Please pick three most important ones (in order, without ties) from the following list (choice per items: First most important, second most important, third most important, not top 3 importance)
	Open-ended	Q18. If you have other motivation for code review that are in the top three but are not listed above, please list them here.
Importance & Difficulty [Tao et al., 2014]	Likert Scale	Q19. On the scale provided, how important is it to identify these common problems during code review?
	Likert Scale	Q20. How easy or difficult is it to identify these problems during code review?
	Open-ended	Q21. Please specify other criteria to evaluate the patches, if you have any.

In the demographics category, we gathered the basic information from the survey participants, including their experience and work area. The reviewer selection category asked developers how they select a reviewer. The review time category questions asked developers about their expectation in terms of review time. Finally, the consulting other reviews section asked developers whether they had consulted earlier reviews when reviewing a new change.

We also added five questions in two categories that had already been reported in previous papers (in different projects): motivation and importance and difficulty. The motivation category is taken from Bacchelli and Bird [2013]. As we discussed in Section 3.2, they queried Microsoft developers about their most important motivations for conducting code review. We used the same question statement and options used by Bacchelli and Bird [2013]. The importance and difficulty category is a reproduction from Tao et al. [2014]. They extracted code review criteria and asked Eclipse and Mozilla developers how important each criterion was and how easy or difficult it was to get information to evaluate the criterion during review. Since they had conducted their survey in 2012, we could observe the trend changes during the last 6 years by asking the same questions.

Based on the above categories, we drafted an initial questionnaire. We shared the draft with 10 stakeholders, including managers, researchers, and engineers, and asked them to provide feedback. Based on their feedback, we revised the questionnaire.

Table 3.1 presents the questions of the final revised questionnaire. We set different answer types for each question to ease the survey process. Multi-choice questions provided multiple options for a participant to select among; open-ended questions allowed the participants to provide their answers or opinions in complete phrases. Since the answers for open-ended questions were written in natural language, the author of this thesis qualitatively analysed these answers, and the collaborators reviewed the analysis to mitigate subjective bias; Numeric questions allowed numeric answers only. For Yes-No questions, the participants were asked to answer either yes or no. Likert Scale questions provided answer options along a 5-point Likert scale. For each question, we used different labels for the 5-point Likert scale. To reproduce the motivation question from Bacchelli and Bird [2013], we also used Top-3 type questions in which participants could select their top 3 elements from among the options.

We selected developers from the code review systems in Eclipse and OpenStack and asked them to complete the survey. To minimise unreachable emails and filter out inactive developers, we targeted developers who have submitted at least one comment within the month preceding the survey start date. We were able to extract 2,144 developers from Eclipse and OpenStack projects and sent them a link to our questionnaire via email. Among them, 110 developers were not reachable (i.e. we received auto-reply, invalid email, or vacation responses). We opened the questionnaire link for 10 days. On the fifth day of the survey, we sent a reminder to encourage developers to answer the survey. In the end, 100 open source developers provided answers.

3.4 Result

We present the results of the survey category by category in this section.

3.4.1 Demographics

The survey answers can vary between different groups of people. Some answers may be a product of the unique environment of participants. For example, C developers may be more concerned than Java developers about memory management. Therefore, it is important to understand the participants' demographics. We asked seven questions in the demographics category designed to provide information on the participants' experience and work environment. Our demographics questions and the initial options for those questions were inspired by Bacchelli and Bird [2013].

Although we selected survey participants who had left at least one comment in the code review tool, participants' primary role could be other than development role. Therefore we asked 'Q1. Which best describes your primary work area?'. Figure 3.2 shows the answers to this question. The question allowed developers to select more than one answer, since a participant can have more than one primary work area in a project (e.g. a manager may manage a team

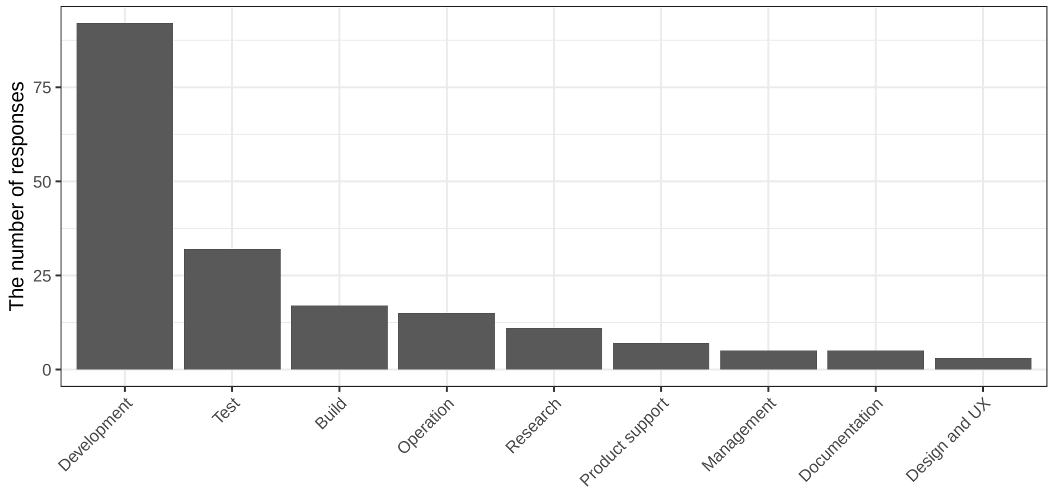


Figure 3.2: Q1. Primary work area of the survey respondents

while also implementing code). We provided selection options based on the pre-defined major work areas in software development. The majority of participants answered that their primary work area was Development (90 responses) followed by Testing (31 responses). The results indicate that Eclipse and OpenStack developers participate in diverse work areas, such as testing and operation.

Because survey participants may have had a work area other than the categories we provided in the question, we also included 'Q2. If your primary work areas are not listed above, please list them here'. Only one participant answered this question with 'Consulting'.

Q1-2. The majority of the survey participants from Eclipse and OpenStack projects work on development. Eclipse and OpenStack developers participate in diverse work other than development.

We asked 'Q3. How many years of software development experience do you have?' because developers' experience level can affect their answers. In addition, we also asked 'Q4. How many years have you been a member of your current team / open source project?' to determine participants' experience level with their

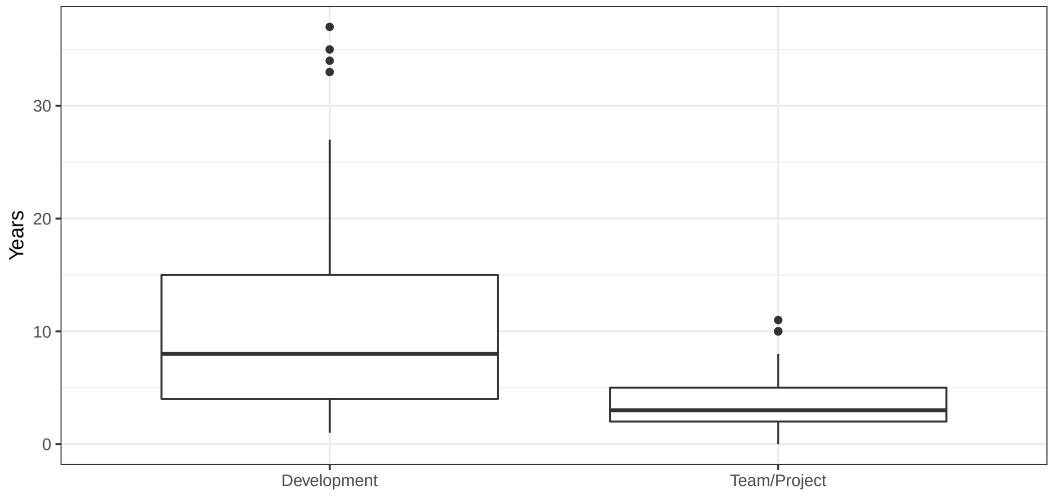


Figure 3.3: Q3 and 4. Overall development experience and experience within the current team or project

current team or project. Figure 3.3 presents results on participants' general development experience and their experience in their current company or project. The left side shows participants' entire development experience, while the right side presents their experience with their current team or project. The median length of development experience is 8 years. The majority of the Eclipse and OpenStack developers have from 4 years to 15 years of development experience.

The team or project experience duration is shorter than the overall development experience, as developers have experience in multiple teams or projects. The responses show a narrow experience distribution. The majority of the Eclipse and OpenStack developers have 2 to 5 years of experience with their current project.

Q3-4. The majority of developers have about 8 years of development experience and have about 2 to 5 years of experience within the current team/project.

The size of a team (i.e. the number of colleagues on a team) may affect the participants' code review experience. Therefore, we asked 'Q5. How many

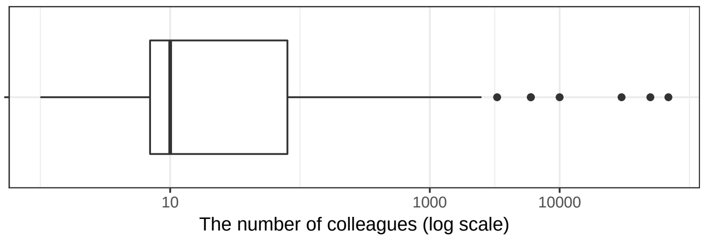


Figure 3.4: Q5. Number of colleagues on a team or project

members are there in your team / project (including yourself)?'. Figure 3.4 displays the responses. Due to the wide range of the answers, we used a log scale to visualise the results. The median number of team members from Eclipse and OpenStack is 10, although some extreme outliers show teams larger than 1,000. The results may reflect that the Eclipse and OpenStack projects invite as many contributors as possible.

Q5. Most teams in the Eclipse and OpenStack projects have a large variation in size (i.e. the number of developers).

Although code review is a widely adopted technique for maintaining software quality, it may not be a mandatory process for the teams and projects. Therefore, we asked 'Q6. Within in your team/project, is it mandatory to perform code review?' Participants were asked to answer yes or no; 82 answered yes. The results indicate that code review is a mandatory and widely used technique in Eclipse and OpenStack communities.

Q6. Code review is mandatory in Eclipse and OpenStack projects.

The last question in the demographics category asked about the kinds of programming languages participants used (i.e. 'Q7. Which of the following languages do you typically work with?'). Since developers can use more than one language in a project, we allowed participants to check multiple answers for this question. Figure 3.5 graphs the languages that the developers used. The options for the question were extracted from the language popularity

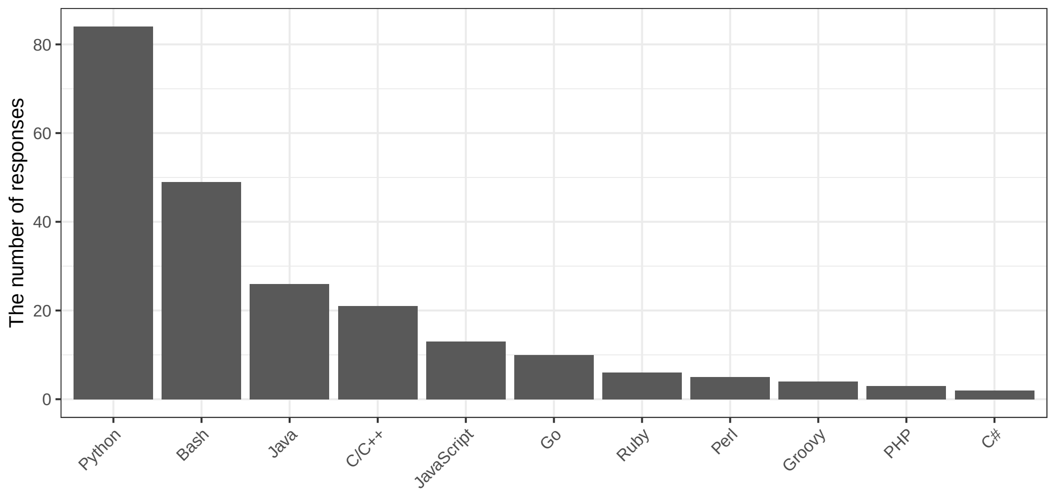


Figure 3.5: Q7. Programming languages the respondents use the most

rank.[3] The most popular programming language is Python in the Eclipse and OpenStack projects (82 answers). Please note that the programming language is strongly dependent on the project type: Eclipse projects are mainly Java projects, whereas OpenStack projects are mainly Python projects.

Since developers can use a programming language other than the ones specified in the above question, we also asked the following: 'Q8. If you use other language that are not listed above, please list them here'. Among Eclipse and OpenStack developers, Ansible (4 answers) was the most popular language used other than the ones specified in the list for Q7. The Eclipse and OpenStack developers also answered that they use Jinja2, yaml, Puppet, and Common Lisp as well.

Q7-8. The most popular languages in the Eclipse and OpenStack projects are Python, Bash, and Java. Other than the popular languages, developers use diverse languages in their work.

In this section, we reported the demographics of the survey respondents. Work experience, size of a development team (i.e. the number of members

[3]TIOBE index (https://www.tiobe.com/tiobe-index)

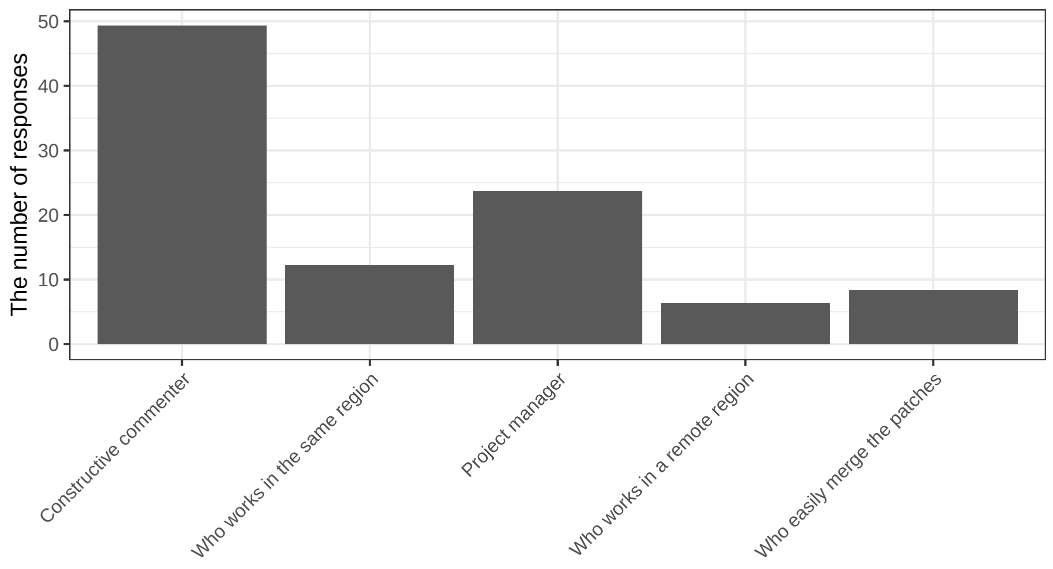


Figure 3.6: Q9. Reviewer preference of the respondents

on a team or project), and programming language may affect the answers reported in the following sections.

3.4.2 Reviewer Selection

One of the developers' concerns we extracted from the preliminary interviews was how to find a proper reviewer when requesting a review. Although diverse automated reviewer recommendation techniques [Balachandran, 2013, Thongtanunam et al., 2015, Rahman et al., 2016] have been proposed, we could not see that any of them had been adopted in the studied organisations. We asked, 'Q9. Whom do you prefer to send your code review to?'.

Figure 3.6 shows the results from the Eclipse and OpenStack projects. We extracted the selection options from the preliminary interviews. The most popular reviewer type in the Eclipse and OpenStack projects is a reviewer who provides a constructive review comment (i.e. constructive commenter).

Of the Eclipse and OpenStack developers, 24 assigned a code review request to their managers.

Q9. Developers prefer to send a code review request to a reviewer who provides them with constructive comments.

Since the selection options for the above question might miss a developer's concerns, we also requested the following: 'Q10. If you have another criteria for selecting a reviewer, please list them below'.

We received 11 answers to this question. Since various stakeholders participate in Eclipse and OpenStack projects, the developers try not to be biased to a specific stakeholder. Therefore they prefer a reviewer who can review a potential bias for a specific stakeholder. In addition, a reviewer who is an expert in security is preferred.

Q9-10. Other than a constructive reviewer, developers also prefer a reviewer who can review a potential bias for a specific stakeholder and provide security related comments.

As presented in this section, the majority of developers prefer to send a code review request to a reviewer who can provide constructive comments. Although the constructive reviewer is the most preferred reviewer type, developers also have other ideal reviewer type (e.g. a reviewer who has security expertise).

3.4.3 Review Time

Another of the developers' concerns we found during the preliminary interviews was that the amount of time needed to perform code review was excessive. Therefore, we included three questions related to this aspect (i.e. Q11, Q12, and Q13). Moreover, we compared the expectations identified through the questionnaire to the actual time taken which we extracted from the code review data.

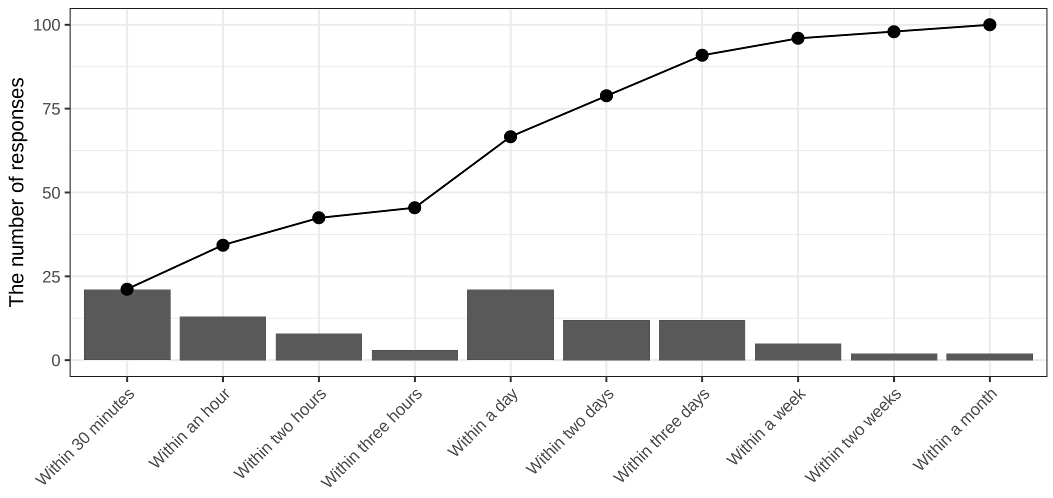


Figure 3.7: Q11. The respondents' expectation on review time

First, we asked 'Q11. Based on your expectation, what is the ideal time to review a review request?'. This question was designed to determine the expected review time for a revision of a code review request. We provide pre-defined options for this question: within 30 minutes, within an hour, within two hours, within three hours, within a day, within two days, within three days, within a week, within two weeks, and within a month. Figure 3.7 shows the expected review time for a revision. The bars present the ratio of each category to total number of answers, while the line shows the accumulated ratio. Of the Eclipse and OpenStack developers, 22 expected to receive a review within 30 minutes, while another 22 expected it within a day. The predominant number of the Eclipse and OpenStack developers expected that reviews should be done within a day (i.e. 67 responses).

Q11. The majority of the developers from Eclipse and OpenStack expected that their code review request receive a review within a day or less.

Second, we asked 'Q12. Based on your expectation, a review request should be resolved (finally merged/abandoned) within how many days in general?'. This question sought to determine the expected overall time to resolve a code review request.

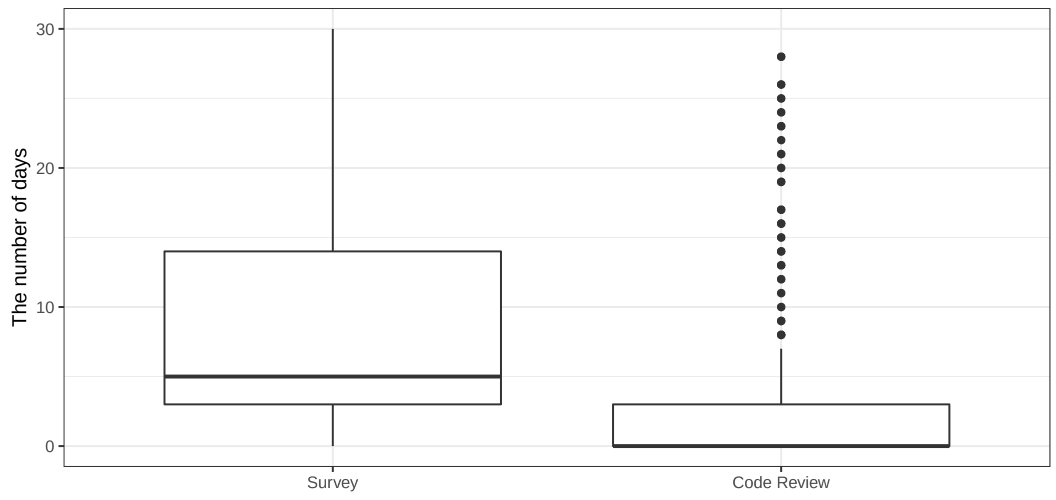


Figure 3.8: Q12. Expected time to resolve a code review request and actual time from code review repository data

Figure 3.8 shows the developers' expectations for the number of days needed to resolve a code review request (survey in the figure) and the actual number of days taken to resolve code review requests in the code review system (code review in the figure). The results indicate that the Eclipse and OpenStack developers' expectation was 5 days on average. The data show that the most review requests were resolved more quickly than the developers' expected. Most code review requests were resolved within a day (i.e. the median value was less than a day).

Q12. The majority of code review requests were resolved within a day in Eclipse and OpenStack projects which was shorter than the developers' expectation.

Lastly, we asked 'Q13. Based on your expectation, a review request should be resolved within how many revisions in general?'. This question asked about the expected number of iterations during code review. Figure 3.9 graphs the developers' expectations and the actual number of revisions to resolve a code review request in the code review system. The majority of the Eclipse and OpenStack developers expected that a code review request should be

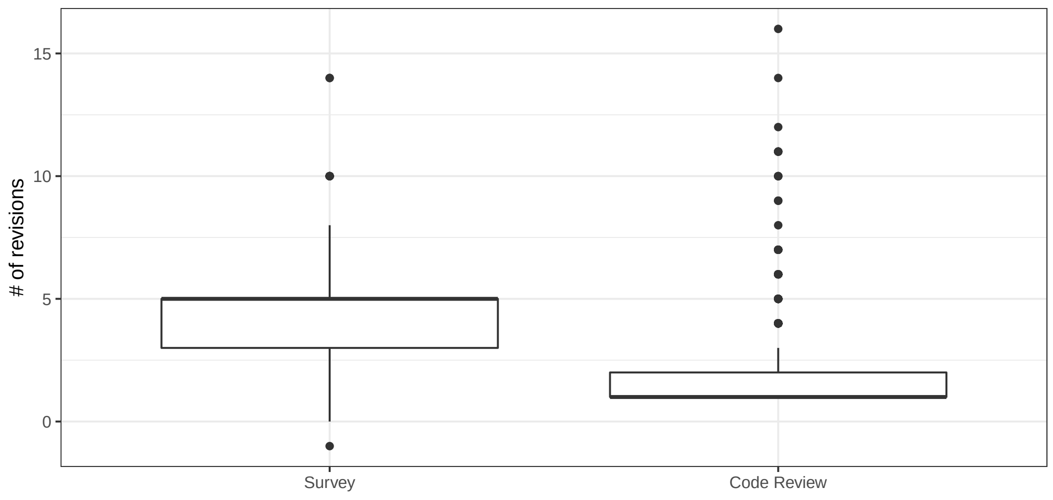


Figure 3.9: Q13. Expected number of revisions to resolve a code review request and actual number of revisions from code review repository data

resolved within five revisions in general. Interestingly, the actual number of revisions was generally lower than what developers expected. The data shows that most of code review requests were resolved within a revision.

Q13. The developers expected that their code review requests should be resolved within five iterations while the actual code reviews are mostly resolved within a single revision.

In this section, we compared the developers' perceptions of desirable code review times with the actual code review times. The Eclipse and OpenStack developers answered that they wanted to get a response from the reviewers within a day. Fortunately, the actual code review data showed that the majority of code review requests get a review within a day and with only one or two iterations.

Table 3.2: Q15. Reasons why developers consult earlier reviews

Category	# of responses
To get context	36
To learn	13
To evaluate a patch	11
To check duplication	11
To extract criteria	3
To increase review speed	2
To find a reviewer	1

3.4.4 Consulting Other Reviews

During the preliminary interviews, we found that developers frequently consulted previous code review requests when reviewing a new one. Therefore, we asked questions regarding consulting other reviews.

The first question was 'Q14. Do you ever consult other/earlier reviews when doing a review?'. This question had two answer choices: yes or no. In total, 81 of the Eclipse and OpenStack developers answered 'yes'.

Q14. The majority of developers have consulted other/earlier code review requests to review a new one.

Why do developers consult earlier reviews? We put the open-ended question in the questionnaire: 'What are the reasons you consult other/earlier reviews?'. We got 63 answers. The authors performed open coding on these answers. After extracting representative words and conducting axial coding to categorise the answers, we extracted seven categories for why developers consult other reviews: to get context, to evaluate a patch, to extract criteria, to learn, to check duplication, to increase review speed, and to find a reviewer. Table 3.2 shows the number of answers for each category. Please note that an answer can be categorised into more than a category (i.e. an answer might be categorised into both 'to get context' and 'to evaluate a patch'). The most frequently cited category was to get context; the second most frequent was to learn.

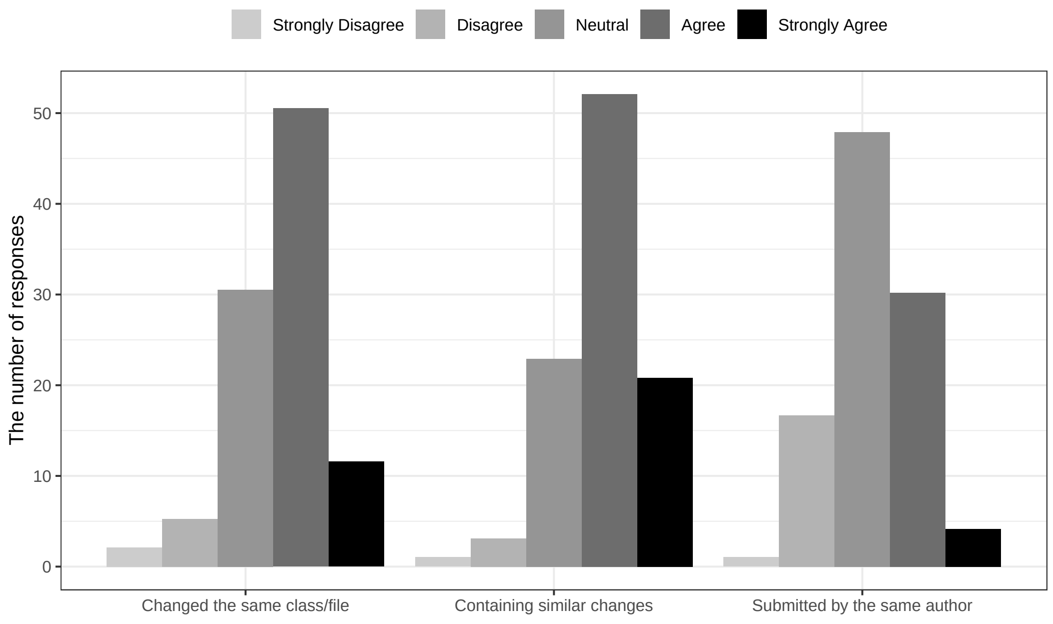


Figure 3.10: Q16. Level of agreement with three statements regarding the helpfulness of consulting other reviews

Q15. The result shows that developers can get diverse benefits from consulting other/earlier reviews. The most popular reason for consulting other/earlier review is to get context followed by to learn.

To determine what kind of information from earlier code review requests is helpful, we asked 'Q16. In the context of carrying out code review, how much do you agree or disagree with the following statements?'. We provided the three statements with answer choices on a 5-point Likert scale (i.e. strongly disagree, disagree, neutral, agree, and strongly agree):

- Previous reviews that changed the same class/file will be helpful.
- Previous reviews containing similar changes will be helpful.
- Previous reviews submitted by the same author will be helpful.

Figure 3.10 illustrates the ratio of answers on the 5-point Likert scale for each item. Agree is the dominant answer for changed same class/file and containing

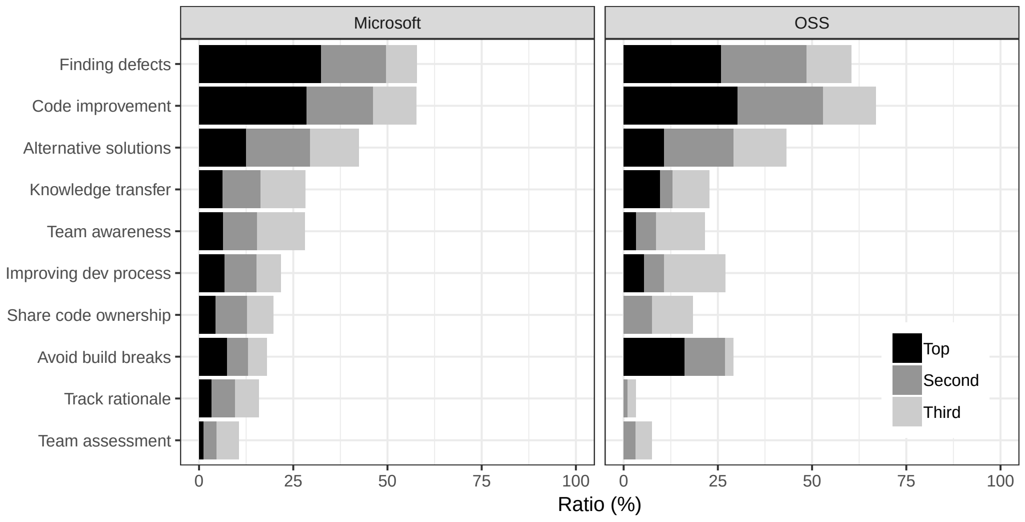


Figure 3.11: Q17. Developers' top three motivations for code review

similar changes, while most of the developers selected neutral for the submitted by the same author category. This may indicate that the developers think previous reviews by the same author contain less useful information than do the actual changes (i.e. changes in the same file or similar changes).

Q16. The Eclipse and OpenStack developers are interested in the other/earlier code review requests that contain either changes in the same class/file or similar changes.

As reported in this section, developers answered that consulting previous reviews has diverse benefits, providing development context and review criteria. In addition, developers could learn their team's code base from the other reviews [Bacchelli and Bird, 2013].

3.4.5 Motivation

Bacchelli and Bird [2013] asked the following question to Microsoft developers, 'Q17. What are your motivations for code review? Please pick three most important ones (in order, without ties) from the following list (choice per items: First most important, second most important, third most important, not top 3 importance)'. We reproduced the same question for the Eclipse and OpenStack projects to check the difference between Microsoft and the Eclipse and OpenStack projects. Figure 3.11 shows the results for the question from Microsoft and the Eclipse and OpenStack developers. Please note that the Microsoft results were presented in Bacchelli and Bird [2013]. Since the organisations have different numbers of survey participants, we use ratio to visualise the results to allow for an easy comparison between the organisations.

Overall, the organisations' developers provided similar answers, as presented in the figure. Most participants answered that finding defects and code improvement were the most important motivations for conducting code review. Microsoft developers identified finding defects as the most important motivation, while Eclipse and OpenStack respondents identified code improvement as the most important motivation. We found a peak in the Eclipse and OpenStack answers for avoid build breaks. This may suggest the Eclipse and OpenStack developers are suffering more from build breakages than are industrial developers.

Q17. Finding defects and code improvement are the most important motivations for code review in both Microsoft and the Eclipse and OpenStack projects.

3.4.6 Importance and Accessibility of Review Criteria

Tao et al. [2014] investigated what kinds of review criteria were important to developers and how easy or difficult was it to check those criteria during code review. We reproduced the same four questions Tao et al. [2014] asked in their paper.

The first question was 'Q19. How important is it to identify these common problems during code review?'. As Tao et al. [2014] did, we asked developers to answer the question using a 5-point Likert scale: Unimportant, Of Little Importance, Moderately Important, Important, and Very Important. Figure 3.12 presents the results from the current survey (i.e. OSS) and the results from Tao

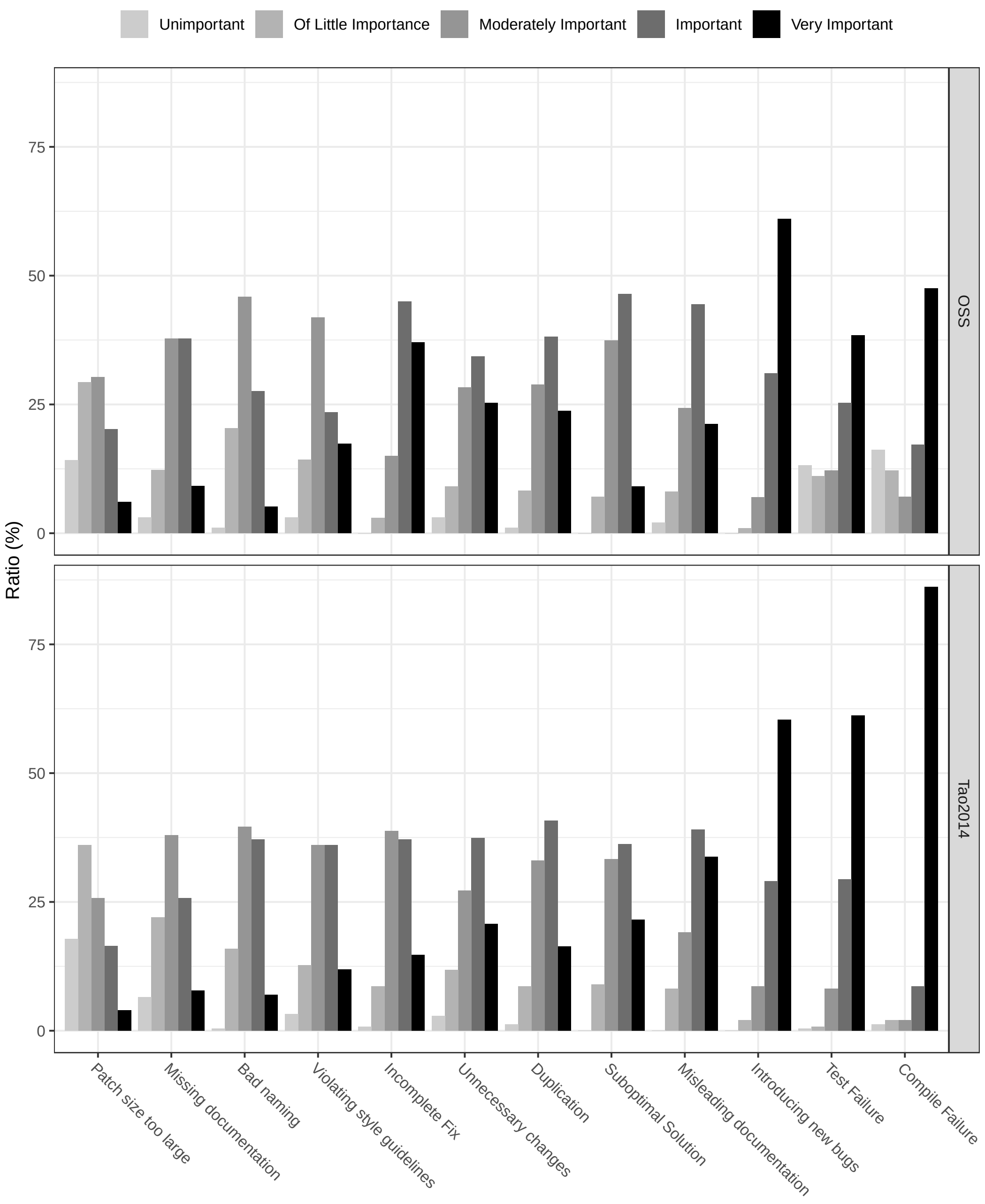


Figure 3.12: Q19. How important it is to identify each problem during code review

et al. [2014] (i.e. Tao2014). Overall, introducing new bugs is the criterion that developers think the most important to check for during code review. It is interesting that compile failure was the most important criterion in Tao et al. [2014], whereas it is comparably less important in the recent results. One potential reason is that the wide adoption of CI relieves the developers' burden in checking for compile failure during code review.

Responses to the incomplete fix criterion differ notably between the surveys. This may denote that the developers' perceptions regarding incomplete fixes have changed within open source projects. In Tao2014, the majority of developers rated incomplete fix as moderately important or important. However, the recent results are more weighted to very important.

Ratings for Compile failure, test failure, and misleading documentation also differ notably between the past and the current surveys. Both compile failure and test failure declined as concerns from Tao2014 to OSS. This may suggest that CI now can relieve the developers' burden of checking compilation failure and test failure manually. OSS results show a greater concern for misleading documentation than do the Tao2014 results. This change may represent higher requirements for accurate documentation in open source communities.

Q19. Detecting new bugs introduced in a change and incomplete fix are the most important issues during code review in both the past and present surveys. Despite that complication and test failure can and should be identified through CI before code review, it is interesting to see that they are still considered important problems during code review.

The second reproduced question was 'Q20. How easy or difficult is it to identify these problems during code review?'. As with the importance question, we used a 5-point Likert scale, but we attached different labels: Very Easy, Easy, Moderate, Difficult, Very Difficult. Figure 3.13 displays the results for this question. The developers indicated that compile failure, test failure, and patch size too large were relatively easy to identify during code review. In all results, the most difficult criterion was introducing new bugs.

The OSS and Tao2014 results differ for only three criteria: compile failure, missing documentation, and patch size too large. These differences might provide

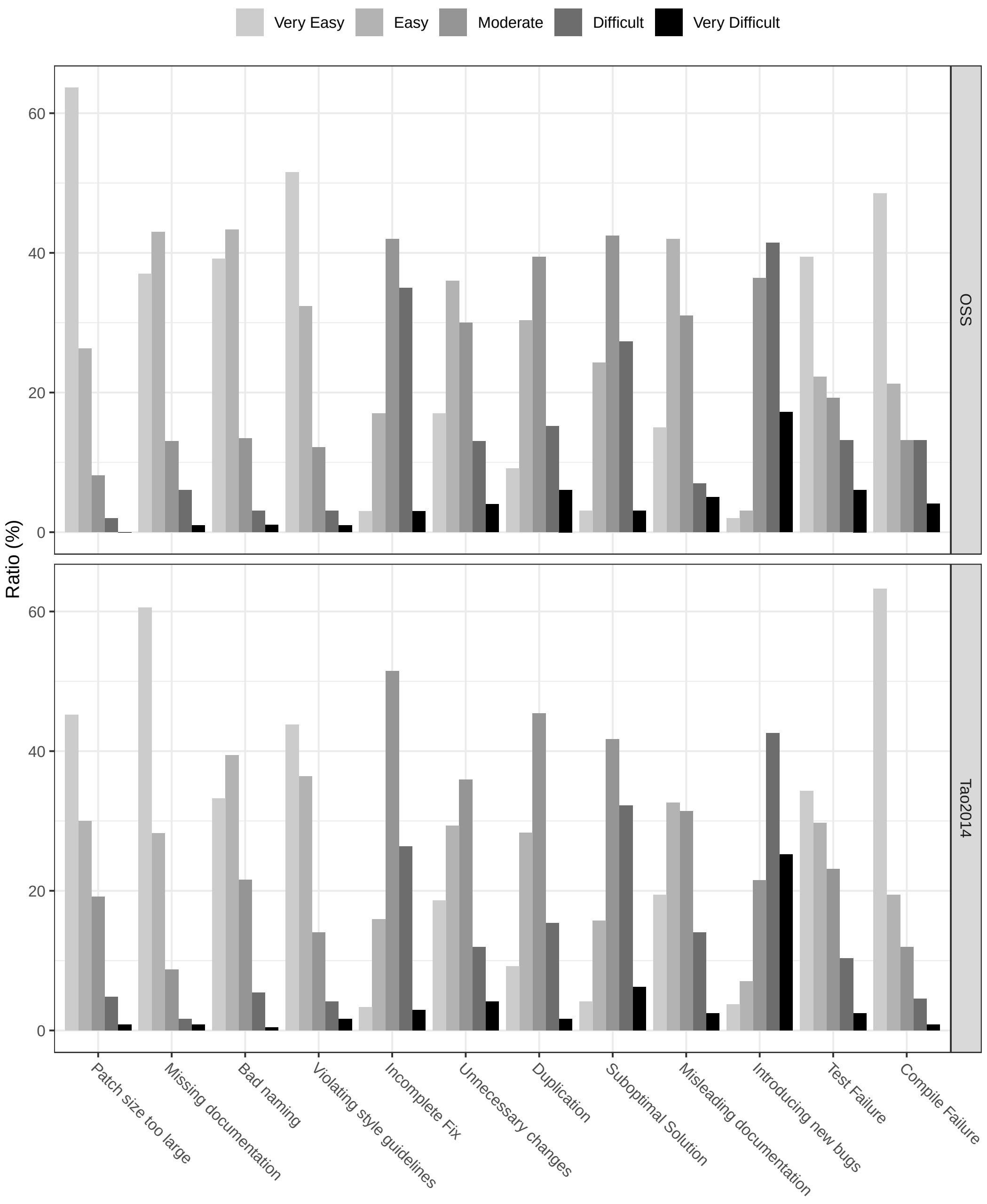


Figure 3.13: Q20. How easy or difficult it is to identify each problem during code review

a warning signal for the open source community, since developers feel the same difficulty in accessing the required information. If a sufficient tool or process improvement had been provided, developers would have experienced less difficulties to check the criteria. However, the results show no big difference. This findings call for more work to support developers.

Q20. Although detecting new bugs introduced in a change and incomplete fix are the most important, it is still difficult to identify the problems.

Since the above questions used the specified criteria from the previous study, we also asked developers 'Q21. Please specify other criteria to evaluate the patches, if you have any'.

We received 10 answers to this question. Interestingly, six developers answered that compile and test failures are detected by CI, so they did not understand why these categories were included in the questionnaire. When the Tao2014 survey was conducted, however, there was no CI support for the studied subjects, and developers manually inspected for these failures. Other notable comments from the Eclipse and OpenStack developers were as follows:

> *Usually a patch should change one thing that is easy to understand for reviewers.*
>
> *In case of a new API, the signatures of API should be optimum. Need to think about the necessity of each argument.*

Q21. Other than the categories presented in Tao et al. [2014], developers have their own criteria to evaluate a patch such as test coverage, commit message, and security.

Developers use diverse criteria when reviewing a patch. By using the same question and answer set used by Tao et al. [2014], we could compare open source in the past and present.

3.5 Discussion

In this section, we discuss notable survey results in detail.

3.5.1 Expected and actual review time

Since code review is a process in which a human reviewer inspects changes, it takes time and it is difficult to predict how long it will take. The majority of developers in Eclipse and OpenStack expected their patches to be reviewed within a day or less.

As to resolving a code review request (i.e. until it is finally merged or abandoned), the Eclipse and OpenStack developers expected this to be done within 5 days. In the actual code review data, however, most code reviews were resolved within a day.

In terms of the number of revisions needed to resolve a request, the Eclipse and OpenStack developers expected fewer than five revisions. The actual data shows a lower average number of revisions than the developers' expectations, with most code review requests are resolved within two revisions for Eclipse and OpenStack projects.

Overall, the majority of code review requests were resolved faster than the developers expected. This could be interpreted as a positive signal, since the actual resolution time was shorter than developers' expectations. However, developers must manage code reviews carefully, since the process introduces many interruptions. For example, developers get notification from a code review tool for diverse review events (e.g. a new review comment) via email or messenger. If developers conducts code review whenever they get a notification, they need to switch context multiple times. Previous studies show that interruptions and context switching leads to lower software development productivity [Kersten and Murphy, 2005, Meyer et al., 2014]. Therefore, it is important for developers to manage code review in such a way as to reduce the number of interruptions.

3.5.2 Consulting Earlier Code Reviews

During our preliminary interviews, we found that developers often consult previous code review requests. We confirmed that the majority of developers have consulted earlier code review requests during code review. More than half of Eclipse and OpenStack developers answered that they had consulted previous code reviews to get context. Developers also consulted earlier code reviews to evaluate a patch, to extract criteria, and to learn. The Eclipse and OpenStack developers indicated that earlier code reviews that changed the same file or that contain similar changes are particularly helpful when reviewing a new request.

As the results show, leveraging previous code review requests can mitigate developers' review burden. This finding directly motivates the study presented in Chapter 5. Since past code review requests contain a great deal of information based on developers' discussion in the comments, we should leverage this valuable information to enhance software productivity. Further studies should be conducted to find and leverage related code review requests.

3.5.3 Motivation for code review

We asked what Eclipse and OpenStack developers' motivations are for conducting code review. The results from open source (i.e. Eclipse and OpenStack) and Microsoft (i.e. as reported in Bacchelli and Bird [2013]) show that developers expected to find defects or to improve code during code review. Our results supported the results presented in Bacchelli and Bird [2013]. As reported in Q20, however, finding defects (i.e. catching problems that may be introducing new bugs) is one of the most difficult task for developers. Bacchelli and Bird [2013] also reported that the intended code review outcome is to improve code and understand rather than detect defects. This result calls for further research into a tool or methodology for detecting defects during code review. Despite the differences in the groups, they share similar motivations.

An interesting finding is that Eclipse and OpenStack developers cast more Top votes for avoid build breaks than Microsoft developers did. As Tao et al. [2014] reported, detecting a compilation failure is one of the most important parts of code review. However, it is interesting that Microsoft developers reported relatively lower motivation for detecting a compilation failure. This result suggests that Eclipse and OpenStack developers put more manual effort into detecting a build break than Microsoft developers. It is recommended that Eclipse and OpenStack developers review their tooling to detect build breaks.

3.5.4 Importance and accessibility of review criteria

For each criterion, we asked how important it was and how easy or difficult it was to access. We can compare the differences between past and present, as the same questions were answered by Tao et al. [2014]. It is interesting that our survey results showed that detecting new bugs is the most important criterion to check for during code review, while Tao et al. [2014] identified detecting compilation failure as the most important criterion. In line with Tao et al. [2014], checking whether a patch is introducing new bugs remains the most important and the most difficult to judge criterion in code review. This calls for further research and support to detect a new bug introduced in a patch.

The results of Tao et al. [2014] have more very important answers for compile failure and test failure. Tao et al. [2014] reported 1.9% and 5.5% of patches they manually investigated were rejected because of compilation failure and test failure, respectively. The decreased level of importance in our results for compile failure and test failure potentially shows that developers' burden for manually detecting these issues has been mitigated.

Another notable difference between our survey and Tao et al. [2014] is in responses to the incomplete fix category. Our survey of developers returned more very important answers for the incomplete fix category than seen in the results of Tao et al. [2014]. This result shows that current developers have a higher expectation than past developers did that a patch should be complete one when it is submitted for a review.

3.6 Threats to Validity

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

3.6.1 Internal Validity

Given the length of our questionnaire, to increase participation we did not push the developers to answer all questions on the survey (i.e. questions were not mandatory). However, we confirmed that most of the survey participants answered most questions except for the open-ended questions. All questions had at least a 91.0% (91 responses) response rate.

In designing the questionnaire, we interviewed only company developers. As we did not interview Eclipse and OpenStack developers, their concerns may not be fully reflected in our survey.

3.6.2 External Validity

We studied the code review process of Eclipse and Open Stack developers. Although 100 developers in total answered our survey, the results may not generalise to other companies or open source projects.

Because of the differences between code review tools, results may not be fairly comparable. Bacchelli and Bird [2013] and Tao et al. [2014] conducted surveys on different code review platforms. Bacchelli and Bird [2013] used CodeFlow, Microsoft's internal code review tool, and Tao et al. [2014] used Bugzilla data from Eclipse and Mozilla. On the other hand, our data is from the Gerrit tool used with Eclipse and Open Stack.

3.7 Conclusion

In this chapter, we investigated developers' concerns over code review. We surveyed developers in open source projects (i.e. Eclipse and OpenStack).

Our questionnaire consisted of four categories of questions extracted from a preliminary interview and two categories of questions reproduced from previous surveys (i.e. Bacchelli and Bird [2013] and Tao et al. [2014]). Through our survey, we learned how developers conduct code review in Eclipse and OpenStack projects. We summarise our findings below

When searching for an optimal reviewer to send a code review request to, Eclipse and OpenStack developers looked for a reviewer who could provide constructive comments. In terms of code review time, the majority of developers expected to get a review done within a day. We analysed actual code review data and found that the most code reviews were in fact done within a day, as the developers expected. During our preliminary interviews, we found that developers consulted the previous code review requests when reviewing a new code review request. The majority of the developers answered that they consulted previous code review requests to get context. They also said that those previous code review requests that changed the same classes or files were particularly helpful.

In our reproduction of the survey questions of Bacchelli and Bird [2013] (i.e. motivation for code review), we found that there was no major difference between developers working on Microsoft and open source projects. We confirmed that developers' most important motivations for code review were finding defects and improving code quality. Our assessment of the importance and accessibility of code review criteria questions (i.e. our reproduction of the Tao et al. [2014] survey) identified some differences between past and present open source projects.

4 Pretty Patches: An Empirical Study of Coding Conventions During Code Review [1]

4.1 Introduction

The adoption of coding conventions, or programming style guidelines, is one of the most widely accepted best practices in software development. It is assumed that adherence to coding conventions increases not only readability but also maintainability of software. However, adoption of style checking tools does not necessarily lead to projects with no or only a few violations [Smit et al., 2011].

Since coding conventions are usually not enforced by the programming languages, individuals, teams, or organisations can freely use their own coding conventions. For Java, for example, coding conventions [Sun Microsystems, 1999] were defined and introduced early on by Sun Microsystems. Organisations' own coding conventions, however, often deviate from the original. For example, Eclipse and JetBrains deviated from the coding convention of Sun Microsystems in the default configurations of their respective Java integrated development environments (IDEs) [Eclipse, IntelliJ]. In addition, developers involved in a particular software project often adjust the standard coding conventions to their own preferences [Google, OpenJDK, EGit].

[1]This chapter contains work that was submitted to the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE) in 2019. I am the main author of the paper and co-authored it with Jens Krinke, Matheus Paixao, and Chaiyong Ragkhitwetsagul at University College London, and Giovanni Rosa at the University of Molise.

Code review, a popular method for maintaining high-quality software, is the process of conducting reviews of source code changes by developers other than the change author. Diverse empirical studies report benefits of code review, such as detecting defects [Mäntylä and Lassenius, 2009], improving code quality [McIntosh et al., 2014], and sharing knowledge among team members [Bacchelli and Bird, 2013]. Code review has widely spread in both open source projects and industrial projects [Shimagaki et al., 2016, Sadowski et al., 2015, Feitelson et al., 2013], and is usually supported by platforms such as Gerrit [Mukadam et al., 2013], a widely adopted code review tool.

Although adhering to coding conventions does not affect a program's behaviour, developers spend manual effort on detecting coding convention violations during code review. Tao et al. [2014] reported that developers believe coding conventions are important criteria to apply in evaluating a patch (i.e. a code change). If a patch does not follow coding conventions, the patch will be rejected during the review. Furthermore, Tao et al. [2014] observed that 21.7% of patches in Eclipse and Mozilla projects are rejected if they, for example, violate coding conventions, include poor naming, or are missing documentation). Their results show that developers are still struggling with checking coding conventions manually, although automated tools for checking coding conventions are available such as Checkstyle and PMD. Previous studies have reported that developers ignore about 90% of warnings generated by automated tools [Kim and Ernst, 2007, Panichella et al., 2015].

Panichella et al. [2015] investigated how convention violations raised by automated tools disappear during code review. They mainly focused on how a patch under review affects the density of convention violations at the beginning and end of a code review. However, they did not investigate whether the appearance or disappearance of a convention violation was actually due to the code review or was coincidental.

In this chapter, we investigate what kinds of convention violations are addressed during code review and how much time is devoted to checking and fixing. We used the code review open platform (CROP) data [Paixao et al., 2018], a data set created from project repositories and their corresponding Gerrit code review repositories [Mukadam et al., 2013]. The CROP data contains code review data, the changes that were reviewed, and the entire source code for each change, which is usually not recorded in a Git repository. In total, we analysed 16,442 code review requests from four sub-projects of Eclipse with a further detailed investigation of 1,268 review requests and 2,172 reported style violations.

The contribution of this chapter is an in-depth investigation of coding convention violations during code review. We investigate what kind of coding convention violations are introduced, removed, and addressed during code review. Furthermore, we investigate whether manual coding convention checks done by reviewers delay the code review process.

We also discuss possible explanations for our observations, based on our examination of coding conventions in four open source projects and how automatic tools are used. The discussion highlights how developers adopt coding conventions and what difficulties they face.

4.2 Experimental Design

4.2.1 Research Questions

The goal of this chapter is to investigate in detail how developers deal with coding convention violations in real-world software projects during code review. To this end, we want to answer the following research questions:

***RQ1**. How many coding convention violations are introduced during code review?* As a preliminary research question, we investigate how many convention violations are introduced in the first (i.e. initial) patch of a code review request and how many convention violations are introduced in the last (i.e. final) patch. If code review is effective, the last patch should have fewer introduced convention violations.

***RQ2**. What kinds of convention violations are addressed during code review?* While RQ1 investigates how many convention violations are introduced in the first and in the last patch, RQ2 investigates convention violations that are introduced in the first patch but are no longer presented in the last patch — that is, introduced violations that have disappeared. Moreover, we want to know how many convention violations disappear because they are addressed and fixed in response to a code review comment that points out a convention violation. For example, a code snippet which contains a convention violation may be deleted to fix a defect, not because of a style issue. This research question identifies those convention violations that seem sufficiently important to developers so that the violations are raised during code review and are fixed in subsequent patches under review.

By answering this research question, we can extract the set of coding conventions that are perceived as being important to developers during code review and can study such convention violations in more detail. Previous studies have reported that developers ignore about 90% of warnings generated from automated tools [Kim and Ernst, 2007, Panichella et al., 2015], and the answer to the above research question can help to identify important violations so that unimportant violations can be removed from report, preventing 'Static Analysis Fatigue' [Regehr, 2010].

***RQ3**. Do convention violations delay the code review process?* As we presented in Section 4.4 and as also reported by others [Kim and Ernst, 2007, Panichella et al., 2015], Checkstyle and other automated coding convention checking tools are not widely adopted in practice. However, if such tools were used only to check for violations that are important to developers, giving the tools a low false positive rate, their use could eliminate the need for human developers to check and fix violations, speeding up the code review process. We investigate the delay caused by manual checking for convention violations that might be mitigated by adopting automated tools.

4.2.2 The CROP Data Set

To answer the research questions, we need not only to analyse source code for coding convention violations, but also to analyse changes submitted for code review and for every patch during the change and to investigate the review comments related to those patches. Instead of mining code review data ourselves, we used CROP [Paixao et al., 2018], a code review data set that links code review data to software changes. All projects in CROP are open source and use Gerrit for their code review. The data set contains code review data (e.g. description, changed files, and comments) and the complete code base before and after a change revision. Please note that Git repositories of the projects contain only accepted revisions (i.e. the changes in the final patches in a code review request), whereas CROP stores all revisions, including the revisions rejected during code review.

We selected all projects that had Java as their primary language. The statistics of the four projects are displayed in Table 4.1. The table shows the date of the first review ('Start Date') in the data set (the last review date is November 2017 for all four projects) and the number of code review requests ('Reviews') for each project in the original data set. The table also shows the number of code review requests that were finally merged ('Merged Reviews') and how many of those code review requests consisted of more than a single revision ('Multiple Patches'). Only merged reviews with multiple patches are of interest to us because the initially submitted patch was revised based on reviewers' comments, and the patch was finally accepted and merged into the codebase.

Table 4.1: Code review data sets statistics

Item	Platform UI	EGit	JGit	Linux Tools
Start Date	Feb'13	Sep'09	Oct'09	Jun'12
Reviews	4,756	5,336	5,382	5,105
Merged Reviews	3,802	4,502	4,463	3,695
Multiple Patches	2,985	2,899	2,533	3,438

4.2.3 Extracting Introduced Violations

Diverse automated tools have been introduced to support developers in adhering to coding conventions. To identify coding convention violations, we used a well-known tool, Checkstyle [Checkstyle], in version 8.8. Checkstyle is flexible and extensible, making it adjustable to any deviation from a Java coding convention. Checkstyle validates all Java source code against a set of rules by checking the actual style against the encoded conventions, and it reports all cases in which the code is not compliant with the coding conventions (i.e. coding convention violations). Checkstyle can check for 153

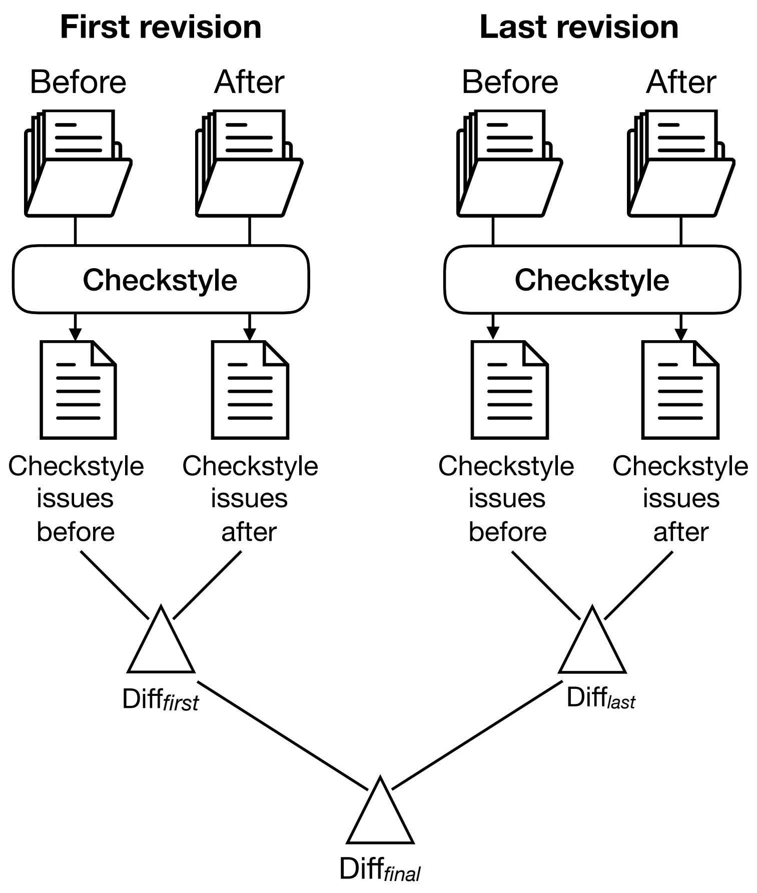


Figure 4.1: Extracting introduced violations during code review by comparing violations before and after a patch

different types of convention violations, grouped into 14 categories. However, only 62 conventions grouped in 11 categories are enabled for Sun's Java coding conventions. Based on the coding conventions of the four projects in our study, we set up our own Checkstyle configuration that complies with the conventions used by the projects.

First of all, we needed to understand what kinds of style violations are introduced at the beginning of a code review request for a new change. Since static analysis tools such as Checkstyle require the entire source code, we extracted style violations from two different versions of source code: before and after applying the first patch of a code review request. Figure 4.1

shows the process we used to extract violations that appeared or disappeared during code review. By comparing the lists of violations, we could extract those violations that were newly introduced by the first patch of a code review request (Diff$_{first}$). In matching the lists of violations, we ignored line numbers, file names, and so on, because those elements may be changed by the patch. Moreover, only the added — that is, newly introduced — violations were considered, because code review should prevent the introduction of convention violations. It is important to note that the system may have changed while the patch was being revised. Therefore, the state before the last revision may be different from the state before the first revision (in technical terms, the commits of the subsequent revisions were rebased). Changes in violations between the state before the first revision and the state before the last revision need to be ignored. Therefore, we repeated the same steps above on the last patch of a code review request to extract the list of introduced violations at the end of the code review (Diff$_{last}$).

The next step (for RQ2) was to determine whether the newly introduced violations disappeared during the code review. A violation was considered to have disappeared if it was introduced in the first patch but it was no longer present in the last patch. We determined what kinds of violations had been introduced in the first patch (Diff$_{first}$), and then disappeared in the last patch (Diff$_{last}$) by comparing the difference between Diff$_{first}$ and Diff$_{last}$. The comparison result (Diff$_{final}$) contains the violations that were added (appeared) and deleted (disappeared) during the code review. For RQ2, only the disappearing violations are needed (the violations deleted in Diff$_{final}$).

Table 4.2 shows the overall statistics for our violation extraction from code review data. The 'Unchanged' row presents the number of code review requests that have no violation changes between the first and last patch (i.e. Diff$_{first}$ = Diff$_{last}$). The 'Changed' row shows the number of code review requests that contain violation changes (violations either appeared or disappeared) between the first and last patch (i.e., Diff$_{first} \neq$ Diff$_{last}$). The 'Improved' row states the number of code review requests that were violations that disappeared at the end of the code review. From the 4,502 merged EGit reviews, for example, we extracted Diff$_{first}$ and Diff$_{last}$ for 2,899 code review requests that have multiple patches. Among the 2,899 reviews, 2,138 have no changes in violations, while only 761 reviews have differences in violations between the first and last patch. A total of 375 reviews out of these

Table 4.2: Overall convention violation variations found between first and last patches of a code review request

Item	Platform UI	EGit	JGit	Linux Tools
Unchanged	2,636	2,138	1,799	2,886
Changed	349	761	734	552
Improved	199	375	397	314

Table 4.3: The number of manually investigated convention violations and the number of conflicts between human investigators. The conflicts were all resolved based on discussion.

	Platform UI	EGit	JGit	Linux Tools
Violations	313	622	640	597
Conflicts	16 (5.11%)	55 (8.84%)	80 (12.50%)	73 (12.23%)

761 contain violation changes in which a violation was introduced in the first patch but is no longer present in the last patch (i.e. the code review has improved the change).

Although we have the list of convention violations that appeared and disappeared during code review, we do not know whether an appearing convention violation disappeared because it had been flagged by a reviewer during the code review and addressed by the patch author. For example, EGit has 375 improved code review requests based on the definition above. The 375 code review requests contain 622 violations that were introduced in the first patch and disappeared in the last patch. Among these 622 violations, however, we do not know how many were addressed during code review (i.e. they did not coincidentally disappeared because of other addressed issues). To determine whether reviewers detected and raised the convention violations during code review, we manually investigated the review comments from the four projects for all patches of such reviews. We focused primarily on finding whether a reviewer's comment (e.g. 'please remove the trailing whitespace') points out the convention violation that appeared in Diff_{first}, but is no longer present in Diff_{last}. If we could locate a comment pointing out the style violation, we labelled the violation as 'Confirmed'; otherwise, we labelled it as 'No Evidence'. To mitigate subject bias of the manual investigation, two of the authors (investigators) inspected the comments independently. They read the review comments and looked for mentions of the reported style violations. Sometimes they also checked the patch to see how a violation disappeared. After finishing the independent rounds of investigation, the two investigators held a discussion to resolve conflicts. They discussed a conflict until both agreed on the same label.

As shown in Table 4.3, from 313 (Platform UI) to 640 (JGit) convention violations were introduced in the first patch and resolved in the last patch of code review requests. Please note that we did not investigate code review requests that had only one patch or that had no style violations introduced and removed (i.e. we investigated only those code review requests that introduced and removed at least one coding convention violation). As shown in the table, the investigators initially had conflicting results in 5.11%–12.50% of the cases. However, the investigators resolved the conflicts by discussing until both of them agreed.

4.3 Results

The results from the above described experiment are used to investigate and answer the three research questions.

4.3.1 RQ1. How many convention violations are introduced during code review?

Figure 4.2 shows the accumulated numbers of violations appearing in the first and last patch of a code review request for the four projects. The figure shows the number of violations for each violation category. There are 11 categories for 62 convention violation types overall. For example, a violation of type RegexpSingleline belongs to the Regexp category. The grey bar shows the number of violations in the first patch, while the black bar shows the number of violations in the last patch. As shown in Figure 4.1, we ran Checkstyle before and after applying the first patch and the last patch, respectively. By computing the differences between before and after applying a patch, we derived the list of introduced violations in the patch (i.e. Diff_{first} and Diff_{last} for the violations introduced in the first and the last patch, respectively).

A Mann-Whitney U test showed no statistically meaningful difference between the the numbers of convention violations introduced in the first and the last patches (p-value > 0.05) per violation type. Although there is no statistical difference per violation type, the number of added convention violations in the last patch is larger than the number in the first patch for all convention categories except for the Regexp convention category.

One cannot conclude that code review introduces more violations than it prevents, because the number of violations naturally increases as the code base grows larger. Figure 4.3 shows the distribution of the numbers of inserted and deleted lines in the first and last revisions of a code review request. We conducted a Mann-Whitney U test to determine the statistical difference between the first and the last patches in terms of the numbers of inserted and deleted lines (i.e. the difference between the number of inserted lines in the first and the last patches and also the difference between the number of deleted lines in the first the and last patches). All four projects show a statistically meaningful difference between the first and the last patches for both inserted and deleted lines (p-value < 0.05). As shown in Figure 4.3, the last patch usually has more added lines than the first patch. This gradually increasing pattern is similar to the pattern shown in Figure 4.2. It shows that many violations appear in the first patch of a code review request, the violations do not disappear, and more violations appear until the last patch, which is also usually larger than the first patch. Therefore, the number of violations increases as the number of inserted lines increases.

Figure 4.4 shows how the number of appearing violations increases in the number of added lines during code review. For each code review request, we extracted the number of appearing violations in the first and the last patches and the y-axis shows the difference (increase) from the first to the last patches, and the x-axis shows the difference (increase) in the number of inserted lines between the first and the last patches. We consider only the number of inserted lines, since deleted lines cannot introduce convention violations. Please note the number of inserted lines can be negative, since the first patch can add more lines than the last patch. Moreover, the first patch can have more appearing violations than the last patch. Although there is

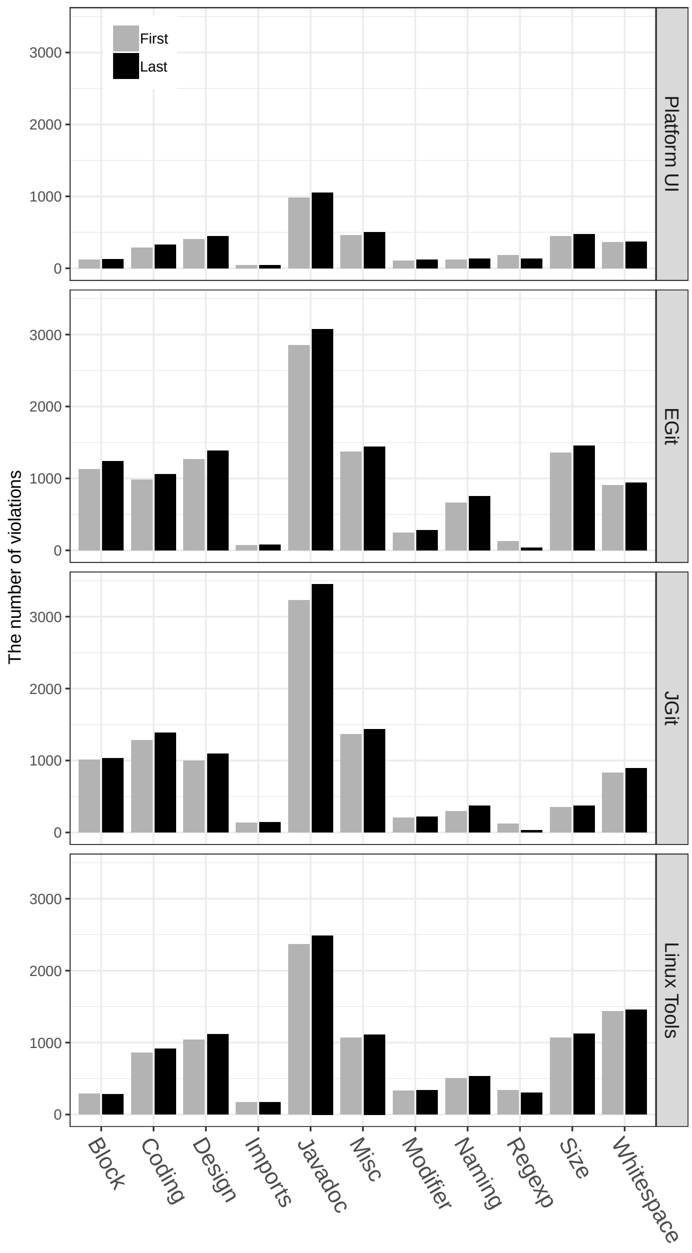


Figure 4.2: The accumulated numbers of convention violations detected in the first (grey) and last (black) patches

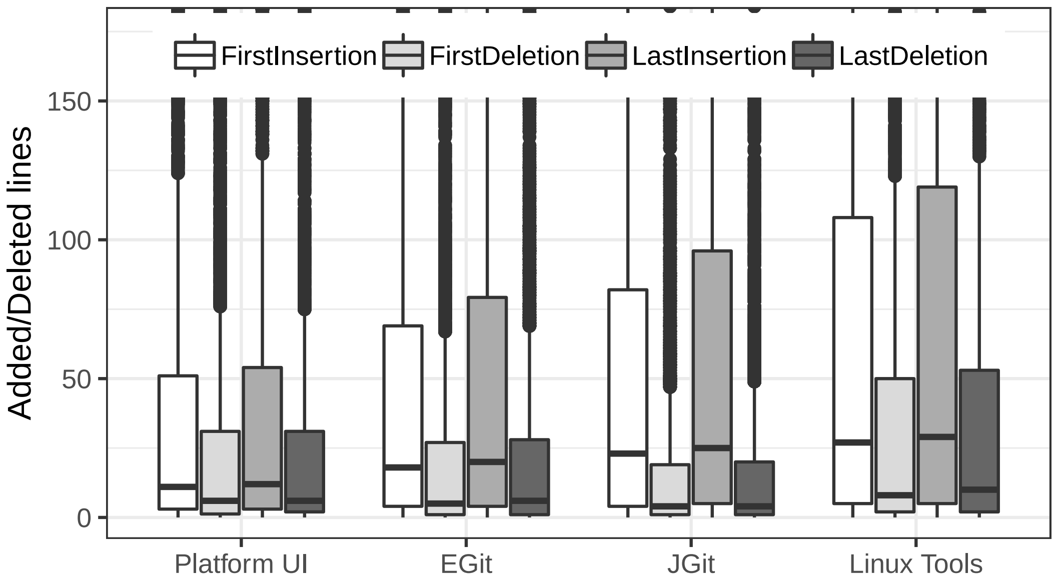


Figure 4.3: Number of lines added or deleted in the first and last patches

a variance, the number of violations generally increases as the number of inserted lines increases.

Considering the different violation categories in Figure 4.2, the Javadoc conventions category shows the largest number of violations in all four projects. This observation is in line with a previous study that showed that developers tend to treat missing documentation as a relatively less important issue than other conventions [Tao et al., 2014].

The Regexp conventions category shows the lowest number of violations in both first and last patches. In addition, the number of added Regexp violations in the last patch is smaller than the number of violations in the first patch (i.e. $|\text{Diff}_{first}| > |\text{Diff}_{last}|$). This shows that the violations in this category may have been addressed during code review (i.e. spotted by reviewers, raised, and fixed). The Regexp convention violation category contains the RegexpSingleline convention violation, a configurable check that is usually set to check for trailing whitespace (although most whitespace-related conventions belong to the Whitespace category). No other violation of the Regexp category has been reported, thus Regexp represents trailing whitespace violations. We discuss the trailing whitespace issue in detail in Section 4.4.3.

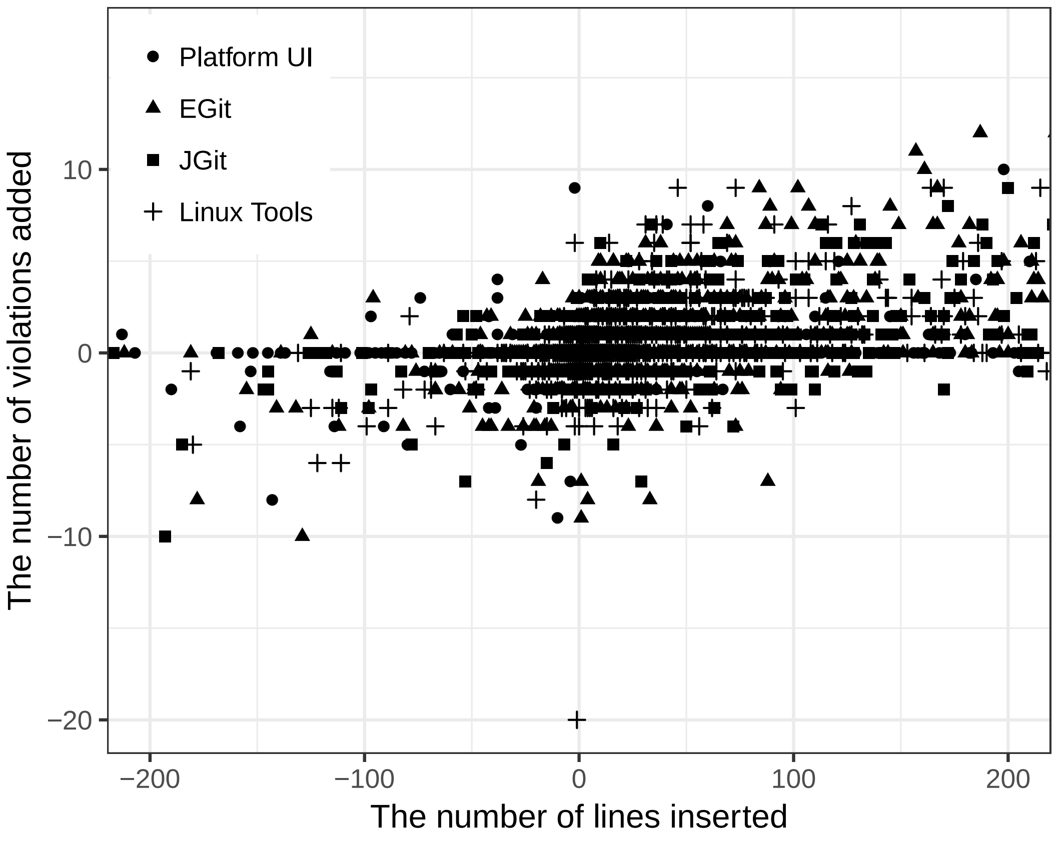


Figure 4.4: The number of violations varies over the number of inserted lines between the first and the last patches

Please note that the conventions in the Whitespace category provide finer-granularity checks for whitespace (e.g. GenericWhitespace checks the whitespace around the Generic tokens '<' and '>'). On the other hand, the convention violations might simply have disappeared when issues other than convention violations were addressed (e.g. bug fixes or to removal of redundant code). We therefore investigate in RQ2 whether a convention violation disappeared because it had been addressed during the code review.

Key observation: The number of convention violations usually increases during code review as the size of the patches increases.

While this key observation is not a surprising result, it confirms a correlation of the number of introduced violations and the size of a patch. This suggests that code review is not effective in preventing the introduction of convention violations.

4.3.2 RQ2. What kinds of convention violations are addressed during code review?

Figure 4.5 shows the number of code review requests in which the violations of a specific category disappeared during the code review between the first and last patches. The reported numbers are the result of the manual investigation of all 2,172 disappearing style violations, which affect 1,268 review requests. For 55 out of the 62 violation types, we found at least one disappearing violation.

The x-axis presents different categories of coding conventions, while the y-axis shows the number of violations for each category. The 'Confirmed' bars (grey) show the numbers of reviews in which a convention violation of a specific category appeared in the first patch, the convention violation was raised by a reviewer in a code review comment, and the final patch no longer contains the violation (that appeared in the first patch). The 'No Evidence' bars (black) show the numbers of reviews in which a convention violation of the specific category appeared in the first patch, and the final patch no longer contains the violation, but in which the reviewers did not raise concerns about the violation (i.e. the investigators could not locate any code review comments regarding the violation). Mann-Whitney U tests for all

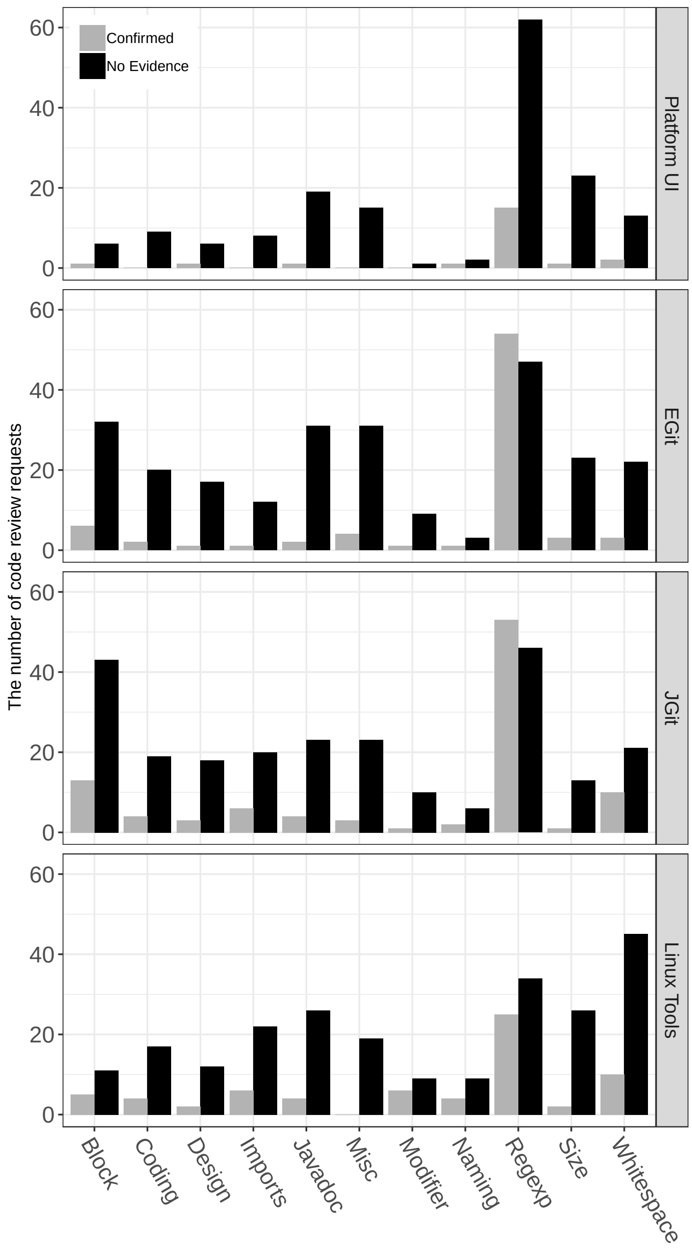


Figure 4.5: The numbers of code review requests in which violations were addressed during code review as confirmed by manual investigation ('Confirmed' — grey) and those in which violations disappeared without evidence ('No Evidence' — black) found in the manual investigation

four projects show statistically meaningful differences between the numbers of investigated violations and confirmed violations (i.e. $p\text{-}value < 0.05$) in each project. The statistical tests indicate that the numbers of violations confirmed in our manual investigation were not affected by the total number of investigated violations.

In 55 convention violation types, at least one violation disappeared in the last patch. In only 38 of these violation types could manual investigation confirm at least one violation had been raised during code review. It may seem that developers did not care about coding convention violations for the other 17 violation types.

It is interesting to see that the majority of cases in which violations of a specific category were introduced in the initial patch but disappeared in the last patch were not due to reviewers raising the issue ('No Evidence' in the figure). We found two scenarios illustrating why a violation disappeared even though no review comment had mentioned the violation. In the first scenario, a patch author changed a patch to address a reviewer's comment, and although the comment was not about the convention violation. The changed patch no longer contained the violation. For example, a reviewer points out a bug in a patch which also introduces a convention violation. While fixing the bug, the patch author removes the violation as well (without knowing about the violation). Please note that if the reviewer had pointed out the convention violation in the comment, we would have labelled this occurrence as 'Confirmed'. In the second scenario, the change was not in response to a reviewer's comment, but the new patch no longer includes the convention violation. In this scenario, it is unclear why the patch has been changed. Most often this was due to self-review (i.e. the patch was created and reviewed by the same developer without a review comment).

We can also see that the rankings of confirmed violations are similar for all four projects. In considering both 'Confirmed' and 'No Evidence' categories, it becomes clear that the problems of trailing whitespaces (i.e. Regexp) were pointed out by a human reviewer and fixed afterwards in large numbers.

For JGit and EGit, the second largest number of raised and fixed convention violations was the requirement that blocks always be enclosed in braces (i.e. the NeedBraces violation in the Block category). As will be discussed in Section 4.4.4, as of 27 January 2015, EGit and JGit no longer allowed braces around single-line blocks, and not enclosing a block in braces became a violation. Before this update of the coding convention, among the 31 and 29 disappearing NeedBraces violations in EGit and JGit, respectively, none were confirmed. After the new convention was adopted, there were 6 and 13 confirmed NeedBraces violations out of 7 and 27 disappearing NeedBraces violations, respectively.

The next largest number of raised and fixed convention violations (second largest for Linux Tools and Platform UI, third largest for JGit and EGit) is in the Whitespace category.

Key observation: Whitespace is the convention violation that is most often raised and fixed during code review — in particular, trailing whitespace.

For the violation types that reviewers care about, one would expect that not only would violations introduced in the first revision be removed in the final revision, but also that no new violations would be introduced. For most (33) of the violation types, however, this was not the case. There were only five violation types for which, in all four projects, there are fewer reviews where the violation is introduced in the final revision compared to the number of reviews introduced in the initial revision. As the violation type RegexpSingleline (trailing whitespace) was the most often raised and fixed issue for all four projects, it appears that it is the violation that reviewers cared about most. However, even for RegexpSingleline, a significant number of violations still appear in the last revision of a patch (as can be seen in Figure 4.2).

It seems that although reviewers care somewhat about the 33 violation types, they do not care enough to ensure that new violations of the type are not introduced during the code review, and they do not apply consistent and rigorous checking.

Key observation: Many coding conventions violations are ignored by developers and reviewers.

One possible reason that coding convention violations are ignored may be that raising a coding convention violation would require the patch author to fix the violation and resubmit the patch for review. This would delay the

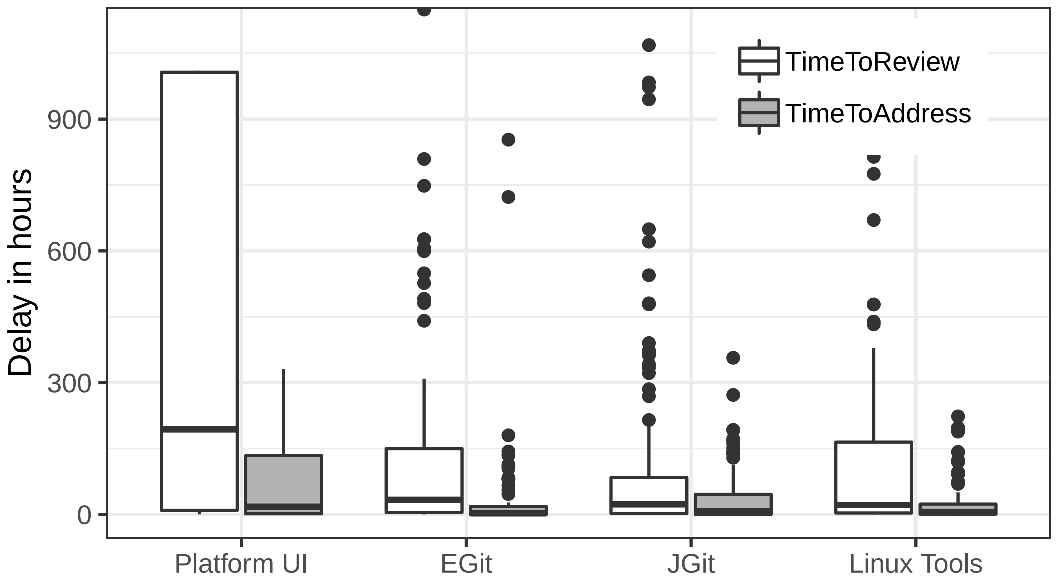


Figure 4.6: Delay in detecting ('TimeToReview') a convention violation during code review and delay in addressing ('TimeToAddress') a convention violation

integration of the change and consume additional developer and reviewer time. This is a case we investigate further below.

4.3.3 RQ3. Do convention violations delay the code review process?

Although automated convention checking tools support developers by detecting convention violations instantly, they are often not adopted in practice, and many convention violations are still detected through manual inspection by a reviewer. In this section, we investigate the delay caused by manually detecting (and fixing) a convention violation during code review.

During the manual investigation for RQ2, we recorded the timestamps of review comments that point out convention violations and the timestamps of the subsequent patches. Figure 4.6 shows the time (in hours; not all outliers are shown) from the introduction of a convention violation in the first patch (which is the time of the initial patch submission) to a review comment

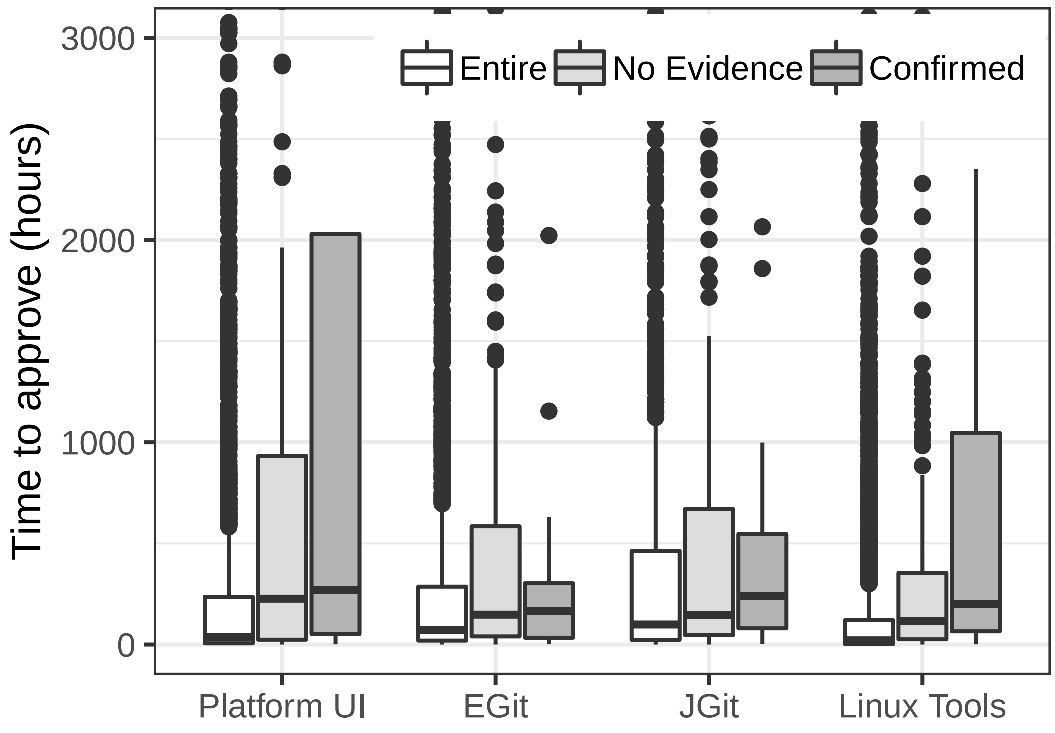


Figure 4.7: Time taken for code review requests to be approved

that points out the violation ('TimeToReview') and the time from the review comment to the next patch that removes the violation ('TimeToAddress'). It takes more than 24 hours (the median is 24.21 hours) to receive a review comment that points out a convention violation. It then takes more than 6 hours (the median is 6.90 hours) until the violation is addressed and fixed in the next patch. If developers used an automated convention checking tool such as Checkstyle, they could immediately detect violations (with no delay of 24 hours) and address them even before submitting a patch for review.

To investigate whether detecting and addressing convention violations actually causes longer code reviews, we measured the time between the initial patch submission and the last comment on a review request, which is the automated comment that the final patch has been merged. Figure 4.7 shows the measured time for three different datasets: The first dataset ('Entire') is for all code review requests that have at least two patches, meaning a reviewer has raised an issue that needed to be addressed. The second dataset ('No Evidence') consists of all code review requests in which at least one violation disappeared but there is no evidence that a reviewer identified and raised a convention violation. The third dataset ('Confirmed') consists of all review requests in which the manual investigation has confirmed that a reviewer identified and raised a convention violation. The figure clearly shows that the median time for review requests in which a convention violation is commented on by a reviewer is higher than for the other two datasets.

Key observation: Overall, manual checking and fixing of a convention violation may delay the code review process.

4.3.4 Discussion

The results for the three research question suggest some key observations about coding conventions during code review.

Convention violations accumulate as code size increases

As we presented in Section 4.3.1, convention violations not only accumulate as code size increases, but the number of violations that a change introduces also increases during code review. The only exception over the four analysed projects was the Regexp category (i.e. trailing whitespace). The accumulated convention violations decrease code readability and maintainability; they become technical debt. Therefore, our results call for further studies on how to prevent accumulating convention violations as code size increases.

Developers manually check convention violations during code review

Although diverse convention checking tools are available, developers still manually check for convention violations during code review. Since code review is the gatekeeper process to assure code quality, the check for convention violations seems natural. Our results in Section 4.3.2 show that many convention violations are detected manually by developers, including very simple trailing whitespace violations. However, the results also show that humans are neither effective nor consistent in preventing the introduction of convention violations. For almost all violations that we could confirm were spotted and raised by a reviewer, we observed that the number of violations introduced by a change was higher at the end of the code review than in the initial version of the change. The notable exception was again trailing whitespace.

Convention violation checking delays code review

The answer to RQ3 highlights that the manual convention violation checking can cause delays during code review. Surprisingly, a patch author faces a median wait of more than 24 hours to receive a reviewer's comment raising a convention violation and it takes a median 6 additional hours to fix the violation. More importantly, even if the change is otherwise ready to be accepted, the presence of a convention violation raised by a reviewer will not only require time to fix the violation but will also require another round of code review, including a CI run. Overall, the median time needed from the initial submission of a patch to the approval of the patch is greater for code review requests in which a reviewer raises a coding convention violation. We believe this delay can be reduced if developers can employ a trustworthy automated convention violation checking tool.

4.4 Coding Conventions in Practice

The experiment presented in the previous sections showed that convention violations can cause delays during code review because reviewers check them manually and authors need to fix violations that they were made aware of. Moreover, reviewers are not effective at preventing the introduction of convention violations. One can make the case that automatic convention checking (and fixing) tools should be used to allow authors and reviewers to focus on issues that cannot be automatically checked. To understand why such tools are not used, we investigated four open source systems to see how automatic tools are used (or not used) and how they handle convention violations. Moreover, we looked for reasons why trailing whitespace violations were spotted and fixed more often than any other violation, and why, for JGit and EGit, the second largest number of raised and fixed violations was the requirement to always enclose blocks in braces.

4.4.1 Checking Tool Adoption

Our experiment has used Checkstyle, which is often used for convention violation checking. However, none of the four projects we investigated showed evidence that Checkstyle had been used. In May 2011, there was a discussion among Eclipse developers about adopting Checkstyle in their projects[2], but the tool does not seem to have been widely adopted. Instead, one advice is to use the Eclipse Formatter. One reason why Checkstyle is not introduced into existing codebases is that the source code does not adhere to the coding conventions. As one Eclipse developer expressed it, '*I would never impose Checkstyle on an existing code base. Even in well behaved code one would get thousands (more likely tens of thousand) warnings/errors.*'.

According to JGit's and EGit's coding conventions, both projects integrate FindBugs [Hovemeyer and Pugh, 2007, Ayewah et al., 2007] and PMD's copy-paste-detector [PMD] as part of their build process. However, JGit developers currently do not use these tools in their build process.

The Eclipse Platform UI project has adopted SonarQube [SonarQube] to conduct code quality analysis. However, it does not seem to be used regularly, because as of 9 January 2019, 32,796 issues had been reported,[3] 757 of them categorised as critical. SonarQube was mentioned only once[4] during code review discussions on Gerrit for Eclipse Platform UI, suggesting that the SonarQube quality analysis is not important enough to Eclipse Platform UI developers.

[2]`https://bugs.eclipse.org/bugs/show_bug.cgi?id=347666`

[3]`https://sonar.eclipse.org/dashboard/index/eclipse.platform.ui:eclipse.platform.ui`

[4]`https://git.eclipse.org/r/#/c/65695`

Although Checkstyle is not used in the Eclipse projects that we investigated, another project contained in the CROP dataset uses it. Spymemcached is a lightweight Java implementation of a memory caching system. The project was terminated and became the groundwork for the Java client for the Couchbase NoSQL database [Couchbase]. Although the project is no longer maintained, we found an interesting case regarding coding conventions. At the beginning of the project, spymemcached developers did not employ any automated coding convention checking tools in their development process. Therefore, developers spent considerable time during code review discussing and fixing style issues. In August 2011, the project integrated Checkstyle into its development process. As an integration step, a developer executed Checkstyle for the complete project code and fixed all the style violations that were reported in a single commit.[5] Because of this 'big-bang' style change, most of the files in the code base were changed by a single developer in a single commit, and the change history was tainted. For example, developers can no longer track when the last change was introduced and who made the change in a file farther back than the 'big-bang' commit. To avoid this problem, the projects that we investigated have decided against fixing present convention violations. Moreover, we have seen during our investigation that code reviewers often reject changes that fix only convention violations. Instead, changes should not introduce new coding violations.

4.4.2 Fixing Tool Adoption

Assuming that Eclipse developers themselves use Eclipse, it was surprising to see a large number of convention violations that could have been prevented by Eclipse's automatic formatting system. For example, the Eclipse Platform UI project has documented coding conventions together with instructions on how to adhere to them. It has adopted the Eclipse Coding Conventions [Eclipse], and it provides IDE configurations for code formatting that a contributor needs to import. Clear instructions are given on code formatting:

[5] http://review.couchbase.org/#/c/8644/

Avoid formatting whole files — as this can generate pseudo-changes (whitespace related) when committing changes to existing source files. The easiest way, for Java files, is to have 'Format edited lines' activated

Given these explicit instructions on how to use automatic code formatting, one would assume that violations to the corresponding documented conventions do not occur. As presented in the previous sections, however, there was an abundance of code reviews in which violations were discussed by developers. Some discussions even mentioned that the Eclipse automatic formatting system was not working correctly at some point.[6]

4.4.3 Trailing Whitespace

The introduction of trailing whitespace was the only convention violation that decreased in number during code review. One might conclude that trailing whitespace is a convention violation that developers are specifically interested in. A developer contributing to the Eclipse Platform UI even mentioned trailing whitespace in discussing coding conventions during code review:

The general rationale behind coding style, namely improving readability, is very important to code reviews, because the main task of code reviews is to read code. In Eclipse Platform, my experience is that coding style is handled quite strictly. Even a trailing space at the end of the line can lead to a rejection of the change set and needs to be fixed in order to be included.

However, the explanation for the role of trailing whitespace may be much simpler, because Gerrit's visualisation of changes highlights trailing whitespace in red. Figure 4.8 shows an example from the Eclipse Gerrit repository that illustrates how a reviewer[7] pointed out trailing whitespace in a change. The patch author introduced two meaningless trailing whitespaces (highlighted in red) while making a change. Then the reviewer pointed out the

[6] https://bugs.eclipse.org/bugs/show_bug.cgi?id=477476
[7] The reviewer's photo and name are blinded for privacy.

issue, the patch author revised the change, and the trailing space disappeared in the following revision. Given that striking visualisation of trailing whitespace in Gerrit, it is no wonder that reviewers often raise the violation explicitly.

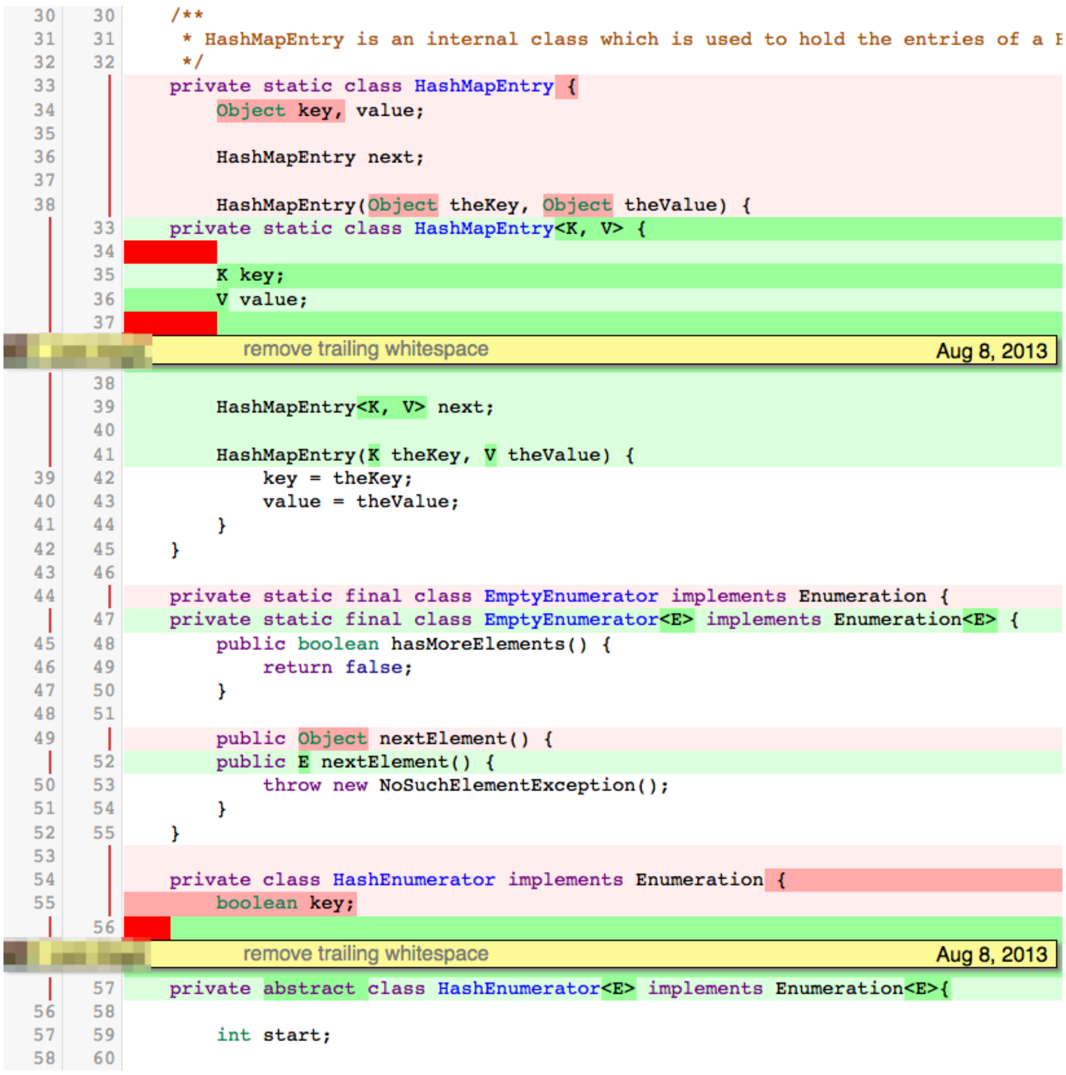
4.4.4 Enclosing Blocks in Braces in JGit and EGit

For JGit and EGit, the second largest number of raised and fixed convention violations related to the requirement to always enclose blocks in braces (i.e. the NeedBraces violation in the Block category). Both JGit and EGit use deviations from Eclipse coding conventions [EGit]. Their coding conventions raise two particular cases: First, they highlight that trailing whitespace should be automatically removed (similar to Eclipse Platform UI). Second, they discuss the need for braces around one-line statements. The second element conflicted with the Eclipse coding convention. Before 27 January 2015, EGit and JGit did not allow braces around single-line blocks, and many reviewers requested the removal of braces in cases where single-line blocks were enclosed in them. For example, in code review request #11558, the reviewer pointed out that the patch author should remove the braces around a single line. A proposal to change this deviating coding convention was raised in Bugzilla (Bug #457592) by one of the main JGit/EGit developers.

> *I found this rule pretty annoying for many reasons: the rule itself is not only an exception from the general rule to have braces around any blocks but also contains another 2 exceptions that you *have to* use braces in some special cases. The code containing multiple if/else block with and without braces looks inconsistent, refactoring often leads to left-over braces. Especially that Eclipse formatter does not remove 'unnecessary' braces automatically is kind of a showstopper - one can't even trust Ctrl+Shift+F.*

Based on this discussion, the developers updated the contributor guide [EGit]. The guide currently says:

> *Starting with 3.7.0 braces are mandatory independently of the number of lines, without exceptions. The old code will remain as is, but the new changes should use the style below:*



```
/**
 * HashMapEntry is an internal class which is used to hold the entries of a
 */
private static class HashMapEntry {
    Object key, value;

    HashMapEntry next;

    HashMapEntry(Object theKey, Object theValue) {
private static class HashMapEntry<K, V> {

    K key;
    V value;

          remove trailing whitespace                                    Aug 8, 2013

    HashMapEntry<K, V> next;

    HashMapEntry(K theKey, V theValue) {
        key = theKey;
        value = theValue;
    }
}

private static final class EmptyEnumerator implements Enumeration {
private static final class EmptyEnumerator<E> implements Enumeration<E> {
    public boolean hasMoreElements() {
        return false;
    }

    public Object nextElement() {
    public E nextElement() {
        throw new NoSuchElementException();
    }
}

private class HashEnumerator implements Enumeration {
    boolean key;

          remove trailing whitespace                                    Aug 8, 2013
private abstract class HashEnumerator<E> implements Enumeration<E>{

    int start;

```

Figure 4.8: A reviewer pointed out trailing whitespace during code review. Gerrit highlights trailing whitespace in red.

```
if (condition) {
    %doSomething();
}
```

The main reason for the change was ***to simplify the review process****, coding guidelines and to make them more consistent with Eclipse code formatter.*

The developers updated their coding conventions to reduce confusions. However, we found that updating the coding conventions failed to reduce confusion, because our investigation of code reviews showed that the developers were still confused and spent manual effort detecting and fixing the violations of this convention.

Deviations from generally accepted coding conventions may lead to confusion and unnecessary discussions during code review and should be avoided. Removal of such deviations can also cause confusion and unnecessary discussions.

4.5 Threats to Validity

We selected four projects (i.e. Platform UI, EGit, JGit, and Linux Tools) from the Eclipse Foundation. Although the projects have large amounts of code review data, as shown in Table 4.1, the results may not be generalisable to other commercial or open source projects. Moreover, all four projects use Gerrit as the code review platform, and this has only limited support for including automatic checking tools.

We focused only on the first and the last patch of a code review request. However, it is possible that we may have missed convention violations in intermediate patches — for example, a case in which a developer violated a coding convention in the second patch, a reviewer spotted it, and the violation was addressed in the third patch. However, we assume there will be no significant difference between the first patch and the following patches in a code review request, since Eclipse and other open source and proprietary projects limit the size of patches [Weißgerber et al., 2008].

Since our investigation for RQ2 was a manual process, it risked being subjective. To mitigate this threat, two authors investigated the data independently. If the two investigators found a conflict between their investigation results, they discussed the conflict until they agreed on the same decision.

The delay reported in this chapter might not be representative, since a code review comment and patch may address multiple issues at once. Therefore, the time we measured between them in this chapter may not be solely spent on the checking and fixing of convention violations.

4.6 Conclusion

In this chapter, we described how developers handle coding convention violations during code review. First, we investigated how many convention violations are introduced (appear) in the first patch of a code review request and disappear in the last patch. Then, we investigated whether violations disappeared because they were addressed in response to reviewer comments. We found that developers manually check coding conventions during code review. Our investigation results highlight that adherence to coding conventions while introducing changes is important, and that some conventions are ignored during code review. Our results also show that changes to already agreed coding conventions should be avoided. Finally, we investigated how manual convention checks by developers can delay the code review process. Code review requests in which reviewers raise coding convention violations take longer to be finally accepted and merged.

Our study calls for further future work on coding conventions. For example, the majority of coding convention violations can easily be detected by automated tools, rather than by a reviewer's manual inspection. On the other hand, it is important to provide fewer false-positives (i.e. violation warnings that are either unimportant or unnecessary) to developers, and it is necessary for the tools to be able to analyse changed code only. Based on the results presented in this chapter, automated tool support can save developers' time and boost development speed.

5 Recommending Related Code Reviews[1]

5.1 Introduction

Because code reviews focus on the submitted change, they do not provide the full context surrounding the change. One way to understand the context is to visit related code reviews—for example, previously performed reviews of similar changes. There are various scenarios in which visiting related reviews can provide important information. For example, reviewers may want to know if there is something that needs special attention as it has been overlooked by developers in the past, as pointed out in previous reviews. In addition, they can check previous code reviews to come up with review criteria and comprehend the context of changes. Another example concerns a developer submitting a change. Even before submitting changes for review, a developer can consult earlier reviews of similar changes to avoid mistakes. This could be especially helpful for a new team member, who can compare and contrast related reviews to broaden her understanding and familiarise herself with the team's standards Bacchelli and Bird [2013].

Although code review tools keep track of developer discussions and the development context, manual effort is required to locate code review requests. If a developer does not manually link related code review requests to a new code review request or does not add meaningful metadata, developers may not be able to leverage available discussions and their development context

[1]This chapter contains work that was submitted to Transactions on Software Engineering (TSE) in 2018. I am the main author of the paper and co-authored it with Manal Alonaizan at King Saud University, Jens Krinke and Chaiyong Ragkhitwetsagul at University College London, and Daniel Schwartz-Narbonne and Bill Zhu at Amazon Web Services.

that is stored in code review tools and repositories. Current code review tools cannot find and link to related code review requests, and their search capabilities work only if there is matching metadata in previous code review requests. Developers may waste valuable time repeating similar discussions, or, even worse, may overlook issues that have been pointed out in previous reviews of similar changes. As code review data accumulates, it gets more difficult to manually keep track of related code review requests without having a recommendation system in place. For example, on average, 74.4 code review requests are submitted every day to Qt's Gerrit repository Xia et al. [2015].

We present a technique that recommends code review requests with similar patches from a code review history to a newly submitted patch based on the similarity of the patches. This technique can help developers to easily locate similar code review requests in a project's code review history.

The main contributions of this chapter are as follows:

- A related code review recommendation system
- An empirical study on the related code review recommendation system

The remainder of this chapter is organised as follows: Section 5.2 describes the special characteristics of the code review processes at Amazon and in open source projects. Section 5.3 provides the motivation for our approach. Section 5.4 presents an overview of our approach, research questions, and the experimental setup. The results of the evaluation are presented in Section 5.5 and discussed in Section 5.6. Threats to validity are discussed before the chapter concludes.

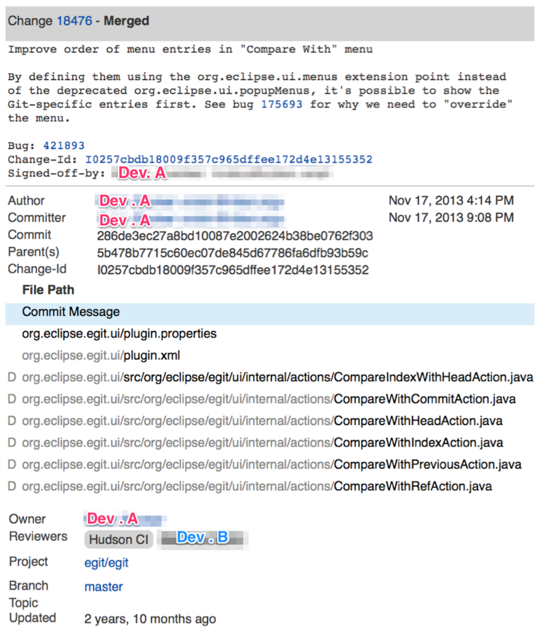
5.2 Code Review

Gerrit provides specialised code review support features that may not be offered by other tools. One of the features is the *Change-Id*. The Change-Id is a unique identifier for grouping patches that belong to the same code review request. When patches are merged into the Git repository, the Change-Id is stored in the commit message. The Change-Id enables developers to trace commits back to the code review request in Gerrit. Gerrit provides additional information to easily locate other code review requests and patches including *Related Changes*, *Topic*, *Cherry Picks*, and *Conflicts With* information. The *Related Changes* information shows the ancestors and descendants of the current code review request (i.e. requests on which the current review request depends and open review requests that depend on the current review request). The *Topic* information shows only open code review requests that have the same topic as the current review request. Gerrit's topics are based on the topic branches in Git (i.e. temporary branches that a developer pushes to commit a set of logically-grouped dependent commits). The *Cherry Picks* information shows the cherry-picked code review requests, filtering out abandoned code review requests. Cherry-picking in Git is typically designed to introduce particular commits from one branch into a different branch. A common use is to forward or backward commit from a maintenance branch to a development branch. The *Conflicts With* field shows the code review requests that conflict with the current code review request, with abandoned code review requests are filtered out.

Code review is a highly recommended process at Amazon. The internal code review tool is highly integrated with Amazon's internal infrastructure and provides a convenient review environment. The overall code review process is similar to the process described above. Similarly to the other code review tools, the internal code review tool provides overall information required for code review, including the author of a change, assigned reviewers, and syntactically highlighted patches.

5.3 Motivation

Figure 5.1 shows a motivating example of related code review requests. The example is from the Eclipse EGit project. The developers' names are blinded for privacy. Instead, we use aliases for the developers' identities (i.e. Dev. A and Dev. B). Both code review requests are authored by Dev. A and reviewed by Dev. B (and automatically checked through CI via Hudson). Please note that the 'Reviewers' shows the assigned reviewers (i.e. that who are requested for the review), while 'Committer' shows who has committed the patch for review. These two code review requests are related, as



Change 18476 - **Merged**

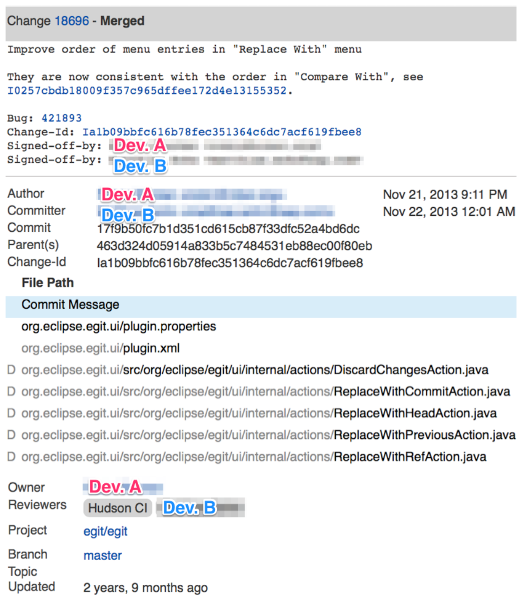
```
Improve order of menu entries in "Compare With" menu

By defining them using the org.eclipse.ui.menus extension point instead
of the deprecated org.eclipse.ui.popupMenus, it's possible to show the
Git-specific entries first. See bug 175693 for why we need to "override"
the menu.

Bug: 421893
Change-Id: I0257cbdb18009f357c965dffee172d4e13155352
Signed-off-by: Dev. A
```

Author	Dev . A	Nov 17, 2013 4:14 PM
Committer	Dev . A	Nov 17, 2013 9:08 PM
Commit	286de3ec27a8bd10087e2002624b38be0762f303	
Parent(s)	5b478b7715c60ec07de845d67786fa6dfb93b59c	
Change-Id	I0257cbdb18009f357c965dffee172d4e13155352	

	File Path
	Commit Message
	org.eclipse.egit.ui/plugin.properties
	org.eclipse.egit.ui/plugin.xml
D	org.eclipse.egit.ui/src/org/eclipse/egit/ui/internal/actions/CompareIndexWithHeadAction.java
D	org.eclipse.egit.ui/src/org/eclipse/egit/ui/internal/actions/CompareWithCommitAction.java
D	org.eclipse.egit.ui/src/org/eclipse/egit/ui/internal/actions/CompareWithHeadAction.java
D	org.eclipse.egit.ui/src/org/eclipse/egit/ui/internal/actions/CompareWithIndexAction.java
D	org.eclipse.egit.ui/src/org/eclipse/egit/ui/internal/actions/CompareWithPreviousAction.java
D	org.eclipse.egit.ui/src/org/eclipse/egit/ui/internal/actions/CompareWithRefAction.java

Owner	Dev . A
Reviewers	Hudson CI, Dev . B
Project	egit/egit
Branch	master
Topic	
Updated	2 years, 10 months ago

Review #18476



Change 18696 - **Merged**

```
Improve order of menu entries in "Replace With" menu

They are now consistent with the order in "Compare With", see
I0257cbdb18009f357c965dffee172d4e13155352.

Bug: 421893
Change-Id: Ia1b09bbfc616b78fec351364c6dc7acf619fbee8
Signed-off-by: Dev. A
Signed-off-by: Dev. B
```

Author	Dev. A	Nov 21, 2013 9:11 PM
Committer	Dev. B	Nov 22, 2013 12:01 AM
Commit	17f9b50fc7b1d351cd615cb87f33dfc52a4bd6dc	
Parent(s)	463d324d05914a833b5c7484531eb88ec00f80eb	
Change-Id	Ia1b09bbfc616b78fec351364c6dc7acf619fbee8	

	File Path
	Commit Message
	org.eclipse.egit.ui/plugin.properties
	org.eclipse.egit.ui/plugin.xml
D	org.eclipse.egit.ui/src/org/eclipse/egit/ui/internal/actions/DiscardChangesAction.java
D	org.eclipse.egit.ui/src/org/eclipse/egit/ui/internal/actions/ReplaceWithCommitAction.java
D	org.eclipse.egit.ui/src/org/eclipse/egit/ui/internal/actions/ReplaceWithHeadAction.java
D	org.eclipse.egit.ui/src/org/eclipse/egit/ui/internal/actions/ReplaceWithPreviousAction.java
D	org.eclipse.egit.ui/src/org/eclipse/egit/ui/internal/actions/ReplaceWithRefAction.java

Owner	Dev. A
Reviewers	Hudson CI, Dev. B
Project	egit/egit
Branch	master
Topic	
Updated	2 years, 9 months ago

Review #18696

Figure 5.1: Two related reviews in Eclipse's EGit project as found in the Gerrit code review system. The developers' names are blinded for privacy.

they refer to the same bug: number 421893. Moreover, the reviewer Dev. B explicitly linked the change of code review request #18696 to code review request #18476 by adding a reference in the commit message to the Change-Id `I0257cbd...5352`, which is reviewed in code review request #18476. The reviewer for code review request #18696 can visit the linked code review request #18476 to get more context and establish review criteria before investigating the review request. Note that the link has been created *manually* by reviewer Dev. B, as no system currently exists to recommend or link related code review requests.

As presented in Chapter 3, we asked developers from open source projects: '*Do you ever consult other/earlier reviews when doing a review?*'. Among open source developers, 83% (81 out of 98) answered that they may consult earlier code review requests when reviewing a newly submitted code review request. As shown in Table 3.2, 36 developers had consulted earlier code review requests to get context for a current code review request. While other reasons exist for consulting past requests, getting context is the one mentioned most often. In a preliminary interview for the survey, a developer stated that *'[Referring to other code review requests] is the most convenient way to understand what is going on in my team and the code base rather than go through the entire code base'*.

Despite the above discussed need to locate and consult related reviews, the authors are not aware of work that supports locating and consulting related reviews, and therefore a code review recommendation approach is presented and evaluated in the rest of this chapter.

5.4 Methodology

This section offers an overview of our methodology for recommending related previous code review requests for a newly submitted patch. It also contains a description of the goal and the research questions we address in this chapter.

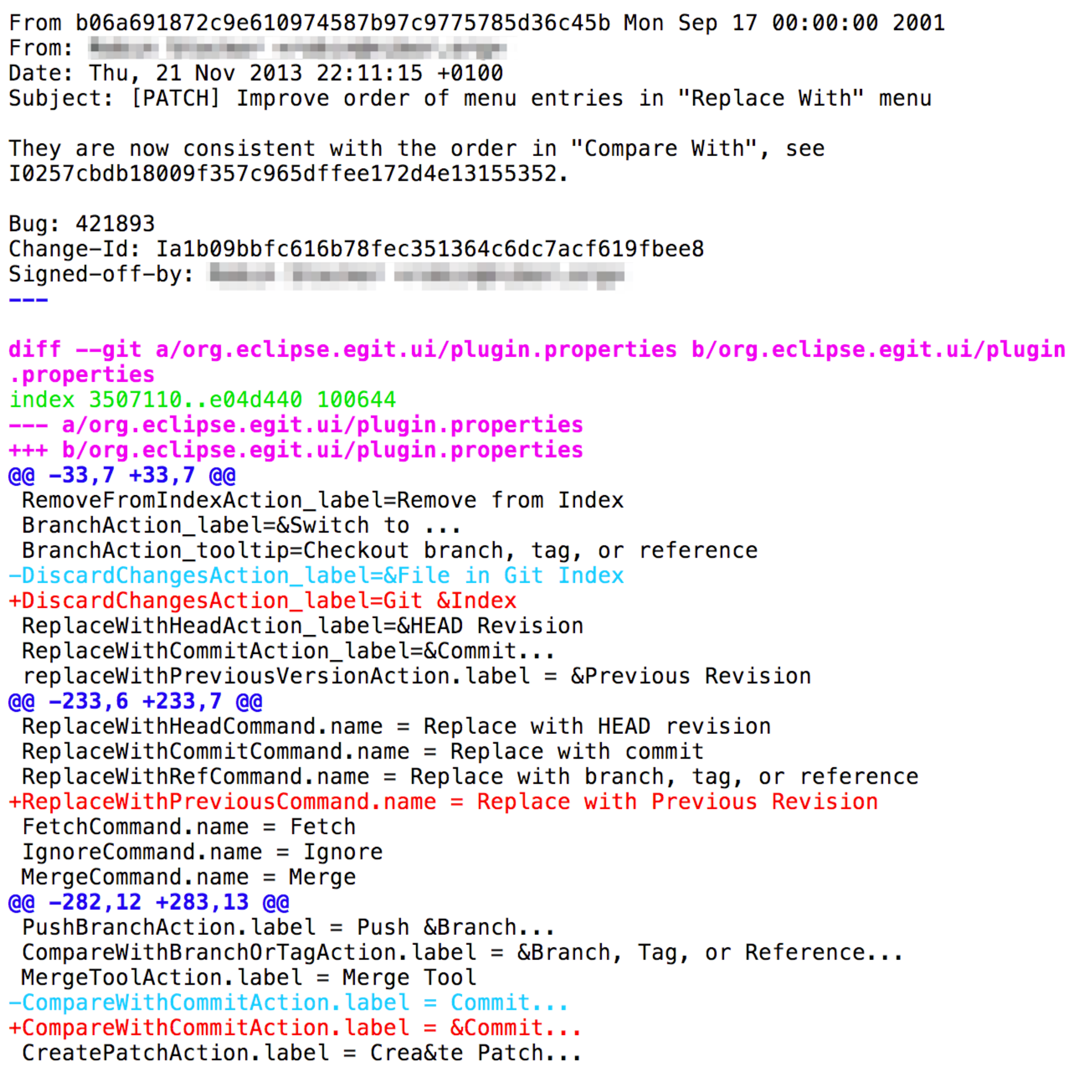
Our approach finds and recommends the most related previous code review requests for any newly submitted patch based on the similarity of the patch fragments. To illustrate, when a developer submits a new code review request with an initial version of a patch, a pairwise comparison is applied between the newly submitted patch and all the patches that have previously been reviewed. Then, we rank all the patches according to similarity. The patches are linked to their code review request, so that the recommendations consist of code review requests with specific patches that contain similar modifications. A developer can investigate the related code review requests to learn what has been important and to decide what to focus on in the freshly submitted patch. Moreover, as the recommendation points to a specific patch in the history of the related code review request, the developer can consider the list of potential issues in the new patch using the issues raised for the recommended patch and in light of how the issues were addressed in the patch's succeeding revisions within the same code review request.

Since our technique relies on the textual similarity, some recommendations may provide no useful information to developers. For example, a developer may receive code review request recommendations that are not actually related but just show a certain level of similarity. Therefore, we define a threshold to filter out the non relevant recommendation results and recommend only those patches with a similarity value greater than the threshold. The choice of the similarity threshold is not straightforward [Ragkhitwetsagul et al., 2016, 2018], and we must find an effective way to choose a threshold. Please note that the threshold can be changed to allow a developer to retrieve more related code review requests or to increase the precision of retrieved patches.

5.4.1 Similarity Measures

To compare the patches (i.e. to compare the newly submitted changes that need to be reviewed with all changes that have previously been reviewed), the similarity of two patches needs to be measured.

We set four requirements for the similarity measures to be used by our approach. First, the similarity measures must generally be applicable to compare patches (diffs) for any type of document, not just source code. For example, patches for HTML, XML or JSON files often occur in reviews. Therefore, any programming language-specific measures cannot be used. Ragkhitwetsagul et al. [2018] have shown that textual similarity measures can perform well on source code with modifications. We use the entire diff text in a patch to compute the similarity without any pre-processing. Figure 5.2 shows an example patch that we used to compute the similarity between patches. A patch consists of two elements that are divided by a separator (‘---’): meta-information and the diff (i.e. the actual changes to files). From the patch we exclude the meta-information (e.g. SHA-1 hash id, date, and commit message) and use only the diff content that represents the actual changes. Since the meta-information uses a consistent form and keywords, it will always have a certain level of similarity, which can be considered as potential noise for our approach. Second, the similarity measures should be normalised. Otherwise, small patches may not get recommendations due to too low similarity values although they actually have related patches (i.e. false-negatives). On the other hand, large patches get recommendations that may not actually be related to the patches (i.e. false positives). Third,



```
From b06a691872c9e610974587b97c9775785d36c45b Mon Sep 17 00:00:00 2001
From: [redacted]
Date: Thu, 21 Nov 2013 22:11:15 +0100
Subject: [PATCH] Improve order of menu entries in "Replace With" menu

They are now consistent with the order in "Compare With", see
I0257cbdb18009f357c965dffee172d4e13155352.

Bug: 421893
Change-Id: Ia1b09bbfc616b78fec351364c6dc7acf619fbee8
Signed-off-by: [redacted]
---

diff --git a/org.eclipse.egit.ui/plugin.properties b/org.eclipse.egit.ui/plugin
.properties
index 3507110..e04d440 100644
--- a/org.eclipse.egit.ui/plugin.properties
+++ b/org.eclipse.egit.ui/plugin.properties
@@ -33,7 +33,7 @@
 RemoveFromIndexAction_label=Remove from Index
 BranchAction_label=&Switch to ...
 BranchAction_tooltip=Checkout branch, tag, or reference
-DiscardChangesAction_label=&File in Git Index
+DiscardChangesAction_label=Git &Index
 ReplaceWithHeadAction_label=&HEAD Revision
 ReplaceWithCommitAction_label=&Commit...
 replaceWithPreviousVersionAction.label = &Previous Revision
@@ -233,6 +233,7 @@
 ReplaceWithHeadCommand.name = Replace with HEAD revision
 ReplaceWithCommitCommand.name = Replace with commit
 ReplaceWithRefCommand.name = Replace with branch, tag, or reference
+ReplaceWithPreviousCommand.name = Replace with Previous Revision
 FetchCommand.name = Fetch
 IgnoreCommand.name = Ignore
 MergeCommand.name = Merge
@@ -282,12 +283,13 @@
 PushBranchAction.label = Push &Branch...
 CompareWithBranchOrTagAction.label = &Branch, Tag, or Reference...
 MergeToolAction.label = Merge Tool
-CompareWithCommitAction.label = Commit...
+CompareWithCommitAction.label = &Commit...
 CreatePatchAction.label = Crea&te Patch...
```

Figure 5.2: Patch example. Developers' names are blinded for privacy.

Table 5.1: Evaluation result of the three similarity measures by using Ragkhitwetsagul et al.'s framework and benchmark data [Ragkhitwetsagul et al., 2018]

	Precision	Recall	Accuracy	F-measure
Jaccard	0.891	0.884	0.978	0.888
Sørensen–Dice	0.885	0.890	0.977	0.887
Cosine	0.497	0.774	0.899	0.605

the similarity measures must be computed efficiently, since our technique uses pairwise comparison at the moment. If a similarity measure has higher than linear complexity (i.e. $O(n)$), it will become difficult to provide useful recommendations to developers in a timely manner as more and more code review data is accumulated. Lastly, the similarity measures should achieve high fidelity in their results.

Our approach uses a programming-language-agnostic off-the-shelf similarity measure, the java-string-similarity library[2], a library implementing different string similarity and distance measures, rather than using dedicated code similarity tools (e.g. a code clone detector). A dozen algorithms, including Levenshtein edit distance and siblings, Jaro-Winkler, Longest Common Subsequence, N-grams, Q-grams, cosine similarity, Jaccard index, and Sørensen–Dice coefficient, are implemented and ready to be used. Although multiple algorithms are available in the library, only three of them satisfy our requirements: Jaccard index, Sørensen–Dice coefficient, and cosine similarity. All other algorithms are either not normalised or are too slow (having a complexity higher than $O(n)$). The library removes space from the strings to be compared (pre-processing) and then generates n-gram [Kondrak, 2005]. Except for space, everything else is preserved (comments, keywords, punctuation, and symbols). The (space-less) strings are converted into sets of n-grams (sequences of n characters), and then the similarity between the two sets is measured. Our approach uses the library's default value of n, which is 3.

We evaluated the three candidate similarity measures from java-string-similarity by using Ragkhitwetsagul et al.'s framework and benchmark data Ragkhitwetsagul et al. [2018], which have been used to compare 30 well-known code

[2]https://github.com/tdebatty/java-string-similarity

similarity analysers. Table 5.1 presents the evaluation results. The Jaccard index and Sørensen–Dice coefficient (over 3-grams) show better performance than most of the 30 other algorithms, including clone detectors such as Deckard [Jiang et al., 2007] and Simian [Harris, 2015]. Only CCFinderX [Kamiya et al., 2002] showed better results than the Jaccard index and the Sørensen–Dice coefficient, however, CCFinderX does not satisfy our requirements because the technique is language-specific and has a high complexity.

We chose the Jaccard index to generate examples to evaluate our technique because it is widely used in research areas involving similarity measures (e.g. duplicate detection [Hajishirzi et al., 2010], clone detection [Keivanloo et al., 2014], and software plagiarism detection [Chae et al., 2013]) and because of its very good performance, as presented in Table 5.1.

5.4.2 Goal and Research Questions

The goal of our study is to evaluate the effectiveness of our approach in recommending related code review requests based on similarities between the patch fragments. To achieve our goal, we address the following research questions:

RQ1: Do the recommended patches provide useful information during code review? Although we mentioned our recent survey result in Section 5.3, we do not know whether developers can use our technique in practice, as there is no known tool or technique to recommend related code review requests, to the best of our knowledge. Therefore, we tested our technique by interviewing seven software development engineers (SDEs) from a team at Amazon. We showed them examples of actual recommendations generated from internal Amazon code review data, and we interviewed them, asking questions about whether the recommendations could be understood and how useful they were.

RQ2: How precisely are the recommended patches related to the submitted patches? Although RQ1 evaluates the applicability of the recommendation technique via developer interviews, the study has two threats to validity. First, we interviewed the developers who are directly involved in the projects.

Since the developers have enough knowledge and context in the projects, the results from them might be subjective. Second, the interviews rely on industrial review data which means the result may not be applicable to open source projects. To mitigate the threats to validity, we evaluated the precision of our approach for recommending related code review requests through manual inspection of the actual recommendation results from three Eclipse projects.

RQ3: How does the similarity threshold affect the accuracy of the results? Although our approach does not rely on a specific threshold, we used a conservative threshold (discussed later in this section) for both RQ1 and RQ2. While RQ1 and RQ2 are answered with a threshold, RQ3 investigates how sensitive the recommendations are to the threshold in terms of the number of recommendations, precision, and recall.

RQ4: How robust is the result compared to other similarity measures with different thresholds? Although we used the Jaccard index because of its high fidelity, it may affect the robustness of our approach (i.e. the recommendations may be different when using other similarity measures). Therefore, we evaluate the robustness of the Jaccard index by comparing the results with results from the Sørensen–Dice coefficient and cosine similarity, since they also satisfy our similarity measure requirements.

5.4.3 Experimental Setup

To answer the research questions, we selected an Amazon internal project (RQ1) and three projects from the Eclipse foundations (RQ2–RQ4). For the interview study at Amazon (RQ1), we selected an internal web project that is actively used by internal Amazon developers on a daily basis.[3] The three projects from the Eclipse foundations are EGit, Linux Tools, and JGit. These projects had had the greatest number of code review requests in Gerrit at the time of the experiment. Our experimental evaluation used a *replay* approach in which all extracted patches were assumed to be freshly submitted, one-by-one, in the order of the original submission. For the projects, we extracted all code review requests and their patches. Each patch was then used as a query,

[3]Due to the company's policy, we can not present the details here.

Table 5.2: Collected data sets for each project

	EGit	JGit	Linux Tools
# of Review Requests	4,752	4,408	4,546
# of Patches	7,050	6,457	9,232
Average # of patches/requests	1.48	1.46	2.03
Median # of patches/requests	2	2	1

and our approach compared the query to all previously submitted patches. Table 5.2 shows the number of code review requests and patches for the three data sets from the Eclipse foundations. We crawled code review data from the beginning of the Gerrit repository until May 2016. As of August 2017, the project we selected from Amazon had 7,680 patches reviewed in the internal code review tool, which is comparable to the number of patches in the projects from Eclipse.

To establish the threshold used in RQ1 and RQ2, we considered the distribution of the similarities for each project individually. For each patch, we extracted the most similar previous patches from different code review requests. Patches are expected to have a certain level of similarity to other patches in the same project, since they share the same programming language and the same terminology. Because a generous threshold may introduce improper recommendations (i.e. more false positives), we chose a conservative threshold. We considered similar patches to be outliers of the similarity distribution over patches based on the assumption that related patches will show unusually high similarity values. To detect outliers, we used Tukey's method [Tukey, 1977] and defined the upper outlier fence as $1.5 \times$ IQR (interquartile range) from the third quartile (Q3). We assumed measured similarities higher than the upper outlier fence to be indicative of actual similarity. The main goal in adopting Tukey's method was to set a threshold to answer RQ1 and RQ2. Figure 5.3 shows the similarity measure distribution for the Jaccard index. The x-axis denotes different projects while the y-axis shows normalised similarity values from 0 to 1. We present the combination of a violin plot and a box plot in the same figure to clearly represent data population and quartiles simultaneously. The similarities are widely distributed and outliers are clearly visible. Since there are many closely neighbouring outliers, the outlier representations in the figure look like bold vertical lines.

With Tukey's method, we established the thresholds for each project: 37% for EGit, 34% for JGit and 36% for Linux Tools. Note that the thresholds are close to each other despite being project specific.

To address RQ1, we interviewed seven developers in a team at Amazon. The team follows most recommended practices at Amazon. We extracted recommendation examples from the code review data stored in the Amazon internal code review tool. The interviews were a semi-structured 1:1 interviews between a developer (interviewee) and the author of this thesis (interviewer). The semi-structured interview is an interview technique that leverages general topics and questions rather than relying on an exact set and order of questions. Semi-structured interviews are widely used to determine what is happening and to seek new insights [Lindlof and Taylor, 2011, Bacchelli and Bird, 2013]. We asked developers to review three code review requests with recommendations extracted via our approach and to narrate their findings. The interviewer's role was just to follow interviewees' activities. To minimise intrusion by the interviewer, we limited the interviewer's role to motivate developers to keep narrating their findings by asking simple questions (e.g. Which line are you looking at? What are your findings in this example?). The interviewer was prohibited from providing potential clues to the relationships between code review requests. We set up a 27-inch monitor with two separate browser windows. A newly submitted code review request was located on the right side of the screen, while the recommendation result was shown on the left side. For each interview, we declared that the interview result would be handled anonymously, and we recorded the interview scripts with the interviewees' consent.

For the interviews, we conducted purposive sampling [Teddlie and Yu, 2007] to select three recommendation examples with the below criteria:

- The similarity between the two patches should exceed the threshold (i.e. the upper outlier fence).
- The number of changed lines in the recommended patch and the newly submitted patch should be less than 50 in each. Otherwise, a developer would have to spend more time to comprehend the changes, which would reduce the interview time with the developer.
- On the other hand, the number of changed lines in the recommended patch and the newly submitted patch should be greater than 10 in

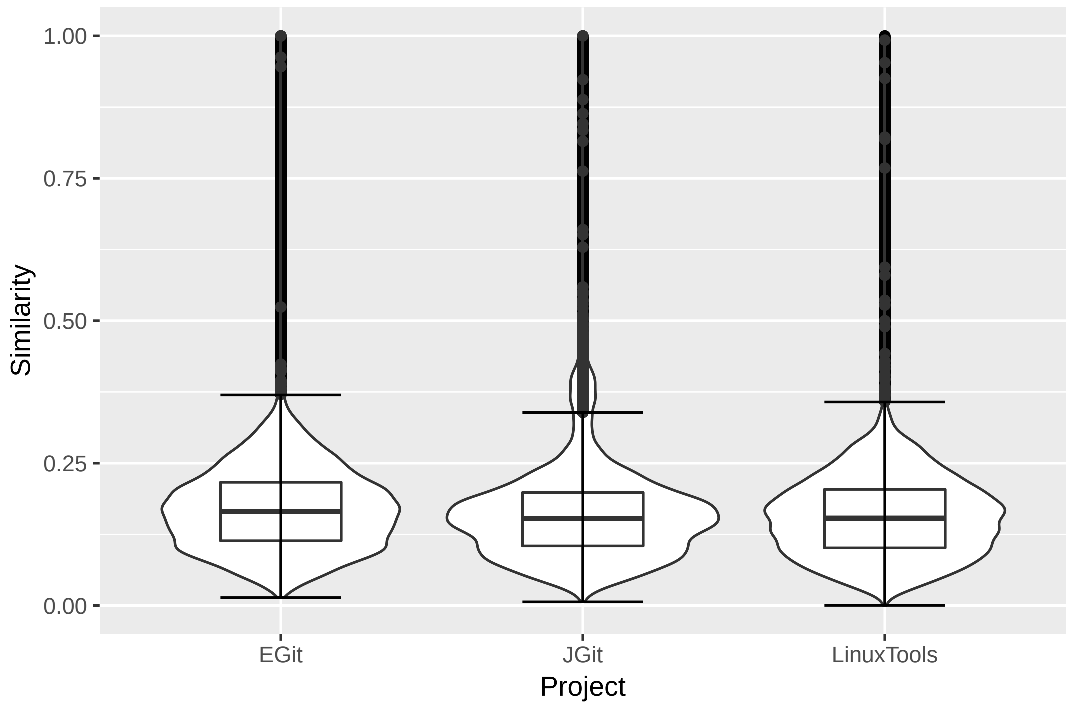


Figure 5.3: Similarity measure distribution for each individual patch.

each, since it can be hard to understand a code review request (either a recommended or a new one) if too little information is provided by the patch.

- Each recommended code review request should have more than two comments submitted by developers. If a code review request contains no comment, it denotes that no context was recorded during the code review.
- Both the recommended code review request and the newly submitted code review request should have been submitted at least 2 months before the interview date. If the examples had been recently submitted, developers might still have been familiar with the code review requests if they had been involved in the reviews.

To select the interviewees, we contacted a development team at Amazon and interviewed the team's developers. The team develops and maintains the internal services that are used by Amazon developers on a daily basis. Ta-

Table 5.3: The interviewees' experience at Amazon and interview duration

SDE	Experience at Amazon	Interview time
SDE#1	5 years 2 months	12 minutes 59 seconds
SDE#2	10 months	15 minutes 11 seconds
SDE#3	5 years 2 months	22 minutes 21 seconds
SDE#4	7 years 2 months	8 minutes 10seconds
SDE#5	3 years 1 month	12 minutes 52 seconds
SDE#6	4 months	24 minutes 01 second
SDE#7	2 years	11 minutes 06 seconds

ble 5.3 shows the interviewed SDEs' experience at Amazon and the interview time duration. The developers have experience within Amazon ranging from 4 months (SDE#6) to more than 7 years (SDE#4). As shown in the Table 5.3, interviews varied in their duration. The mean interview time was 15 minutes 14 seconds. All developers used a side-by-side diff view (a screen split into two parts with the left side presenting the code before the change and the right side showing the code after the change) for the interview except SDE#3. SDE#3 used an inline view (visualising the change as a unified diff).

Since there is no ground truth for related patches (i.e. no correct answer set), we could not evaluate the recall of our approach. If a recommendation technique shows a high precision, it may have a lower recall. Therefore, we computed the relative recall by using a partial ground truth to measure the sensitivity of our technique [Jiang et al., 2014]. To construct a partial ground truth, we extracted the Change-Ids from the code review data and identified related patches. If a patch contained a Change-Id of another patch (i.e. the Change-Id of another patch appeared in the description or comments), we assumed that they were related. We extracted 406, 936, and 1,265 related patches in this way from EGit, JGit, and Linux Tools, respectively. The relative recall was computed as the ratio of recommendation pairs that have a similarity value higher than the threshold and appear in the partial ground truth for the number of pairs in the partial ground truth. Remember that Change-Ids only appear in the description or comments of another review if they have been manually identified and inserted, thus they represent only that part of the ground truth that the developers were aware of and where they considered the relationship important enough to make it explicit. Because identifying related changes requires manual effort, the actual number of related patches is expected to be higher.

Based on the partial ground truth, we created a balanced data set to evaluate the accuracy of our recommendation technique in terms of both precision and recall. We randomly selected patch pairs that did not have evidence of a relationship to match the numbers of pairs used as the partial ground truth (i.e. 406, 936, and 1,265 patches for EGit, JGit, and Linux Tools, respectively). Because most patch pairs are not related, the random selection of pairs that are not in the ground truth will be very likely return pairs that are not related. For example, we extracted 406 random patch pairs that did not have clear evidence of a relationship for EGit since EGit has 406 pairs in the partial ground truth. By merging 406 randomly selected pairs and the 406 pairs in the partial ground truth, we got 912 pairs in a balanced data set, i.e. 50% of pairs were related while 50% of pairs were (very likely) not related. We could then measure the precision and recall based on the balanced data set. Since this evaluation data contained randomly selected patch pairs, we repeated the evaluation 100 times [Lee et al., 2011, 2016], each time with a newly generated, balanced data set.

5.5 Results

In this section, we answer the research questions raised in the previous section. First, we test whether our technique can provide useful information during code review by interviewing developers at Amazon with actual recommendation examples (RQ1). We evaluate our technique by manually investigating how many recommended code review requests are actually related to a submitted patch (RQ2). Finally, we compare the recommendation results using different thresholds (RQ3) and among three different similarity measures (RQ4).

Table 5.4: Recommendation examples for the interviews. 'External' means a code review request was authored or reviewed by a developer on the external team; otherwise, we specify SDE alias

Example	Date gap	Recommendation			New code review request	
		# comments	Author	Reviewer	Author	Reviewer
R1	4 days	2	SDE#7	SDE#4	External	SDE#7 & External
R2	40 days	22	External	SDE#7 & External	External	SDE#7 & External
R3	1 day	20	External	SDE#7 & External	External	SDE#7 & External

5.5.1 RQ1: Do the recommended patches provide useful information during code review?

Table 5.4 presents details about three selected examples. Please note that we used the Jaccard index to extract the recommendations. The project we investigated is a developers' tool that is maintained by multiple teams, but mainly by the team we contacted. Therefore, developers from other teams (shown as 'external' in the table) also contribute to the project. The date gap column shows the number of days between the recommended code review request and the newly submitted code review request. The '# comments' column represents the accumulated number of comments across all revisions in the recommended code review request. Please note that all seven SDEs received optional reviewer requests,[4] although SDE#4 and SDE#7 seemed to be the only reviewers from the team for the three examples. SDE#4 and SDE#7 were also the final approvers of the code review request.

Since we cannot present the actual example in this chapter because of the company's policy, we explain each example and discuss the SDEs' comments.

The *R1* example in the table represents a *reverting changes* scenario. The recommended previous review is a temporary bug fix that disabled a problematic feature, while the newly submitted review is a clean fix for the bug and re-enables the feature. However, we could not find a specific reference to the bug report or to a previous code review request in either the target code

[4]The internal review tool supports sending review requests using a team alias. In such cases, all SDEs on a team receive code review request notifications. The project we studied adheres to the practice of using the team alias for all review requests.

review request or the recommended code review request. The two comments in the recommended code review request describe the author's intention (i.e. 'I tried to fix...'). All interviewers could correctly identify the relationship between the recommendation and the newly submitted code review request. Although SDE#4 reviewed the recommended code review request at the time, he could not specifically remember the code review request. SDE#7 commented on this example:

> *I think there is another code review request before this one [the recommended code review request]. I wrote this change [the recommended code review request] to fix bug 'A' introduced by the previous one, but, my fix introduced bug 'B' then, this fix [newly submitted code review request] is submitted to fix 'B'.*

We confirmed that our technique also recommended the previous code review request that introduced bug '*A*', although its patch had a lower similarity value than the patch with the recommended review request used in the example.

The *R2* example is a *change similar location* scenario. Although the *R2* recommended request does not handle the same issue, we could see that the changed files and their locations in the newly submitted code review request and the recommended code review request were very similar. As shown in Table 5.4, this example contained the greatest number of comments. SDE#6 said that:

> *Both review requests were written by the same developer, and I can find some comments regarding style. So I think I can use the comments on the recommendation as guidelines to review the new code review request*

One interesting observation is that SDE#4 and SDE#5 immediately collapsed (i.e. hide) the comments on the recommended code review request. The interviewer asked them why they had collapsed the comments. SDE#4 stated that:

> *I've been working on the project from the beginning and am familiar with the context. So, I more focus on the change itself, but, I could see how changes are evolved from your recommendation. It looks like the Git blame feature for code review.*

On the other hand, SDE#5 provided following explanation for why he collapsed the comments:

> *I usually collapse the comments when I review code changes because I don't want to be biased by others comments.*

The example of SDE#4 may imply that our technique is more useful for novice developers than for experienced developers. In addition, the example of SDE#5 calls for further work on how to support developers in producing non-biased reviews.

The last example, *R3*, is a *developer's mistake* scenario. The author of *R3* had just joined the company and was not familiar with the internal code review tool. Although he needed to submit a new revision of the code review request that was recommended by our technique, he created an entirely new code review request instead (the one for which the recommendation was generated). This is why the date gap between code review requests is very short (i.e. just a day). Even worse, these two review requests have different titles. All interviewees mentioned that the two code review requests look very similar, almost identical. SDE#1 said:

> *The title of code review requests are different and commit IDs are different. So it seems like a totally different review request, but, the changed parts are the same. It is very weird.*

At the end of the interview, we asked for general feedback on our approach. All interviewees agreed that our proposed technique can provide useful information to review a new code review request. Table 5.5 shows the interview overview for each SDE. The 'relationship' column represents whether an SDE could locate the relationship between the code review requests. If an SDE could locate that relationship between code review requests, we note the example id (i.e. *R1*, *R2*, and *R3*). The 'find intention' column denotes whether an SDE could determine the intention of the previous (i.e. recommended) changes. The 'check comment' column indicates whether an SDE extracted context from the comments in a recommended code review request. As we mentioned above, SDE#4 and SDE#5 did not investigate the comments of the recommended code review requests. Useful information as demonstrated by the interviews is given below:

Table 5.5: The interview overview for each SDE

SDE	Relationship	Find intention	Check comment
SDE#1	*R1*, *R2*, *R3*	*R1*, *R2*, *R3*	*R1*, *R2*, *R3*
SDE#2	*R1*, *R2*, *R3*	*R1*, *R2*, *R3*	*R1*, *R2*, *R3*
SDE#3	*R1*, *R2*, *R3*	*R1*, *R2*, *R3*	*R1*, *R2*, *R3*
SDE#4	*R1*, *R2*, *R3*	*R1*, *R2*, *R3*	–
SDE#5	*R1*, *R2*, *R3*	*R1*, *R2*, *R3*	–
SDE#6	*R1*, *R2*, *R3*	*R1*, *R2*, *R3*	*R1*, *R2*, *R3*
SDE#7	*R1*, *R2*, *R3*	*R1*, *R2*, *R3*	*R1*, *R2*, *R3*

- **Intention of the previous change**: In the *R1* example, the intention of the recommended code review request was the reversion of a previous change. The developers could understand the intention of previous changes.
- **Link to the previous change**: As shown in the *R1* and *R3* examples, the developers could locate previous code review requests that handle the same issue without the use of additional metadata.
- **Review criteria**: In the *R2* example, the developers could consider other reviewer's comments in the recommended code review request. The comments can highlight common mistakes and provide review criteria.

In addition, we could confirm that our technique can be useful for code review request authors as well. SDE#7 stated that:

> *This technique can benefit not only reviewers but also authors. Before the authors send a review request, they can evaluate their changes by using this technique.*

The interviewees pointed out that sometimes it is difficult to locate useful information from the recommendation, since there is no user interface (UI) support. Integrating proper UI support for our technique is a task for future work. SDE#1 recommended the following:

> *Through the three examples, I can understand how your technique works, but, it is hard to locate similar changes from the recommendation since there is no UI support. It would be great to integrate the UI for the recommendation such as highlighting the same file changed in both code review requests*

Table 5.6: A statistical summary of the evidence categories and the types of evidence within each category

	EGit		JGit		Linux Tools	
Total Sample Size	204		302		169	
Bug id	29.9%	(61)	17.2%	(52)	3.6%	(6)
Topic	7.8%	(16)	–	–	–	–
Related change	17.7%	(36)	9.6%	(29)	9.5%	(16)
Recommended Change-Id	8.8%	(18)	14.2%	(43)	5.3%	(9)
Same Change-Id	18.6%	(38)	33.1%	(100)	23.1%	(39)
Recommended Change-Id in comments	18.1%	(37)	22.9%	(69)	7.1%	(12)
Description in the commit message	1.0%	(2)	3.3%	(10)	6.5%	(11)
Todo in code	2.5%	(5)	–	–	–	–
Total Evidence	82.4%	(168)	85.4%	(258)	52.1%	(88)

From the three examples, developers could identify the relationship between recommendations and newly submitted code review requests and gain useful information from our recommendation.

5.5.2 RQ2: How precisely are the recommended patches related to the submitted patches?

To answer RQ2, we investigated whether the recommended code review requests actually were related to the new patch. For each project, we randomly sampled 45 code review requests from among those code review requests with patches that received recommendations based on the $Q3 + 1.5 \times IQR$ threshold. We manually analysed them to identify whether the recommended code review requests are actually related. For each recommended code review request of the sample, we manually investigated the details and categorised how the recommended code review request is actually related to the sampled code review request. One of the collaborators in this research (i.e. the investigator) manually investigated the metadata, comments, and the actual changes of the code review request and recommended code review requests in pairs. The investigator recorded evidence that clarified the relationship between the two code review requests in a pair. Based on this recorded evidence, we categorised the relationship. To mitigate subjective bias, the thesis author reviewed the results with the investigator in a cross-validation

Table 5.7: A list of different relationship categories for the suggested patch sets

Category	**EGit**		**JGit**		**Linux Tools**		**Total**	
	Same files	Other files	Same files	Other files	Same files	Other files	Same files	Other files
Change of similar code	–	1.0% (2)	3.0% (9)	1.3% (4)	5.3% (9)	5.3% (9)	2.7% (18)	2.2% (15)
Related change	2.9% (6)	14.7% (30)	3.0% (9)	6.6% (20)	10.1% (17)	–	4.7% (32)	7.4% (50)
Revert change	2.9% (6)	–	5.3% (16)	–	4.1% (7)	–	4.3% (29)	–
Fix the same bug	3.9% (8)	–	11.3% (34)	–	–	–	6.2% (42)	–
Same change submitted twice (with minor change)								
- Same branch	26.0% (53)	–	24.5% (74)	–	26.0% (44)	–	25.3% (171)	–
- Different branch	31.9% (65)	–	33.1% (100)	–	25.4% (43)	–	30.8% (208)	–
Change in similar location	3.4% (7)	–	3.6% (11)	–	–	–	2.7% (18)	–
Refactoring	5.9% (12)	–	7.6% (23)	–	21.9% (37)	–	10.7% (72)	–
Fix newly introduced bug	1.0% (2)	–	0.3% (1)	–	–	–	0.4% (3)	
Update metadata	3.9% (8)	0.5% (1)	0.3% (1)	–	1.7% (3)	–	1.8% (12)	0.1% (1)
Not related	–	**2.0% (4)**	–	–	–	–	**0.6% (4)**	–

manner. In total, the 135 code review requests of the sample received 675 related code review requests.

During the manual investigation, we found a large amount of evidence that the two code review requests were actually related. Table 5.6 presents the evidence we found for each recommendation pair in terms of the recommended related code review requests. Please note that the total sample size is the number of recommended pairs, as most of the 45 sampled code review requests received more than one recommendation. The first three rows show the number of pairs that share the same *Bug* id or *Topic* information in the metadata of the paired code review requests, or where the recommended code review request appears in the *Related Changes* list. The next three rows show the numbers of pairs in which the code review requests mention a Change-Id. In the case of *Recommended Change-Id*, the Change-Id of the recommended code review request appears in the metadata (commit message) of the sampled code review request—that is, the author of the sampled code review request explicitly linked it to the (recommended) previous code review request. In the case of *Same Change-Id*, the sampled and the recommended requests both explicitly mention in their metadata the same Change-Id of a third code review request. In the case of *Recommended Change-Id in comments*, the Change-Id of the recommended code review request appears in a review comment: in other words, a reviewer of the sampled code review request explicitly linked it to the (recommended) previous code review request. Lastly, in the *Todo in code* case, we found information that was related to the recommended code review request appearing in the patch as a todo task. As shown in the table, overall, we could find clear evidence relating 168 (EGit), 258 (JGit), and 88 (Linux Tools) recommended pairs. However, we could not find clear evidence for from 14.6% to 47.9% of the recommended review pairs. This does not mean that they are not related, only that there is no *clear* evidence.

In addition, to identify reasons beyond clear evidence of a relationship, we manually investigated the recommended pairs for different relationship categories. Table 5.7 presents the list of relationship categories we identified for the suggested patches. We separately counted the numbers of recommended pairs based on their file location (i.e. whether the recommended patch modifies the same files as the sampled patch or other files). In the *Change of similar code* category, the paired patches modify similar code in a similar way. The *Related change* row shows the numbers of recommended patches in which the sampled review request contains related change information as manually labelled by the developer (clear evidence as above). *Revert change* represents the number of patches that are similar because a developer reverted the change of the recommended patch. In the *Fix the same bug* category, the two paired patches fix the same bug. In the case of *Same change submitted twice (with minor differences)*, the same patch with only minor differences was detected in the paired code review requests, aiming at the *same branch* or at a *different branch*. In the *Change in similar location* category, the two paired patches make a change at a very similar location in the source code, although the actual content of the changes differs. In the case of *Refactoring*, the two paired patches perform the same refactoring, such as renaming a method's name. *Fix newly introduced bug* represents the situation in which the later patch was submitted to fix a bug that had been introduced by the recommended patch. Lastly, in the *Update metadata* case, the paired patches perform a similar update of metadata, such as a project version number. Whenever developers update metadata, they must change all files that contain the metadata.

As the table shows, only 4 (2%) out of the manually investigated 204 cases of suggested patches in EGit have no relationship (i.e. are false positives). Surprisingly, we could not find any false positives in the manually inves-

Table 5.8: Ratios of resubmitted identical patches by reason. The parenthetical values show the numbers of resubmitted patches

Target branch	Category	EGit		JGit		Linux Tools	
Different branch	Cherry pick	77.6%	(38)	70.8%	(17)	86.5%	(217)
	Merge	16.3%	(8)	12.5%	(3)	11.2%	(28)
Same branch	Working on same base	4.1%	(2)	4.2%	(1)	2.0%	(5)
	Mistake	2.0%	(1)	–	–	0.4%	(1)
	Take over	–	–	12.5%	(3)	–	–

tigated recommendation pairs from the JGit and Linux Tools projects. We investigated the four false positives from EGit and found that the actual changes were not related, but a huge file change from outside the code review request was found in a specific revision of a patch. For example, code review request #3645 is recommended as a related request for code review request #16761. We found that patch set #5 of code review request #16761 contains 1,662 changed lines in uitext.properties, while the other patch sets in the code review request contain only two changed lines in the same file. However, the 1,662 changed lines are part of the patch to fix an unresolved conflict which was present in uitext.properties at the time of the patch (it is not clear how it is possible that such an unresolved conflict was presented in a committed revision). Since the code review request #3645 changed 442 (similar) lines in uitext.properties, it was recommended despite it is not being related.

Our approach recommended related code review requests with 98% of precision in the 45 sampled review requests from the EGit project and 100% among the sampled code review requests from the JGit and Linux Tools projects based on the $Q3 + 1.5 \times IQR$ threshold. Moreover, for 82.3%, 85.4%, and 52.1% of the investigated recommended code review requests in the EGit, JGit and Linux Tools projects, respectively, we found clear evidence of a relationship between the paired review requests.

Overall, our manual investigation of the 675 recommended related patches demonstrates that our approach can effectively recommend related code review requests.

While manually investigating the recommendation results, we found patches that are identical to earlier patches in different code review requests (i.e. identical patches are resubmitted as new changes). Among the investigated projects, 0.7% (49/7,050) of EGit patches, 0.4% (24/6,457) of JGit patches, and 2.7% (252/9,232 patches) of Linux Tools patches are resubmitted as new code review requests. Note that these numbers do not include resubmitting identical patches resubmitted under the same code review request,[5] as including them would result in 1,236 identical resubmissions for EGit, 1,364 for JGit, and 1,610 for Linux Tools.

As we found that non-ignorable numbers of patches are resubmitted, we manually inspected to learn why. The manual investigation was performed by one of the collaborators of in this research, and the results have been validated by the author of this thesis. Table 5.8 lists the categories explaining why patches are resubmitted and the numbers of patches belonging to each category. We classified the reasons into two broad classes: relocating to a different branch and resubmitting to the same branch. From 83.3% to 97.6% of identical patch resubmissions are the result of relocating changes to a different branch by cherry-pick or merge. Both cherry-pick and merge are Git features that migrate particular commits from one branch into a different branch. Sometimes, identical patches are resubmitted in the same branch. We classified the resubmission of identical patches to the same branch into three categories. The *Working on the same base* category describes cases in which developers want to improve the same patch in different code review requests. For example, we can assume two different bug issues rely on a patch. Developers want to improve the same patch in different directions and merge both patches later. The *Mistake* category shows developers mistakenly submitting a patch in the wrong code review request. The *Take over* category occurs when the patch author changes while a slightly changed patch is resubmitted as a new code request. We observed this case in the JGit project. A developer stopped writing new patches in a code review request. Another developer then took over the patch and submitted an improved version of the patch as a new review request.

[5]Resubmitting the same patch again into the same code review request occurs often in code review—for example, to restart a failed CI run.

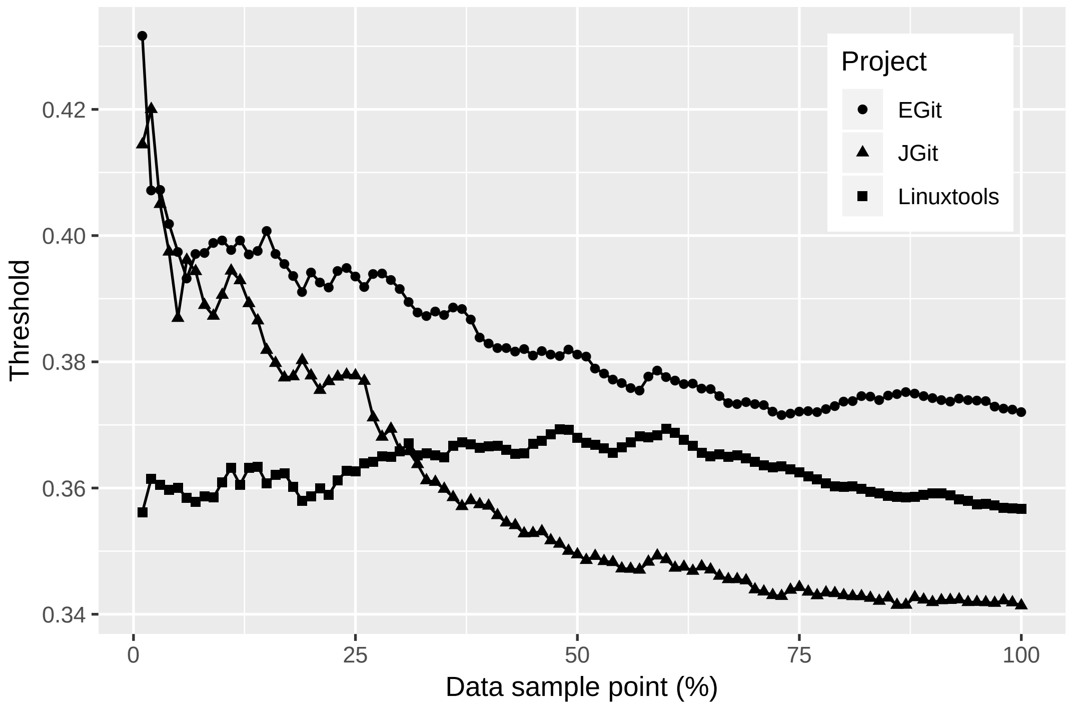


Figure 5.4: Q3 $-$ 1.5 $\times$ IQR thresholds over the different number of patches

5.5.3 RQ3: How does the similarity threshold affect the accuracy of the results?

Although our approach uses a customisable threshold to establish a minimum similarity that a pair of patches has to have to report the second as a recommendation, we chose a fixed threshold (i.e. Q3 + 1.5 $\times$ IQR) to answer RQ1 and RQ2. However, the threshold should be adapted by practitioners to their individual needs. Therefore, we studied the sensitivity of the recommendations to the similarity threshold.

It is well-known that recommendation techniques can suffer the cold start problem (i.e. an insufficient number of data points will lead to poor recommendation results) [Schein et al., 2002]. If the number of patches is small, we might not select a meaningful threshold for our evaluation. Figure 5.4 shows the variation of threshold as defined by Q3 $-$ 1.5 $\times$ IQR with an increasing number of patches. The x-axis shows the number of patches (i.e. data sam-

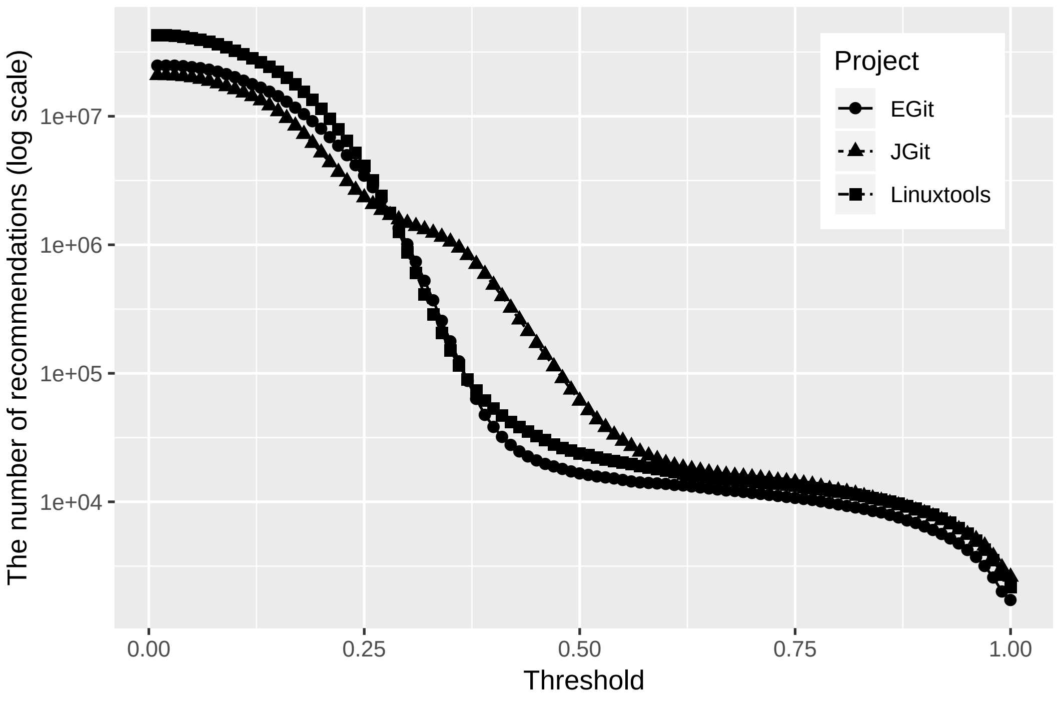


Figure 5.5: Number of recommendations that are made based on different thresholds

ples) used for the threshold computation (from 1% to 100%) while the y-axis shows the corresponding Q3 $-$ 1.5 $\times$ IQR similarity threshold (from 34% to 43%) extracted from the number of patches considered. Please note that we use the ratio of patches instead of the actual number since the projects have different numbers of patches. The patches are sorted by submission order. As shown in the figure, EGit and JGit show sharp decrements and fluctuations with less than 10% of data samples because of the cold start problem. Although the thresholds vary based on the different data sample points, the variation is less than 0.04 above the 25% data sample point for all three projects. Moreover, for almost all data points above 10%, the three thresholds are close within a 0.04 range.

Figure 5.5 shows the number of recommendations produced based on different similarity thresholds. The x-axis shows the similarity threshold from 0 to 1 while the y-axis denotes the number of recommendations that are made with the corresponding threshold on a logarithmic scale. Please note that the

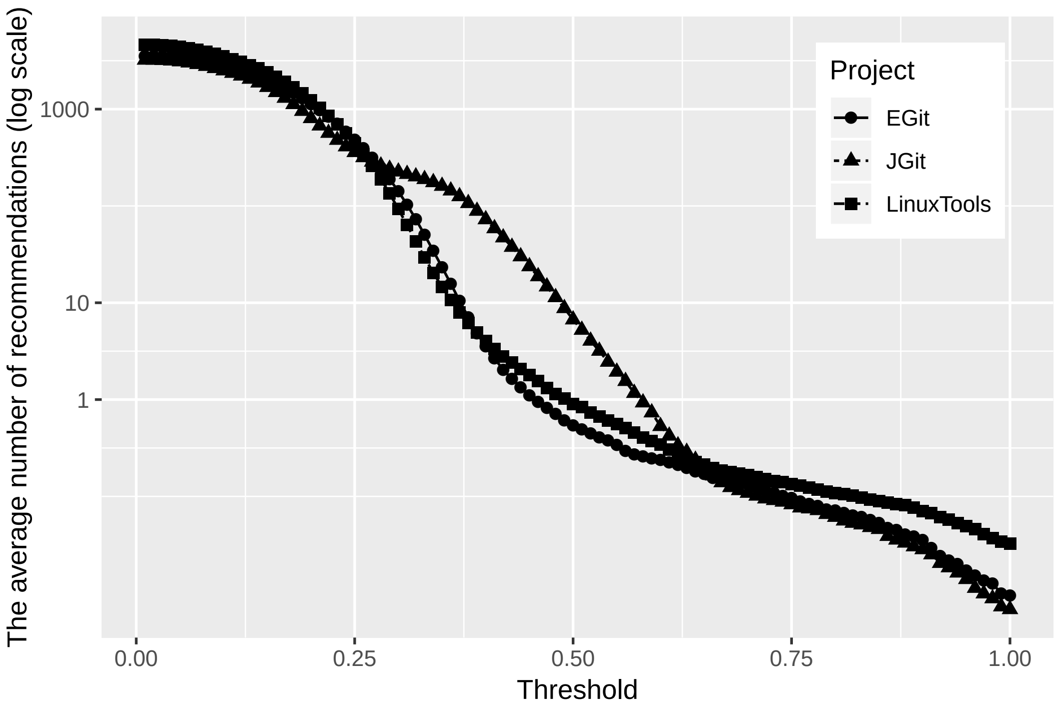


Figure 5.6: Average number of recommendations per patch based on different thresholds

numbers of recommendations are extremely large because we adopted pair-wise comparison by replaying the code review history — that is, comparing a patch to all preceding patches. For example, EGit has up to 24,847,725 patch pairs when replaying its 7,050 patches. We excluded the patches that are in the same code review request. Among the patch pairs, we only counted those patch pairs with a similarity higher than each threshold. As shown in the figure, the number of recommendations decreases as the threshold increases. The number of recommendations sharply decreases between the thresholds of 0.25 and 0.50. Please note that lower thresholds lead to more recommendations with more false positives while higher thresholds lead to fewer recommendations with fewer false positives. We discuss the accuracy of the recommendations based on different thresholds later in this section.

Figure 5.6 presents the average number of recommendations per patch based on different thresholds. Similar to Figure 5.5, the x-axis shows the threshold from 0 to 1 while the y-axis presents the average number of recommendations

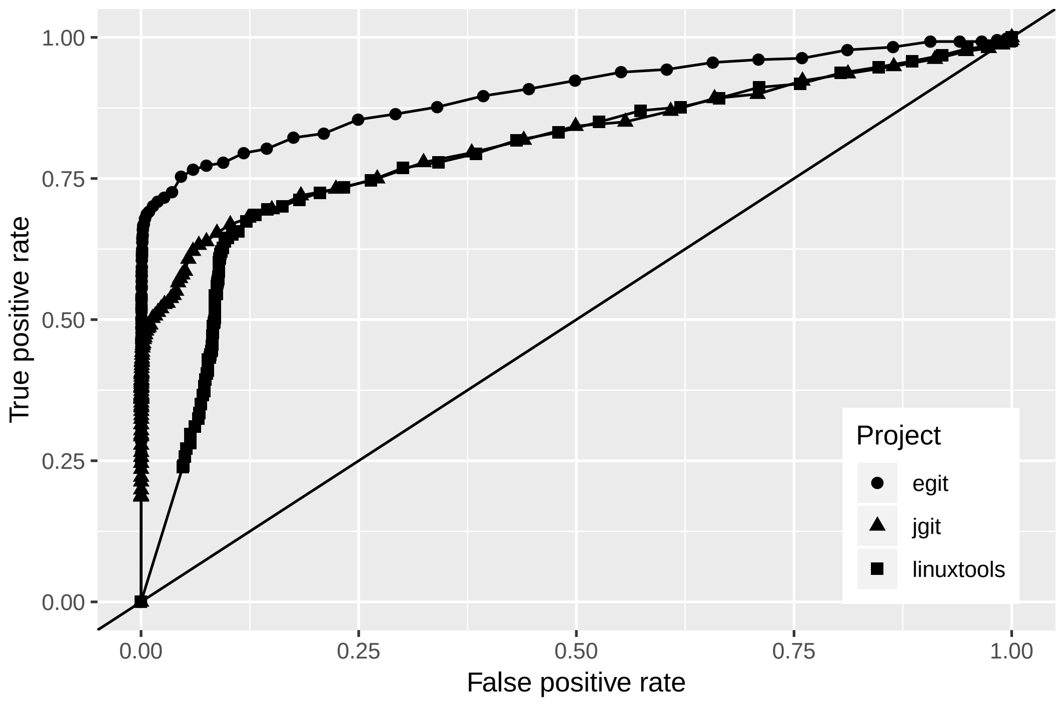


Figure 5.7: Area under the receiver operating characteristics (ROC) curve

that are made per patch on a logarithmic scale. As the threshold increases, the number of recommendations per patch decreases. At a threshold of 0.5, EGit and Linux Tools show less than one recommendation per patch on average, while JGit shows 10 recommendations.

In addition to the number of recommendations, we used the area under the receiver operating characteristic curve (AUC) to measure the sensitivity of our approach over different thresholds [Menzies et al., 2007]. Since AUC is robust with respect to class imbalance and independent of the prediction threshold, it is widely used [Nam et al., 2018, Giger et al., 2012, Lessmann et al., 2008, Rahman et al., 2012, Tantithamthavorn et al., 2017]. By using the partial ground-truth that we described in Section 5.4.3, we compute the AUC of the three projects. The curves in Figure 5.7 are receiver operating characteristic (ROC) curves for each project. Higher AUCs represent the better prediction models, while an AUC of 0.5 (i.e. the straight line without dots in the figure) equals a random prediction model. All three projects

Table 5.9: Six thresholds based on distances from the third quartile for each project with Jaccard index. The distances are computed by using the interquartile range (IQR).

	EGit	JGit	Linux Tools
$Q3 - 0.5 \times IQR$	17%	15%	15%
$Q3 + 0.0 \times IQR$	22%	20%	20%
$Q3 + 0.5 \times IQR$	27%	25%	26%
$Q3 + 1.0 \times IQR$	32%	29%	31%
$Q3 + 1.5 \times IQR$	37%	34%	36%
$Q3 + 2.0 \times IQR$	42%	39%	41%

show high AUC values (i.e. 0.902, 0.820, and 0.791 for EGit, JGit, and Linux Tools, respectively) which means that our technique is robust with respect to different thresholds.

To evaluate the precision and the recall of our approach over the partial ground-truth, we tested six different thresholds based on the similarity distribution. Table 5.9 presents different thresholds for each project based on the similarity distribution. Similarly to Tukey's method, we use different distances from the third quartile (Q3). The thresholds cover the median (i.e. $Q3 - 0.5 \times IQR$) and the third quartile (i.e. $Q3 + 0.0 \times IQR$) as well. As shown in the table, the six different thresholds for each project cover a range from 15% (JGit and Linux Tools) to 42% (EGit).

Figure 5.8 shows the precision and recall for each project with different thresholds over the 100 balanced datasets we constructed as described in Section 5.4.3. The x-axis shows the thresholds (e.g. -0.5 denotes $Q3 - 0.5 \times IQR$) presented in Table 5.9, while the y-axis presents the precision (top) or recall (bottom). Although the plot looks like it has horizontal lines for each threshold due to the narrow range of recall, the horizontal lines actually represent box plots generated over the 100 balanced datasets (i.e. containing quartiles and a median). The narrow range of recall is because only the negatives in the datasets vary while the positives stay the same. As shown in the figure, the precision increases and recall decreases as the threshold increases for all the three projects. For example, EGit shows 0.70 precision and 0.91 recall at a $Q3 - 0.5 \times IQR$ threshold and 1.00 precision and 0.67 recall at a $Q3 + 1.5 \times IQR$ threshold. The precision values show almost the

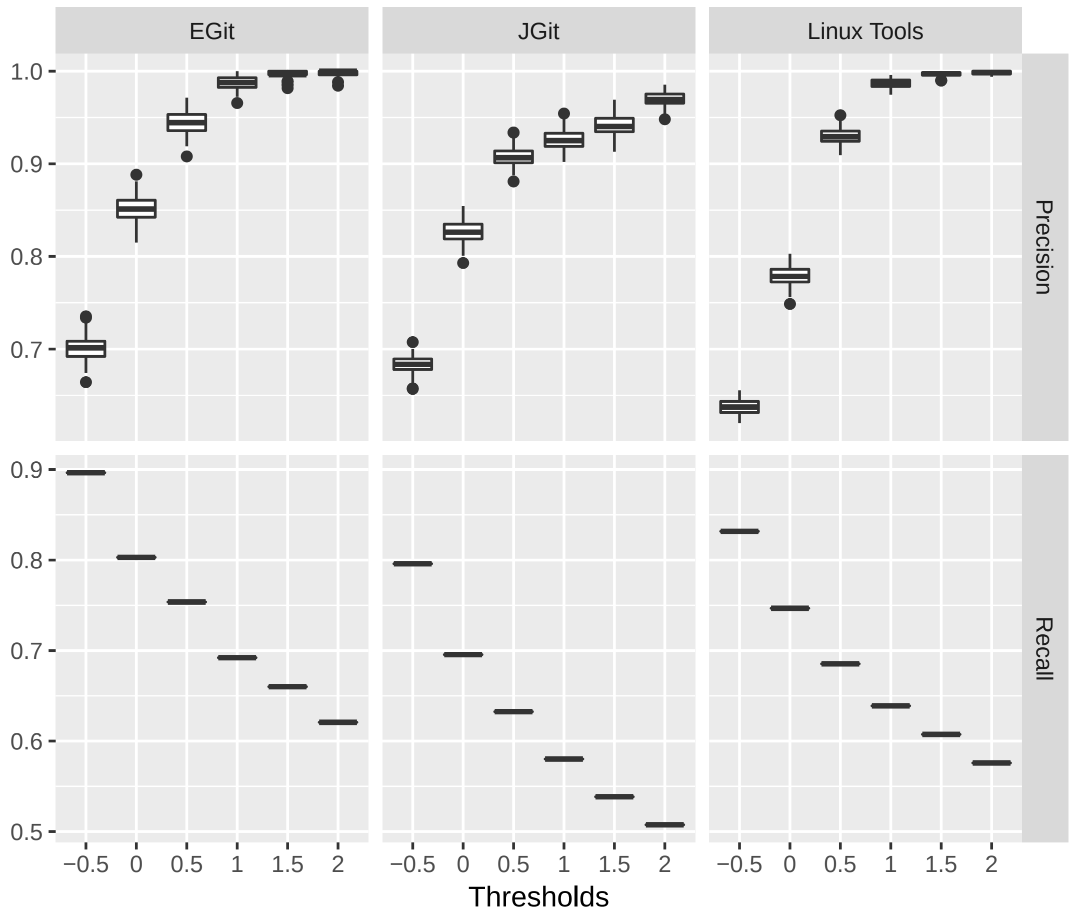


Figure 5.8: Precision (top) and recall (bottom) with different Jaccard index thresholds

same result we saw in our manual investigation (i.e. with the $Q3 + 1.5 \times IQR$ threshold).

Although different thresholds affect the number of recommendations, our technique achieves precision (i.e. precision varies from 0.64 to 1.00) with recall (i.e. recall varies from 0.51 to 0.91) regardless of the thresholds and projects.

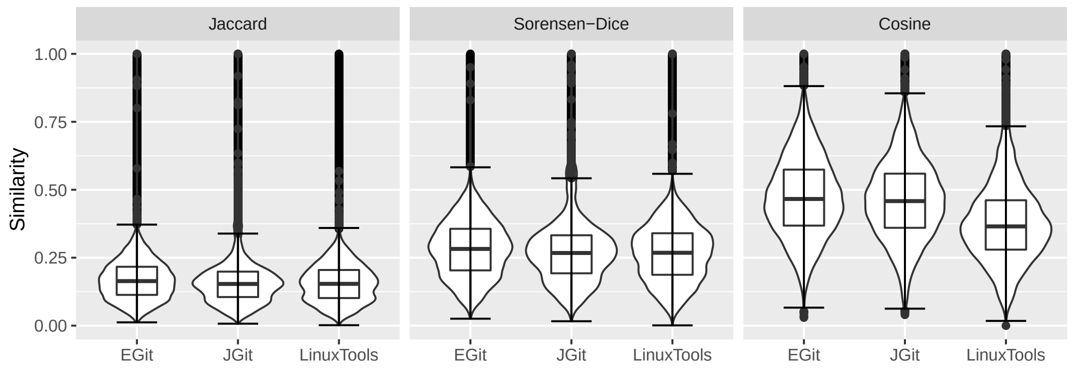


Figure 5.9: Similarity measure distribution for each individual patch

5.5.4 RQ4: How robust is the result compared to other similarity measures with different thresholds?

We used the Jaccard index to answer the previous research questions. However, we are also interested in comparing the results using the Jaccard index to results using other similarity measures to study the sensitivity of the recommendations to the chosen similarity measure. Therefore, we assess two other well-known measures, Sørensen–Dice coefficient and cosine similarity, that satisfy the requirements described in Section 5.4.1.

Figure 5.9 shows the patch similarity distribution using the Jaccard index, cosine similarity, and the Sørensen–Dice coefficient for all three projects. As shown in the figure, the Jaccard index has the narrowest similarity distribution range, while cosine similarity has the widest distribution range. The Jaccard index also has lower similarity values than the Sørensen–Dice coefficient or cosine similarity. The cosine similarity tends to report a much higher similarity than the Sørensen–Dice coefficient and the Jaccard index. One possible explanation is that the vector space model representation of cosine similarity contains not only the words in the documents but also their weights (i.e. the frequency of word occurrences).

Figure 5.10 presents precision and recall for the different similarity measures with a Q3 + 1.5 $\times$ IQR threshold. As in Figure 5.8, we repeated the computation of precision and recall with the 100 balanced datasets described in Section 5.4.3. The x-axis represents similarity measures, and the y-axis

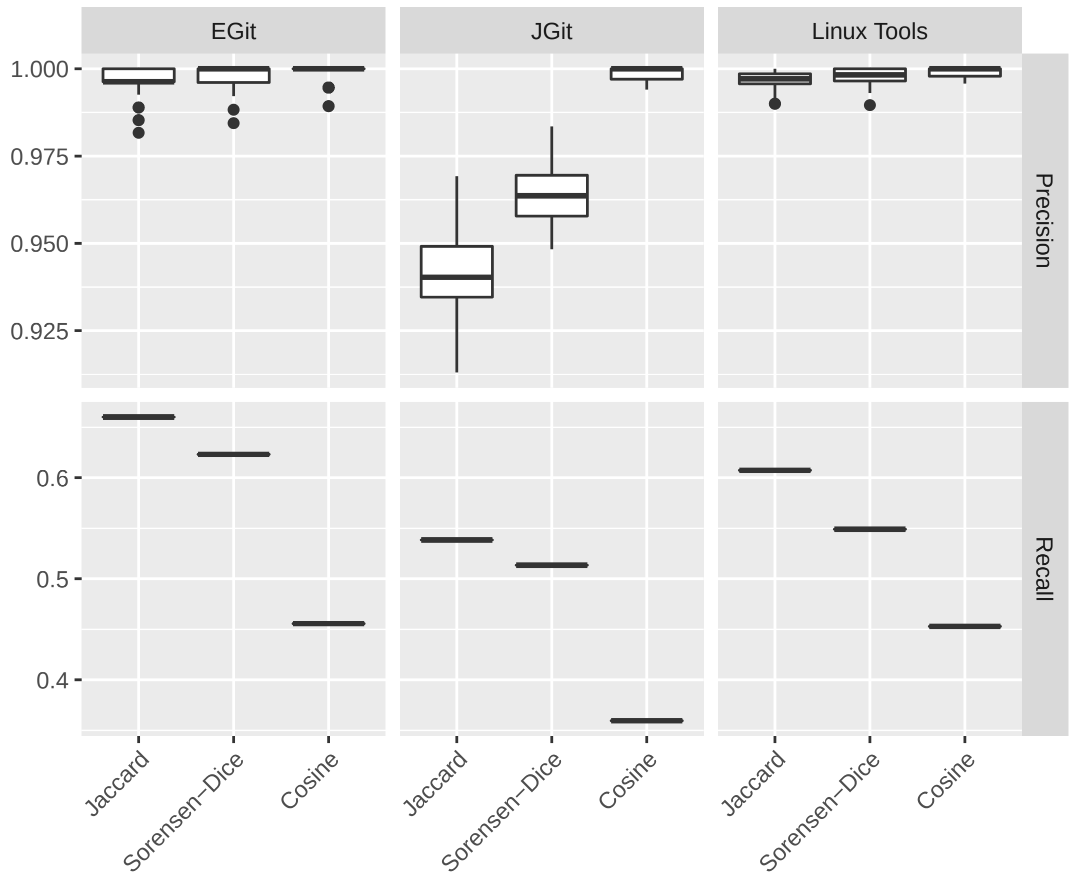


Figure 5.10: Precision and recall with different similarity measures with a Q3 + 1.5 × IQR threshold

shows precision and recall. As shown in the figure, all the three similarity measures show high precision (i.e. precision > 0.92). Cosine similarity shows the highest precision and lowest recall among the three measures whereas the Jaccard index shows lower but comparable precision while showing the highest recall.

Table 5.10: Thresholds for each project and similarity measures

	EGit			JGit			Linux Tools		
	Jaccard	Sørensen–Dice	Cosine	Jaccard	Sørensen–Dice	Cosine	Jaccard	Sørensen–Dice	Cosine
$Q3 - 0.5 \times IQR$	17%	28%	47%	15%	26%	46%	15%	26%	37%
$Q3 + 0.0 \times IQR$	22%	36%	57%	20%	33%	56%	20%	34%	46%
$Q3 + 0.5 \times IQR$	27%	43%	68%	25%	40%	66%	26%	42%	56%
$Q3 + 1.0 \times IQR$	32%	51%	78%	29%	48%	76%	31%	49%	65%
$Q3 + 1.5 \times IQR$	37%	59%	88%	34%	55%	86%	36%	57%	74%
$Q3 + 2.0 \times IQR$	42%	66%	99%	39%	62%	96%	41%	65%	83%

Table 5.11: Average relative precision for different thresholds and similarity measures

	EGit			JGit			Linux Tools		
	Jaccard	Sørensen–Dice	Cosine	Jaccard	Sørensen–Dice	Cosine	Jaccard	Sørensen–Dice	Cosine
$Q3 - 0.5 \times IQR$	0.701	0.700	0.978	0.683	0.671	0.928	0.637	0.603	0.832
$Q3 + 0.0 \times IQR$	0.851	0.844	0.998	0.826	0.813	0.968	0.779	0.762	0.975
$Q3 + 0.5 \times IQR$	0.943	0.946	0.999	0.907	0.905	0.996	0.930	0.927	0.998
$Q3 + 1.0 \times IQR$	0.987	0.994	0.999	0.926	0.932	0.999	0.986	0.990	0.999
$Q3 + 1.5 \times IQR$	0.997	0.998	0.999	0.942	0.963	0.999	0.997	0.998	0.999
$Q3 + 2.0 \times IQR$	0.998	0.999	0.999	0.970	0.990	0.999	0.998	0.999	0.999

Table 5.12: Average relative recall for different thresholds and similarity measures

	EGit			JGit			Linux Tools		
	Jaccard	Sørensen–Dice	Cosine	Jaccard	Sørensen–Dice	Cosine	Jaccard	Sørensen–Dice	Cosine
$Q3 - 0.5 \times IQR$	0.909	0.909	0.916	0.818	0.819	0.831	0.865	0.866	0.920
$Q3 + 0.0 \times IQR$	0.823	0.823	0.860	0.720	0.724	0.734	0.791	0.783	0.874
$Q3 + 0.5 \times IQR$	0.766	0.756	0.771	0.639	0.639	0.639	0.725	0.718	0.801
$Q3 + 1.0 \times IQR$	0.702	0.687	0.663	0.586	0.567	0.546	0.680	0.669	0.735
$Q3 + 1.5 \times IQR$	0.667	0.626	0.571	0.544	0.520	0.458	0.647	0.630	0.668
$Q3 + 2.0 \times IQR$	0.626	0.579	0.453	0.514	0.480	0.373	0.625	0.592	0.597

Similar to Table 5.9, we selected six thresholds to compare precision and recall across three similarity measures with different thresholds. Table 5.10 shows the six different thresholds based on different distances from the third quartile (i.e. Q3). The Sørensen–Dice coefficient and cosine similarity have higher thresholds than the Jaccard index because of their wider similarity value distribution, as shown in Figure 5.3. In particular, EGit with cosine similarity has a 99% threshold at the $Q3 + 2.0 \times IQR$ threshold.

By using the thresholds defined in Table 5.10, we computed precision and recall for the three similarity measures over the 100 balanced datasets. Table 5.11 and Table 5.12 show the average relative precision and recall. As the threshold increases, the precision increases while the recall decreases. With the highest threshold (i.e. $Q3 + 2.0 \times IQR$), all three similarity measures show almost 1.0 precision (i.e. 0.970–0.999).

Surprisingly, and contrary to the evaluation in Table 5.1, cosine similarity with lower thresholds outperforms the Jaccard index: Consine similarity with a threshold of $Q3 + 0.5 \times IQR$ has higher precision and recall than the Jaccard index with a $Q3 + 1.5 \times IQR$ threshold.

Overall, while the Jaccard index shows a good performance, cosine similarity can achieve higher recall and precision at lower thresholds.

5.6 Discussion

Our technique can support developers by recommending a previous code review request, including all discussions between developers about similar changes. Assume that a developer just submitted a new code review request, and our technique has recommended a related previous review. A reviewer can now navigate the discussion between developers in the recommended code review request and can observe how the patches evolved to address the comments of the previous reviewers. Moreover, our approach can be integrated into an IDE (integrated development environment) so that developers can get recommendations and investigate related code review requests before even they submitting their patch for code review.

Our manual investigation of 675 recommended related patches revealed a very high precision in the recommendations among the studied samples from the three projects, as we did not find any false positive for JGit and Linux Tools, and only 2% false positive rate for EGit. These results show that our technique can be used effectively for recommending related code review requests to ease code review. In addition, our interviews with Amazon developers showed promising results for our recommendation technique.

In our manual investigation of recommended related code review requests, we saw large amounts of evidence where reviewers explicitly referring to related code review requests, which suggests that this information is not only helpful but often necessary. A recommendation based on just considering the metadata, such as bug ids or Change-Ids would be simpler, but it would not have a sufficient recall rate. Table 5.6 shows that only 29.9% (EGit), 17.2% (JGit), and 3.6% (Linux Tools) of the sample set are related by bug id, suggesting that the recall rates would be in that range. Similarly, the table shows that only up to 45.6% (EGit), 70.2% (JGit), and 35.5% (Linux Tools) are related by Change-Id. Moreover, remember that such links are created manually by reviewers, an activity that would be supported by our approach.

We also found a significant number of situations in which the same patch had been resubmitted unchanged under a new code review request. We are unaware of any previous work that has addressed the issue of resubmission of identical patches, and it is therefore unclear how previous research results are affected by the phenomenon of resubmitted identical patches.

Although we used the upper outlier fence as a threshold to conduct developer interviews and manual investigation, actual practitioners can set a threshold based on their preferences (e.g. either more recommendations or more accurate results). We also investigated the relationship between recommendation results and the selected threshold. Our approach showed promising AUC values for three projects (i.e. 0.902, 0.820, and 0.791 for EGit, JGit, and Linux-Tools, respectively). In addition, we set six thresholds to compare precision and recall. The six similarity thresholds ranged from 15% to 42%. As the threshold increases, precision also increases while recall decreases. With six different thresholds, the precision varied from 0.6 to 1.0, and the recall varied from 0.5 to 0.9.

The recall rates may seem low when high thresholds are used, but this can be explained by how patches can be related. Our approach recommends related code review requests because their patches made similar changes to the code. However, code review requests also can be related for other reasons. For example, code review request #5363 in JGit was about a patch to fix a problem with large files. A reviewer of this request decided that there was an underlying problem and submitted a new code review request #5366 a few hours later that reverts a previous change which introduced the underlying problem. Request #5363 was abandoned, mentioning request #5366 that made the change in #5363 obsolete. The request pair was, therefore, part of the partial ground truth, although the two requests performed different changes. Their patch similarity (via Jaccard index) is 40%, well below the set threshold of 74%. The median Jaccard index similarity in the JGit partial ground truth is 41%, leading to the expectation that less than half of the related pairs in the partial ground truth are performing similar changes.

To evaluate our technique, we mainly used the Jaccard index, but other similarity measures that satisfy our requirements may show different results. Therefore, we compared the recommendation results of three similarity measures (i.e. Jaccard index, Sørensen–Dice coefficient, and cosine similarity) at various thresholds. We found that cosine similarity at lower thresholds can outperform Jaccard index in precision and recall, contrary to results from a framework and benchmark for source code similarity.

Through the diverse perspective validations, we find that our approach can support reviewers by automatically suggesting related code review requests with high precision and recall rather than having developers manually track related code review requests.

5.7 Threats to Validity

In this section, we discuss the threats to validity of our work as follows:

Internal validity. The interview results do not represent the entire Amazon company. Since Amazon has diverse teams across different domains, the team we interviewed might not be representative. However, the team has the typical structure of the teams at Amazon (i.e. a team should be small enough to be fed by two pizzas [Atlas, 2009]) and follows most of the recommended practices at Amazon.

We evaluated our recommendation technique with three well-known similarity measures, including Jaccard index, cosine similarity, and the Sørensen–Dice coefficient. Leveraging other similarity measures may achieve better results. Moreover, we have not performed any pre-processing, such as TF-IDF [Salton and McGill, 1986] to achieve computational efficiency. As we reported earlier in this chapter, however, our approach showed high precision and high recall with a simple and lightweight (i.e. $O(n)$ complexity) similarity measure, Jaccard index over 3-grams. The results for RQ4 confirm that cosine similarity with lower thresholds can achieve even higher precision and recall.

We used an n-gram size of 3, as it was chosen by the creator of the tools. Choosing other n-gram values may provide different results. We have, however, evaluated the performance of the tools based on 3-gram and found that they gave satisfying results.

The manual investigation results may be subjective, since the investigation was carried out by one of the collaborators in this research and was reviewed by the author of this thesis. To mitigate this threat, we initially looked for clear evidence, such as meta-information that is manually labelled by developers, in the code review data and categorised the relationships from a high-level perspective.

Despite cosine similarity having higher precision and recall at lower thresholds than the Jaccard index that was used in the manual investigation, the results of the investigation are not invalidated simply because the use of cosine similarity would have led to an even higher precision.

Construct validity. The criteria we used to select recommendation examples for the interview affects the validity of our results. As we limited the number of changed lines (i.e. more than 10 lines but less than 50) to ease the interview process, it narrowed the size of our samples. We also limited the number of comments (i.e. requiring more than two comments in a code review request) even though developers may find interesting information in code changes in a recommended code review request without previous comments. Moreover, the 2-month delay might not be appropriate, since developers could still remember the context of the code review requests even after 2 months.

Since we used only a partial ground truth to evaluate our technique together with randomly selected code review request pairs, the reported precision and recall are not necessarily the correct ones. However, the precision results in the evaluation over the partial ground truth are consistent with the results of our manual investigation. To mitigate the threats from the random sampling, we repeated the same experiment with 100 randomly sampled data sets.

External validity. We used Gerrit review data from the EGit, JGit, and Linux Tools projects in the Eclipse Foundation and the internal code review data from a project at Amazon. Although we investigated a large amount of code review data, our findings may not generalise to other projects using different review systems.

5.8 Conclusions

In this chapter, we presented a related code review request recommendation technique. The code review request recommendation technique leverages similarity measures to find related code review requests based on the similarity of their patches. If a new code review request is submitted, our technique computes the similarity between the patch in the new code review request and the patches of previously submitted code review requests. If a patch achieves a similarity value higher than the predefined threshold, its code review request is included in the list of recommendations.

Through the developer interviews at Amazon, we could confirm that the proposed technique has the potential to support developers. We provided three recommendation examples to the developers and let them review the patch. All the interviewees could extract information for reviewing a newly submitted code review request from the recommended code review requests. We plan to extend our work by deploying the technique to the daily code review practice at Amazon.

We evaluated our technique by using three large projects (EGit, JGit, and Linux Tools) from the Eclipse Foundation. We manually investigated a sample of 675 recommendation results to confirm the precision of our recommendation approach and to study the relationship patterns of related code review requests. Our approach achieved 100% precision in the recommendation results for the JGit and Linux Tools projects, and 98% precision for the EGit project.

Since our technique uses a similarity threshold to reduce unrelated reviews, we also investigated the sensitivity of our technique over diverse thresholds in terms of precision and recall. We set six thresholds and measured precision and recall for each threshold. As the threshold for the Jaccard index similarity measure increases, the precision increases from 0.64 to 1.00, and the recall decreases from 0.91 to 0.51.

We investigated our technique with three different similarity measures: Jaccard index, cosine similarity, and the Sørensen–Dice coefficient. We also measured the precision and recall at six different thresholds. We found that cosine similarity at lower thresholds can outperform Jaccard index in precision and recall, suggesting that future work should use cosine similarity despite its poor performance in a framework and benchmark for source code similarity.

6 Conclusions and Future work

Code review is a widely used process to maintain software quality in both open source and proprietary projects. Developers check potential improvements or their colleagues' mistakes during code review. In addition, various automated techniques help developers to conduct code review more efficiently.

In this thesis, we mainly focused on two topics: (1) understanding how developers conduct code review and (2) automated techniques to support developers in code review. We investigated the current state-of-the-art code review studies and code review tools via a literature review. The large-scale developer survey provided a better understanding of developers' perceptions and reality, and of potential improvements in code review. The empirical study on coding conventions during code review highlighted that developers still manually detect convention violations and spend a significant amount of time on this process, despite the fact that various automated convention checking tools are available. Based on the survey, we found that developers have difficulty in locating related patches when reviewing a new patch. Therefore, we proposed the related review recommendation technique based on the textual similarity between patches.

6.1 Summary of Achievements

6.1.1 Developer Survey on Code Review

We surveyed 100 open source developers (i.e. Eclipse and OpenStack), as presented in Chapter 3. We extracted 16 questions within four categories from preliminary interviews with 12 developers. In addition, we reproduced five questions from two previous survey papers and compared our results with those previous results to extend the base of empirical knowledge. Overall, our survey results provide a better understanding of code review practice in open source projects in terms of demographics, reviewer selection methodologies, expected and actual review times, processes for consulting previous reviews during a new review, motivation for code review, and review criteria. Interesting findings reported in this thesis include these: 1) developers often consult previous code review requests when reviewing a new request; 2) developers usually get reviews for their changes within a day, as they expected; and 3) the most important motivations for code review are finding defects and improving code quality.

6.1.2 Coding Conventions in Code Review

We investigated how coding convention violations are introduced, addressed, and removed during code review by developers. We discussed the results in Chapter 4. To do this, we analysed 16,442 code review requests from four projects of the Eclipse community for the introduction of convention violations (as detected by Checkstyle). Our result shows that convention violations accumulate as code size increases despite the fact that changes are reviewed. We also manually investigated 1,268 code review requests in which convention violations disappeared to determine whether a convention violation was removed because it had been raised in a review comment. The investigation results highlight that when convention violations are manually identified and fixed by developers, the code review process may be delayed significantly.

6.1.3 Recommendations of Related Reviews

We propose a review recommendation technique to support developers, as presented in Chapter 5. Our approach recommends related reviews by computing similarities between newly submitted patches and previously reviewed patches. We showed the potential applicability of our technique in practice by interviewing Amazon developers. The developers could easily locate useful information for code review and development context from the recommendation. To evaluate our approach, we applied our technique to the three projects from the Eclipse Foundation with the largest code review histories. We manually investigated the precision of our recommendation results and found it to be 98% to 100% precise. Furthermore, we evaluated our technique with six similarity thresholds and three similarity measures over a partial ground truth that was manually labelled by developers. Depending on the similarity measure and threshold, our technique showed a precision from 0.603 to 0.999 with recall from 0.920 to 0.453.

6.2 Summary of Future Work

Although various empirical studies have investigated how developers conduct code review and what their concerns are, more empirical studies should be performed to improve our understanding. Building on a better understanding via empirical studies, more automated techniques can be proposed to support developers to reduce their use of resources and time in the review process. We discuss several potential research topics that may follow this thesis:

Patch Acceptability Prediction Code review mainly relies on developers' manual efforts to review a patch. Since the manual process consumes a great deal of developers' time and resources, it is necessary to minimise developers' efforts on code review. One potential solution is patch acceptability prediction. Based on various code review metrics, we can develop a prediction model. The output of the prediction model should be the probability that a patch will be accepted or rejected. In the case of a rejection, it should automatically notify the developer of a potential issue to be addressed.

Patch Risk Measure It may be difficult to know the level of risk of a patch during code review. It would be useful if we can build an extension to a code review tool to inform developers of a risk measure whenever they submit a patch. As Kim and Ernst [2007] reported, a patch that changes recently defective files is highly likely to be defective. Therefore, we can derive the level of risk of a patch by using defective file information from code change and code review history. If developers are informed of the risk metrics, they will be careful when they write a patch, and reviewers will review a patch more closely.

In addition, we can leverage review comments to derive the risk topics of each file. Developers describe flaws or issues in files in natural language by using the comment feature in code review tools. The mapping information between file and risk can identify what kind of risk can potentially be introduced in a new change.

Code Review Request Curation Current code review tools stack code review requests just in chronological order (i.e. recent requests will be shown at the top of issue list). Developers who access the code review request list may find the most recent requests are not of interest to them. Issue curation will sort code review requests based on developers' preferences to help them locate code review requests of interests.

Bibliography

Frank Ackerman, Lynne S. Buchwald, and Frank H. Lewski. Software Inspections: An Effective Verification process. *IEEE Software*, 6(3):31–36, 1989. doi: 10.1109/52.28121.

A. Atlas. Accidental adoption: The story of Scrum at Amazon.com. In *Agile Conference*, 2009.

Nathaniel Ayewah, William Pugh, J. David Morgenthaler, John Penix, and YuQian Zhou. Evaluating static analysis defect warnings on production software. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE '07)*, 2007.

Alberto Bacchelli and Christian Bird. Expectations, outcomes, and challenges of modern code review. In *35th International Conference on Software Engineering (ICSE)*, 2013.

V. Balachandran. Reducing Human Effort and Improving Quality in Peer Code Reviews Using Automatic Static Analysis and Reviewer Recommendation. In *2013 35th International Conference on Software Engineering (ICSE)*, 2013.

T. Baum, K. Schneider, and A. Bacchelli. On the Optimal Order of Reading Source Code Changes for Review. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2017.

Moritz Beller, Radjino Bholanath, Shane McIntosh, and Andy Zaidman. Analyzing the State of Static Analysis: A Large-Scale Evaluation in Open Source Software. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2016.

A. Bosu and J. C. Carver. Impact of Peer Code Review on Peer Impression Formation: A Survey. In *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, Oct 2013.

A. Bosu, M. Greiler, and C. Bird. Characteristics of Useful Code Reviews: An Empirical Study at Microsoft. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, 2015.

A. Bosu, J. C. Carver, C. Bird, J. Orbeck, and C. Chockley. Process Aspects and Social Dynamics of Contemporary Code Review: Insights from Open Source Development and Industrial Practice at Microsoft. *IEEE Transactions on Software Engineering*, 2017.

Dong-Kyu Chae, Jiwoon Ha, Sang-Wook Kim, BooJoong Kang, and Eul Gyu Im. Software Plagiarism Detection. In *Proceedings of the 22nd ACM international conference on Conference on information & knowledge management (CIKM '13)*, 2013.

Checkstyle. Checkstyle. URL `http://checkstyle.sourceforge.net`.

Couchbase. Couchbase NoSQL database. URL `https://www.couchbase.com/`.

Jacek Czerwonka, Michaela Greiler, and Jack Tilford. Code Reviews Do Not Find Bugs: How the Current Code Review Best Practice Slows Us Down. In *Proceedings of the 37th International Conference on Software Engineering*, 2015.

Eclipse. Eclipse coding conventions. URL `http://wiki.eclipse.org/Coding_Conventions`.

EGit. Contributors' guide for Egit. URL `https://help.eclipse.org/mars/topic/org.eclipse.egit.doc/help/EGit/Contributor_Guide/Contributing-Patches.html`.

Mahmoud O. Elish and Jeff Offutt. The Adherence of Open Source Java Programmers to Standard Coding Practices. In *6th IASTED International Conference on Software Engineering and Applications (SEA)*, 2002.

Hamed Esfahani, Jonas Fietz, Qi Ke, Alexei Kolomiets, Erica Lan, Erik Mavrinac, Wolfram Schulte, Newton Sanches, and Srikanth Kandula. CloudBuild: Microsoft's Distributed and Caching Build Service. In *Proceedings of the 38th International Conference on Software Engineering Companion (ICSE '16)*, 2016.

Michael E. Fagan. Design and Code Inspections to Reduce Errors in Program Development. *IBM Systems Journal*, 1976.

Michael E. Fagan. Advances in Software Inspections. *IEEE Transactions on Software Engineering*, 1986.

D. G. Feitelson, E. Frachtenberg, and K. L. Beck. Development and Deployment at Facebook. *IEEE Internet Computing*, 2013.

Emanuel Giger, Marco D'Ambros, Martin Pinzger, and Harald C. Gall. Method-level Bug Prediction. In *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, 2012.

Google. Google Java Style guide. URL `https://google.github.io/styleguide/javaguide.html`.

Georgios Gousios, Andy Zaidman, Margaret-Anne Storey, and Arie van Deursen. Work Practices and Challenges in Pull-based Development: The Integrator's Perspective. In *Proceedings of the 37th International Conference on Software Engineering*, ICSE '15, 2015.

Hannaneh Hajishirzi, Wen-tau Yih, and Aleksander Kolcz. Adaptive Near-Duplicate Detection via Similarity Learning. In *Proceeding of the 33rd international Conference on Research and development in information retrieval (SIGIR '10)*, 2010.

Christoph Hannebauer, Michael Patalas, Sebastian Stünkelt, and Volker Gruhn. Automatically Recommending Code Reviewers Based on Their Expertise: An Empirical Comparison. In *31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2016.

S Harris. Simian–similarity analyser, version 2.4, 2015. URL `http://www.harukizaemon.com/simian`.

Austin Z. Henley, KIvanç Muçlu, Maria Christakis, Scott D. Fleming, and Christian Bird. CFar: A Tool to Increase Communication, Productivity, and Review Quality in Collaborative Code Reviews. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems (CHI '18)*, 2018.

David Hovemeyer and William Pugh. Finding More Null Pointer Bugs, but Not Too Many. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE '07)*, 2007.

IntelliJ. IntelliJ coding conventions. URL `https://www.jetbrains.com/help/idea/code-style-java.html`.

Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, 2007.

Yujuan Jiang, Bram Adams, Foutse Khomh, and Daniel M. German. Tracing Back the History of Commits in Low-tech Reviewing Environments: A Case Study of the Linux Kernel. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '14, 2014.

T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code. *IEEE Transactions on Software Engineering*, 2002.

Iman Keivanloo, Chanchal K. Roy, and Juergen Rilling. SeByte: Scalable Clone and Similarity Search for Bytecode. *Science of Computer Programming*, 2014.

Mik Kersten and Gail C. Murphy. Mylar: A degree-of-interest model for ides. In *Proceedings of the 4th International Conference on Aspect-oriented Software Development*, 2005.

Sunghun Kim and Michael D. Ernst. Which Warnings Should I Fix First? In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC/FSE '07)*, 2007.

Grzegorz Kondrak. N-gram Similarity and Distance. In *International Symposium on String Processing and Information Retrieval*, 2005.

O. Kononenko, O. Baysal, and M. W. Godfrey. Code Review Quality: How Developers See It. In *IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, 2016.

O. Kononenko, T. Rose, O. Baysal, M. Godfrey, D. Theisen, and B. de Water. Studying Pull Request Merges: A Case Study of Shopify's Active Merchant. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '18, 2018.

Taek Lee, Jaechang Nam, DongGyun Han, Sunghun Kim, and Hoh Peter In. Micro Interaction Metrics for Defect Prediction. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, 2011.

Taek Lee, Jaechang Nam, DongGyun Han, Sunghun Kim, and Hoh Peter In. Developer Micro Interaction Metrics for Software Defect Prediction. *IEEE Transactions on Software Engineering*, 2016.

S. Lessmann, B. Baesens, C. Mues, and S. Pietsch. Benchmarking Classification Models for Software Defect Prediction: A Proposed Framework and Novel Findings. *IEEE Transactions on Software Engineering*, 2008.

Thomas R Lindlof and Bryan C Taylor. *Qualitative Communication Research Methods*. Sage, 2011.

M.V. Mäntylä and C. Lassenius. What Types of Defects Are Really Discovered in Code Reviews? *IEEE Transactions on Software Engineering*, 2009.

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.

T. Menzies, J. Greenwald, and A. Frank. Data Mining Static Code Attributes to Learn Defect Predictors. *IEEE Transactions on Software Engineering*, 2007.

André N. Meyer, Thomas Fritz, Gail C. Murphy, and Thomas Zimmermann. Software developers' perceptions of productivity. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014.

Murtuza Mukadam, Christian Bird, and Peter C. Rigby. Gerrit Software Code Review Data from Android. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, 2013.

J. Nam, W. Fu, S. Kim, T. Menzies, and L. Tan. Heterogeneous Defect Prediction. *IEEE Transactions on Software Engineering*, 2018.

OpenJDK. Java style guidelines (draft, v6). URL `http://cr.openjdk.java.net/~alundblad/styleguide/index-v6.html`.

Ali Ouni, Raula Gaikovina Kula, and Katsuro Inoue. Search-Based Peer Reviewers Recommendation in Modern Code Review. In *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2016.

Matheus Paixao, Jens Krinke, DongGyun Han, and Mark Harman. CROP: Linking Code Reviews to Source Code Changes. In *International Conference on Mining Software Repositories (MSR '18)*, 2018.

S. Panichella, V. Arnaoudova, M. Di Penta, and G. Antoniol. Would Static Analysis Tools Help Developers with Code Reviews? In *IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2015.

PMD. PMD – an extensible cross-language static code analyzer. URL `https://pmd.github.io`.

Chaiyong Ragkhitwetsagul, Jens Krinke, and David Clark. Similarity of Source Code in the Presence of Pervasive Modifications. In *Proceedings of the 16th International Working Conference on Source Code Analysis and Manipulation (SCAM '16)*, 2016.

Chaiyong Ragkhitwetsagul, Jens Krinke, and David Clark. A Comparison of Code Similarity Analysers. *Empirical Software Engineering*, 2018.

Foyzur Rahman, Daryl Posnett, and Premkumar Devanbu. Recalling the "imprecision" of cross-project defect prediction. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, 2012.

M. M. Rahman, C. K. Roy, J. Redl, and J. A. Collins. CORRECT: Code Reviewer Recommendation at GitHub for Vendasta Technologies. In *31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2016.

John Regehr. Static Analysis Fatigue. 2010. URL https://blog.regehr.org/archives/259.

C. Sadowski, J. v. Gogh, C. Jaspan, E. Soderberg, and C. Winter. Tricorder: Building a Program Analysis Ecosystem. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, 2015.

Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, and Alberto Bacchelli. Modern Code Review: A Case Study at Google. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '18, 2018.

Gerard Salton and Michael J McGill. Introduction to Modern Information Retrieval. 1986.

S. Sarkar and C. Parnin. Characterizing and Predicting Mental Fatigue during Programming Tasks. In *2017 IEEE/ACM 2nd International Workshop on Emotion Awareness in Software Engineering (SEmotion)*, 2017.

Andrew I. Schein, Alexandrin Popescul, Lyle H. Ungar, and David M. Penn ock. Methods and Metrics for Cold-start Recommendations. In *Proceedings of the 25th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2002.

J. Shimagaki, Y. Kamei, S. Mcintosh, A. E. Hassan, and N. Ubayashi. A Study of the Quality-Impacting Practices of Modern Code Review at Sony Mobile. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, 2016.

D. Singh, V. R. Sekar, K. T. Stolee, and B. Johnson. Evaluating How Static Analysis Tools Can Reduce Code Review Effort. In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2017.

Harvey Siy and Lawrence Votta. Does The Modern Code Inspection Have Value? In *IEEE International Conference on Software Maintenance (ICSM)*, 2001.

Michael Smit, Barry Gergel, H. James Hoover, and Eleni Stroulia. Code Convention Adherence in Evolving Software. In *International Conference on Software Maintenance (ICSM)*, 2011.

SonarQube. Sonarqube. URL https://wiki.eclipse.org/SonarQube.

Sun Microsystems. Code conventions for the Java programming language. 1999. URL http://www.oracle.com/technetwork/java/codeconvtoc-136057.html.

C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto. An Empirical Comparison of Model Validation Techniques for Defect Prediction Models. *IEEE Transactions on Software Engineering*, 2017.

Yida Tao, Donggyun Han, and Sunghun Kim. Writing Acceptable Patches: An Empirical Study of Open Source Project Patches. In *Proceddings of the 30th International Conference on Software Maintenance and Evolution (ICSME '14)*, 2014.

Charles Teddlie and Fen Yu. Mixed Methods Sampling: A Typology With Examples. *Journal of Mixed Methods Research*, 2007.

Patanamon Thongtanunam, Raula Gaikovina Kula, Ana Erika Camargo Cruz, Norihiro Yoshida, and Hajimu Iida. Improving Code Review Effectiveness Through Reviewer Recommendations. In *Proceedings of the 7th International Workshop on Cooperative and Human Aspects of Software Engineering - CHASE 2014*, 2014.

Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Raula Gaikovina Kula, Norihiro Yoshida, Hajimu Iida, and Ken-ichi Matsumoto. Who

Should Review My Code? A File Location-Based Code-Reviewer Recommendation Approach for Modern Code Review. In *IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2015.

John W. Tukey. *Exploratory Data Analysis*. Addison-Wesely, 1977.

C. Vassallo, S. Panichella, F. Palomba, S. Proksch, A. Zaidman, and H. C. Gall. Context is King: The Developer Perspective on The Usage of Static Analysis Tools. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2018.

Lawrence G. Votta. Does Every Inspection Need a meeting? In *Proceedings of the 1st Symposium on Foundations of Software Engineering (SIGSOFT '93)*, 1993.

Peter Weißgerber, Daniel Neu, and Stephan Diehl. Small Patches Get In! In *Proceedings of the 2008 international workshop on Mining software repositories (MSR '08)*, 2008.

Claes Wohlin, Martin Höst, and Kennet Henningsson. *Empirical Research Methods in Software Engineering*. Springer Berlin Heidelberg, 2003.

Xin Xia, David Lo, Xinyu Wang, and Xiaohu Yang. Who Should Review This Change? Putting Text and File Location Analyses Together for More Accurate Recommendations. In *In Proceedings of 31st International Conference on Software Maintenance and Evolution*, 2015.

F. Zampetti, S. Scalabrino, R. Oliveto, G. Canfora, and M. Di Penta. How Open Source Projects Use Static Code Analysis Tools in Continuous Integration Pipelines. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, 2017.

Motahareh Bahrami Zanjani, Huzefa Kagdi, and Christian Bird. Automatically Recommending Peer Reviewers in Modern Code Review. *IEEE Transactions on Software Engineering*, 2016.