NAIST-IS-DD1361209

Doctoral Dissertation

Understanding Human Factors and Social Aspects in Modern Code Review

Xin Yang

August 28, 2016

Graduate School of Information Science
Nara Institute of Science and Technology

A Doctoral Dissertation
submitted to Graduate School of Information Science,
Nara Institute of Science and Technology
in partial fulfillment of the requirements for the degree of
Doctor of ENGINEERING

Xin Yang

Thesis Committee:

Professor Hajimu Iida	(Supervisor)
Professor Kenichi Matsumoto	(Co-supervisor)
Associate Professor Kohei Ichikawa	
Associate Professor Norihiro Yoshida	Nagoya University
Professor Daniel M. German	University of Victoria

Understanding Human Factors and Social Aspects in Modern Code Review*

Xin Yang

Abstract

Code reviewing is an important quality assurance mechanism that helps software development teams improve the quality of software products during the development cycle. Due to the distributed collaborations in Open Source Software (OSS) developments, modern code review techniques conducted in OSS projects differ from the traditional code review that based on formal inspection meetings. The effectiveness of code reviews is heavily relied on human involvement and social interactions between developers. However, only a few studies have been performed from the human and social aspect.

In this dissertation, we use multiply research methodologies to research on the human and social aspects of modern code review in OSS projects. There exist three main steps. First, we examine the review processes of OSS projects and retrieve review histories from five large and successful OSS projects. We establish a method to obtain code review data from Gerrit, a widely used code review tool. Second, we investigate review processes from the communications and interactions among code review participants. We analyze the social relationship between patch authors and reviewers by conducting social networks from the data we retrieved. The results indicate that the social network structure of code review participants corresponds with their activeness and expertise. Moreover, it can help practitioners to assign review work and recommend reviewers. Three, since code review process is a social structure, but little is known about how reviewers treat patches and whether their treatments are fair. Hence, we study

*Doctoral Dissertation, Graduate School of Information Science,
Nara Institute of Science and Technology, NAIST-IS-DD1361209, August 28, 2016.

fairness in the context of the code review process. Furthermore, we investigated the behaviors of reviewers that might lead to unfair code review practices. We further perform an online survey on the developers of OpenStack project about the perception of fairness in code review process. Our results shed light on the existence of fairness issue in the code review process, and practitioners should consider review fairness when they design or improve code review processes.

The primary contribution of this work is understanding modern code review from social aspects. We find that: (1) social structure of code review community is essential to evaluate the performance of developers. (2) reviewers' behaviors strongly affect the review process and outcomes. (3) Fairness issues exist in code review process and should not be ignored.

Keywords:

code review, mining software repositories, social aspect, fairness

Contents

List of Tables

List of Figures

Acknowledgement

I would first and foremost like to thank my adviser Professor Hajimu Iida, who gave me many patient guidance, enthusiastic encouragement and precious suggestion for my research work and my life in Japan. He transformed me from an undergraduate student who only had software development experience to a balanced researcher in software engineering field. He also supports me from many other perspectives. I wish to say a heartfelt thanks to him.

I would like to express my great appreciation to Professor Kenichi Matsumoto of Software Engineering Laboratory. I have benefited from his support for my internship, patient guidance and precious suggestions through seminar and lectures.

I would also like to thank Associate Professor Kohei Ichikawa in SDlab, who gives me suggestions in my research and seminar.

I am very grateful to Associate Professor Norihiro Yoshida, my great adviser who supports me from very beginning. Many thanks for his great advice and supports in the past five years.

My deep appreciation are extended to Professor Daniel M. German, my mentor in UVic, Canada. Thanks to his kind advice and patience, I have learned a lot from his guidance. I also want to thank Dr. Germán Poo-Caamaño (UVic, Canada), Associate Professor Gregorio Robles and Professor Jesus M. Gonzalez-Barahona (Universidad Rey Juan Carlos, Spain) for guiding and helping me in many ways, especially in OpenStack research.

I wish to give my special thanks to Assistant Professor Raula Gaikovina Kula, my advisor and friend, who supported me a lot in research and shared many wisdom with me.

I would like to thank Assistant Professor Eunjong Choi. She gave me many help in research and supports in laboratory life.

I would like to thank Associate Professor Toshinori Takai, Assistant Professor Yasuhiro Watashiba for the help. I also wish to acknowledge the help provided by Ms. Yasuyo Morimoto and Ms. Akiko Ogawa in the support of my laboratory life. I wish to express my special appreciation to Dr. Kenji Fujiwara, who gave me great support both on research and daily since I came to NAIST. I would like to appreciate Dr. Kyohei Fushida, Dr. Camargo Cruz Ana Erika, Ms. Yuuki Ro, Ms. Miho Yoshikawa, and all the faculties and secretaries who worked in SDlab before.

My special thanks are also extended to Assistant Professor Akinori Ihara and Assistant Professor Hideaki Hata in Software Enginnering Laboratory. They helped and supported me in seminars, lectures and research. I also thanks to Associate Professor Masao Ohira in Wakayama University, Associate Professor Akito Monden in Okayama University, Lecturer Masateru Tsunoda in Kinki University for the guidance and encouragements even after they left NAIST.

Thanks the staffs in international student affairs section of NAIST for patient support in my research life in NAIST, they are the best friends of international students in NAIST.

I wish to express my great thanks to MEXT and JASSO for supporting my research and life in Japan. I feel deeply honored to get the scholarship from them and this honor strongly encouraged me to work harder.

My special thanks are extended to all the professors, staffs, and students in SDlab and SElab in NAIST, and SElab in Osaka University, many thanks for the intellectual and friendly people around. Chris, Bevis, Enzo, Arno, and many other good friends, I will never forget the good memories (Pokemon Go hunting trip, GoT board game, indie game group) with all of you.

Finally, I would like to express my thanks to my love, my parents, my parents-in-law, and all the family members. My PhD would not have been possible without the love and support of them.

Dedication

To my parents and my love.

Chapter 1.

Introduction

1.1. Overview

Software peer review (or peer review) refers to the code inspections by developers, rather than the authors themselves. It can be regarded as one of the most important activities to guarantee the quality of software products [7,8]. Software projects adopt code review for two principal reasons: reducing defects and saving development cost. The traditional code review (a.k.a code inspection) was established 30 years ago. Code inspection requires experienced reviewers meet and discuss the source code written by other developers [26, 27]. Code review is a process of evaluating a software contribution (e.g., a patch)through manual inspections and peer discussion. Rigby and Bird found that the most significant content of the code review process is performed by human (i.e., reviewers) [60]. Moreover, Code review is not only an indicator of the quality of source code but also signifies a healthy organization. Stable and growing development communities always hope the experienced developers could share their knowledge with new members [60].

Recently, the code review process of Open Source Software (OSS) varies from the traditional industrial setting. One main reason is most OSS projects are geographically distributed, whereas traditional industry projects take the form of gathering developers in the same room. The OSS code review applies a broadcasting method to announce code review tasks and locate appropriate reviewers. Differ with sending mailing list for review candidates, more and more OSS projects adopt modern, light-weight, traceable tools to manage code review process. Modern code review process is quite different with traditional code inspection or mailing list code review. Furthermore, the effectiveness of code reviews is heavily relied on human involvement and social interactions among developers. However, we find that only a few of studies have focused on modern code review especially from social perspective.

In this dissertation, we use a multi-case study methodology to research the human and social aspects of modern code review in OSS projects. Our methodology includes three main steps:

First, we examine the review processes and retrieve review histories from OSS projects that adopt modern code review techniques. We establish a complete method to obtain code review related histories from project source code repositories and code review repositories. We show how the dataset can be utilized to study code review from three aspects: people, process, and product-related aspects of code review.

Second, we investigate review processes from the communications and interactions among code review participants. To understand how code review contributors work and communicate together, we need to investigate the structure of code review community. In this part, we present PeRSoN, which is a construction of social networks from peer review activities, which is based on our previous work [85] [84]. The results indicate that the social network structure of code review participants corresponds with their activeness and expertise. Moreover, it can help practitioners to assign review work and recommend reviewers.

Three, since code review process is a social structure, but little is known about how reviewers treat patches and whether their treatments are fair. Hence, we study fairness in the context of the code review process. Furthermore, we investigated the behaviors of reviewers that might lead to unfair code review practices. We further perform an online survey on the developers of OpenStack project about the perception of fairness in code review process. Our results shed light on the existence of fairness issue in the code review process, and practitioners should consider review fairness when they design or improve code review processes.

The primary contribution of this work is understanding modern code review from human factors and social aspects. We find that:

- we provide a dataset that could help researchers to perform in-depth study regarding modern code review process.

  The dataset includes five successful, big-scale, industry leading OSS projects, and with a total amount of 370,296 patches, 11,058 review contributors, and 5,531,212 review comments.

- We identified the most important contributor roles in modern code review and how to use social network to evaluate developers' performance.

  We found the most active verifiers have significant more centrality measures than all the other contributors group. Moreover, we found strong correlations between activities and centrality measures in most active verifiers group.

- We investigated that unfairness cases exists in a fairness-perceived system such as OpenStack. And we found human is the most important factor to perceive unfairness in code review process.

  From our survey responses and open-coding results, we found that: (1) Most patch authors (85%) perceive fairness, while 15% of them perceive unfairness. Patch authors perceive unfairness maily from: Delay and neglect, Favoritism, Nitpicky, and Newcomers. (2) Most of the reviewers (97%) perceive they are fair when they perform code reviews. Patch reviewers mainly perceive unfairness of themselves from: Who is the patch author, misunderstanding, and Nitpicky. (3) Most of the developers (86%) perceive the entire OpenStack code review process is fair. Developers mainly perceive unfairness from: Newcomers, Core bias, Projects, etc.

1.2. Outline of Dissertation

We organize the dissertation to make sure that each research stage has a independent section for the literature, research questions, methodology and outcomes. The final discussion chapter ties together the three stages and our findings into a summary of modern code review. This thesis is organized as follows.

1.2.1. Review Process and Data Mining Chap. 3

Objective: What are the review processes used by modern code review? How can we retrieve the review data history and source code from these projects?

Literature: We examine the related data mining studies and techniques in code review process.

Methodology: We investigate the basic process of modern code review using Gerrit code review tool from five big-scale and success OSS projects.

Outcome: We present our data in an easy–to–use relational database, thus making it easy for researchers to import into their tools and techniques. We also provide the scripts and procedures to obtain source code revisions in code review histories.

1.2.2. Peer Review Social Network (PeRSoN) Chap. 4

Research question: Which contributor role is the most important in the peer review community (RQa-1), what is the relationship between contributors' activities and their network position (RQa-2).

Literature: We examine the empirical studies of code review process and the social network research from graph theory.

Methodology: We categorize contributors into different role groups based on their authorities, and we define the review activity as any contribution in the review process. We use the PeRSoN to measure the performance of each individual in review networks and statistical analysis to evaluate it.

Outcome: We applied PeRSoN to three large-scale OSS projects: Android Open Source Project (AOSP), Qt and OpenStack by case study. The results of analysis addressed our research questions and gave hints about the relationships among OSS peer review contributor roles, their activities and their network structure.

1.2.3. Review Fairness Chap. 5

Research question: What is the perception of fairness of developers in code review (RQb-1), what is fairness in code review process (RQb-2), how do reviewers prioritize code reviews and how do prioritization strategies reflect review fairness (RQb-3), what does the data tell us about the practical review fairness (RQb-4).

Literature: We examine the empirical studies of code review in human factors and social aspect. To establish fairness theory in code review process, we perform literature survey for the concept of fairness.

Methodology: We categorize contributors into different role groups based on their authorities, and we define the review activity as any contribution in the review process. We use the PeRSoN to measure the performance of each individual in review networks and statistical analysis to evaluate it.

Outcome: we first performed a literature survey to formulate the fairness concept for the code review context. Then, we performed an online survey with OSS developers to better understand how do they prioritize a patch to examine, and how they perceive fairness in code review process. We also applied a queuing system to measure the degree of prioritization and performed quantitative analysis to investigate the impact that prioritization practices have on the reviewing time. Finally, we use statistical analysis to compare the practical fairness and perceived fairness.

1.2.4. Conclusion Chap. 6

The primary contribution of this work is understanding modern code review from human factors and social aspects. *(1) Social structure of code review community is essential to evaluate the performance of developers. (2) Reviewers' behaviors strongly affect the review process and outcomes. (3) Fairness issues exist in code review process and should not be ignored.*

Our findings identify that modern code review methods and techniques changed with the work flow of development life cycle. Findings also present that human factor significantly influences code review process and outcomes such as behaviors. Furthermore, we find interactions among developers such as treatments to patch contribution affect the perceptions of people and it influences the fairness perceptions.

Chapter 2.

Background and Theory

This section presents the background and existing research which related to this study. Section 2.1 introduces the background of code review. Section 2.2 introduces the background and related work about modern code review in OSS. Section 2.3 introduces social network analysis (SNA) and the measures applied in this study.

2.1. Code Review

Code review comes from traditional industrial code inspections [57, 65, 76, 82]. The traditional code review (a.k.a code inspection) was established 30 years ago. Industry projects perform code inspections in a form of formal inspection meetings. Code inspection requires experienced reviewers meet and discuss the source code written by other developers. In the meeting, developers evaluate software contributions (e.g., patches) through manual inspections and peer discussion. An inspection meeting is not as simple, it cost time, money, and human efforts. To prepare an inspection meeting, project/team leaders need to set up a plan, decide the inspection scope, arrange meeting rooms, even consult the available time with every participants. Since developers' schedules are always tight, performing inspection meetings are actually difficult in practices.

2.2. Modern Code Review in OSS

In the past, OSS development has been regarded as unstructured and disordered when compared to industry software development. Eric S. Raymond referred to the different structures and processes of industry software and open source software as Cathedral and Bazaar [59]. With the rise of OSS projects in the past decade, more research has been performed and more supportive tools have been introduced and developed in OSS developments (such as version control systems and bug tracking systems) [64, 68]. The development of these supportive tools have been successfully adopted and developed even by industry software development organizations.

Peer review (code review) in OSS projects can be regarded as an important and necessary component in OSS development process to find defects in software projects [31]. Some study of OSS code review have been done in recent years [70, 74]. Rigby et al. examined Apache Server Project for two techniques and created some metrics similar to traditional inspection experiments in order to find an efficient and effective OSS review technique [62]. Rigby et al. also have studied the broadcast nature of OSS code review, which is total different with traditional method [63]. Balachandran suggested use review bot to reduce human effort and improve review quality [12]. In the past, OSS contributors mainly used e-mail to communicate to send patch review requests and responses about their review results. But in modern OSS projects, code review tools have been put into development regularly [11]. Researchers also study the pull-request based code review in OSS projects [33,80] Based on code review tools, contributors can easily see whether their source code in new patches have been reviewed. In addition, project managers can manage and control the whole review process easily.

Gerrit is a web-based code review system which is mainly used by software projects using Git* as version control system. It can make the review process easier to observe all the changes and also be able to add inline comments. Gerrit system can manage all the patchset (changes) which developers expect to merge into the software code repository. Each patchset should be reviewed by more

*http://git-scm.com/

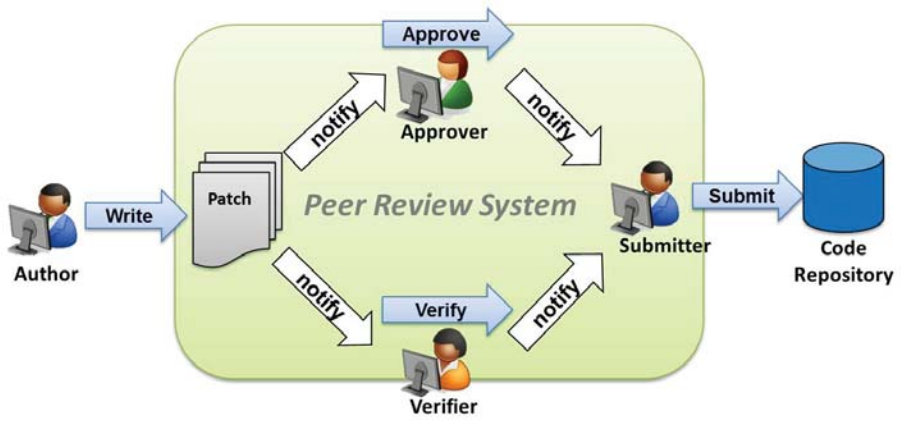


Figure 2.1.: An Example of OSS Review Process

than one experienced reviewers before accepting its integration. Below are some important terms used throughout this study.

- **Contributor.**

  A *contributor* represents a participant who take part in the code review process.

- **Contribution Activity.**

  A *contribution activity* refers to the main activities carried out by contributors. The main contribution activities includes submission new patchset, current patchset update, code review, patchset approval, patchset verification and others.

- **Author, Reviewer and Committer.**

  An *author* represents the contributor who submits a patchset and the owner of review report belonged to this patchset. A *reviewer* represents the contributor who review the patchset to find defect and bug inside. A *committer* represents the contributor who has the authority to commit the patchset into the code repository.

- **Approver and Verifier.**

A *approver* is an experienced reviewer who reviews and approves the patchsets by checking whether the patchset changes follows the best practices that have been established by the project, fits the project's stated purpose, the existing architecture, introduce design flaws, etc. A *verifier* is responsible for building, testing and verifying the patchset and decide whether it is suitable for merging into the source code. In some OSS projects, *verifiers* can be human contributors and automatic tools.

- **Code Review Process.**

  A *code review process* represents the process to perform code review activities by code authors and reviewers in order to guarantee the reliable software. In summary, the following steps are followed: First, authors submit a new patchset and the review system will notify approvers and verifiers. Second, approvers and verifiers will review, build and test this patchset to find whether patchset follows the requirement and it is free for bugs. Finally, the system merges the verified patchset into the code repository and notify the authors whether their patchset pass the review. (see Figure 2.1)

2.3. Social Network Analysis

A social network is a special kind of network structure that vertices (or Actors) represent people and edges represent relationships or interaction between people [55]. Figure 2.2 is a example of developer social network, which vertex present the developers (number represents ID) and edges present they work in same files. Initially social network was applied to research sociology and other sciences.

Social Network Analysis (SNA) is a set of approach and technique which research social network covering sociology, statistics and graph theory. Some researcher in software engineering research distributed OSS development using SNA. Bird et al. extracted and studied the potential structure for latent subcommunities in OSS projects using SNA [16]. In the latest study of SNA research in OSS bug tracking system, Zanetti et al. using SNA to predict bugs into valid

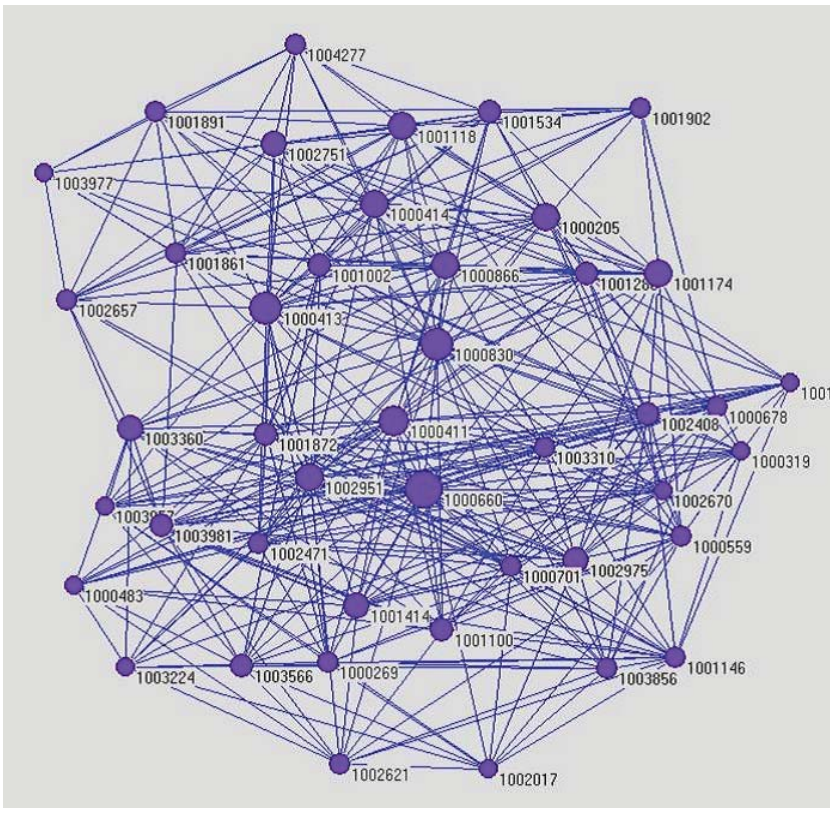


Figure 2.2.: An Example of Social Network

and invalid as bug triage approach of OSS projects [88]. Not only OSS projects, some studies on industry projects also have been done and the analysis results have been confirmed that be able to use for process improvement and failure prediction [51, 56] Other researchers have studied social networks based on social networking media such as Twitter [43]. Different from their previous work which generate social networks mainly from mailing list or bug tracking system, this study generated social networks from code review dataset.

2.4. Related work

Prior work related to this dissertation are listed as below:

We introduce the most important related work with respect to code review studies. Bacchelli and Bird studied the motivations, the expectations, and the outcomes of modern code review [11]. Rigby et al. studied the review policies and examined which metrics have the largest impact on review efficacy in OSS projects [61]. Balachandran suggested using review-bot to reduce human effort and improve code review quality [12]. Bosu et al. investigated the factor of useful reviews to improve the effectiveness of code reviews [19]. Thongtanunam et al. studied traditional code ownership heuristics using code review activities [72]. Baysal et al. found the non-technical factors of code review can significantly influence the code review outcomes [13]. McIntosh et al. found that there exists a negative influence on software quality when the poorly-reviewed code is merged [49]. Jiang et al. found the experiences of the developers impact the patch acceptance and the reviewing time [38]. Tsay et al. found that in some case, even the submitter's contribution is rejected, the core team still fulfill the submitter's technical goals by implementing an alternative solution [77]. Toda et al. Tourani and Adams studied the impact of human discussions, which is related to the interactional fairness [75]. Yang et al. studied the social relationships among the patch authors and the reviewers [85].

We look into the concept of fairness in psychology field and we list the related work with respect to fairness theory research. Leventhal studied the equity is one of the rules of distributive justice, which implies that the rewards and resources should be distributed in accordance with people's contribution [44, 45] Some studies have been done in the research of the procedural justice [69, 79]. Blodgett et al. examined the influences of distributive justice, interactional justice, and procedural justice on complainants' repatronage [17]. Colquitt studied distributive justice, interactional justice, and procedural justice in organizational justice [22]. Sindhav et al. performed survey to test the satisfaction of airport security from the perspective of perceived fairness [67]. Avi-Itzhak et al. studied the measurement of fairness in queuing system [9, 10].

Chapter 3.

Review Process and Data Mining

Many OSS use a modern code review system (e.g., Gerrit, Rietveld) to archive the records of code review activities in their repositories. Many researchers in the MSR field have used these archives for the empirical investigation of code review [40,50,61,71]. Each research group developed their own individual datasets for mining code repositories. However, we need a dataset that can be replicated and used as a benchmark to test related techniques and tools.

As a result, Mukadam et al. [54] and our research group [35] published datasets in the 2013 MSR data showcase. However, compared to our older dataset, the presented dataset is comprised of more projects and has a richer set of content for researchers. Based on the official REST API, our dataset extracts only the key data attributes needed to reconstruct specific aspects of the peer review process.

We present our data in an easy–to–use relational database, thus making it easy for researchers to import into their tools and techniques. Concretely, we show how the dataset can be utilized to study code review from three aspects; (i) people (ii) process and (iii) product-related aspects of code review.

3.1. Peer Review Concepts

We designed our dataset by identifying these three essential aspects that are related to the code review research, as shown in Figure 3.1.

1. `People-related:` refers to social features of software development teams, reviewer roles, and types. Leveraging the socio-technical aspects, we investigate teamwork and collaboration of code members. This can be beneficial

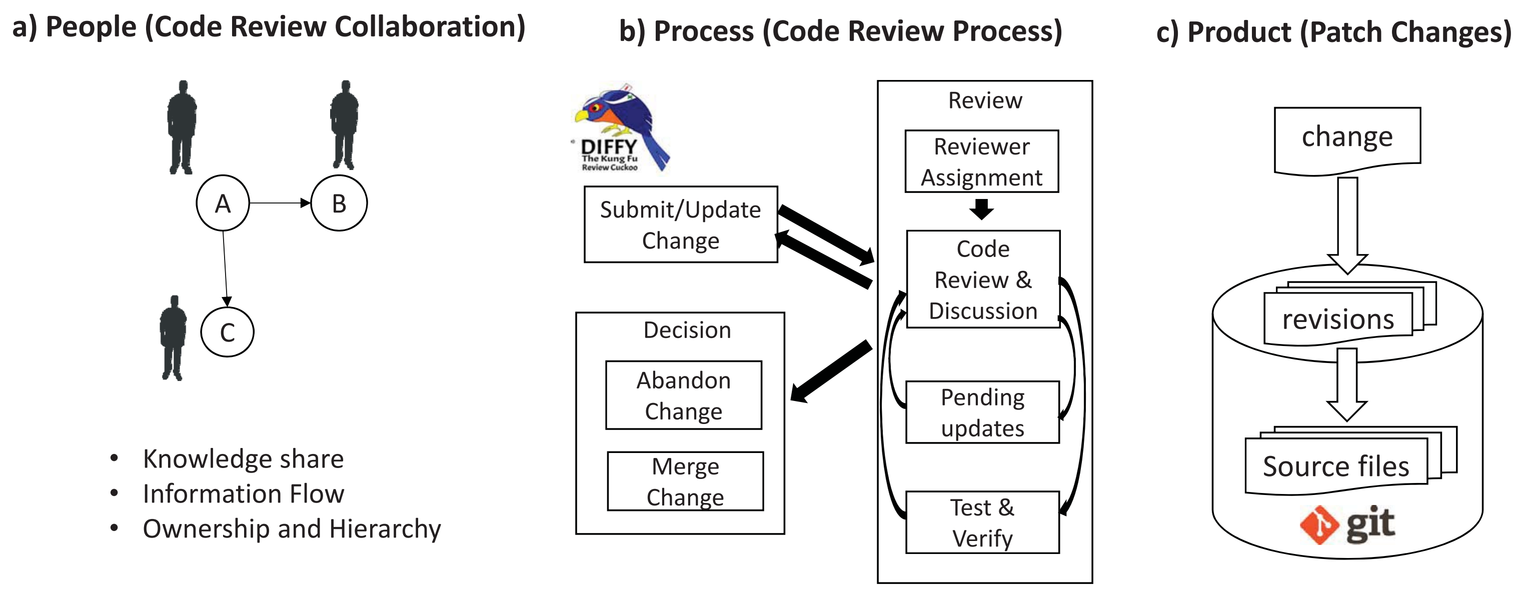


Figure 3.1.: Depicts our proposed aspects of mining a.) People, b.) Process and c.) Product aspects of code review.

Table 3.1.: Dataset Statistics. Projects refers to the number of source code repositories per target.

Project	**Time**	**# Patches**	**# Reviewers**	**# Reviews**	**# Projects**	**DB Size**
OpenStack	2011/07∼2015/04	173,749	5,091	3,961,771	611	1.95 GB
LibreOffice	2012/03∼2015/05	13,597	437	66,618	20	56.5 MB
AOSP	2008/10∼2015/04	63,610	3,334	355,765	567	279 MB
Qt	2011/05∼2015/04	110,172	1,437	1,062,105	111	1.41 GB
Eclipse	2012/02∼2015/05	9,168	759	84,953	189	61.9 MB

to the quality and efficiency of the end product. Typical topics of interest that could be mined are knowledge sharing, collaboration and information flows, code component ownership, and hierarchy within the software team.

2. `Process-related:` refers to review process and review states that are involved in the modern code peer review. Effective and efficient software processes allow for better quality of the code review, which results in a higher quality product. Mining these processes can be utilized to reduce the review time, while making assignments of skilled reviewers to every review.

3. `Product-related:` entails code change, the reviewed code patchset, and associated files. Finally, studying the submitted and merged code patches provides insights into quality aspects, answering such research questions like *'what is the ideal patch size?'* to *'what are critical elements of a successful or unsuccessful patch?'*. Program analysis techniques and code metrics can be utilized to this end.

We now discuss the peer review terms used in this section. **People–People Types**. In a code review, we distinguish the different roles assigned to members of the review community. An `author/submitter` represents the developer who submits a change to Gerrit, and is the owner of this change. A `committer` represents the contributor who has the authority to commit the change to the source code repository. A `reviewer` represents the contributor who performs the code review to any submitted code change. A `verifier` is responsible for building, testing and verifying the changes and decides whether it is suitable for merging. `Verifiers` could be either human or automatic tools (e.g., OpenStack runs testing scripts in Jenkins CI as verifiers). An `approver` is an experienced reviewer who has the authority to approve the changes. An `approver` approves any changes by checking whether the changes fit the best practices established by the project; assessing whether the changes fits the project's stated purpose and the existing architecture. Some projects refer to `approvers` as `core reviewers`. As shown in Figure 3.1 a.), we can use the reviewer types to create a social network, which is useful to analyze social interactions such as knowledge sharing or information flows within the review community.

Process–Code Review States. Every project usually follows a customized workflow, such as the AOSP project*. However, as shown in Figure 3.1 b.), most projects follow these three generic states of code review: `open`, `merged`, and `abandoned` states. An `open` change indicates that a change has not been merged into the source code repository. A `merged` change indicates that the change has already been merged into source code repository, while an `abandoned` change indicates that the change cannot be merged for certain reasons.

As shown in Figure 3.1 b.), the code review states indicates different stages in the code review process. The `open` state can be divided into `new`, `merge conflict` and many other states, specific to a projects workflow. Every change must start from `new` state once the author has submitted it. `merged` and `abandoned` can be regarded as the final decisions of a `open` change. The final decisions of changes usually come from the code review, testing and discussions of core reviewers, which have high authorities in Gerrit system. In addition, projects can tailor their code review states to meet their own needs (e.g., Qt have specialized review states: `Staged`, `Integrating` and `Deferred`).

Product–Code Changes. Shown in Figure 3.1 c.) the code review includes the code changes related to a code review. When a author commits source code, Gerrit will generate a unique `change-id` and create a new `change` in server if not exists. When the author commits a new version of the `change`, it is regarded as a `revision` in Git (it also can be called as a Patch Set in Gerrit). Through Gerrit web server, reviewers can observe the lists of the complete file paths of related `files` in each `revision`, and the summaries of source code changes to files (number of inserted lines and deleted lines). Furthermore, the specific source code changes of files can be observed by showing the *diff* of two different revisions.

3.2. Mining Methodology

Extraction rationale Our dataset is an extraction of the Gerrit repositories through Gerrit official REST API†. Using the REST API, we obtained a raw

* `https://source.android.com/source/life-of-a-patch.html` (Feb 18, 2016)

† `https://gerrit-documentation.storage.googleapis.com/Documentation/2.11.1/rest-api.html` (Feb 18, 2016)

Gerrit dataset from Gerrit servers by sending API requests. The received response will be in the form of a JSON format

However, we identified two reasons why researchers may find it difficult to use the JSON format:

- Complex querying - Querying the JSON format for aspects such as the reviewer types or the process states can quickly become tedious.
- Portability - We would like to represent the data in a format that is easily imported into researchers analysis tools. Thus, we transform the data into a relational database format.

Mining Scripts. We have created a set of mining scripts, which allowed us to mine the dataset easily. Specifically, we choose Python to develop the mining scripts, with MySQL to store the extracted dataset. In Section 3.3, we introduce the detail of review dataset and the database schema.

To obtain the changed code from the pending Git repositories, the following script can be used: `git ls-remote | grep [change-id]`. This will list all the commit-id and path of revisions for a change. The Git command then can be used to obtain certain revision and the diff. Similar useful scripts[‡] were used for other aspects of extraction.

Challenges and Limitations. The main challenge faced when mining the repositories, was the adaptation of the mining script to correctly extract the data from each project. This is because each project has customized the review process. As a result, we had to modify our mining scripts to fit the different API versions of each Gerrit server. For example, we had to change our scripts for the AOSP project, as it adopted the newer version of Gerrit API.

Since reviewer profiles are based on the registration, a possible threat is email aliasing. This is where members may use multiple accounts. We propose for future work to use a semi-manual process of cross-checking the username, name, and email address to remove duplicates. In addition, we currently only identify the file path and size of the patch submitted. For future work, we would like to capture the actual source code changed.

[‡]`https://github.com/saper/gerrit-fetch-all` (Feb 18, 2016)

3.3. Dataset

Mined Repositories.The core of the dataset comes from projects that use GIT as their source code repository, and are also integrated with the GERRIT§ modern code review system. As shown in Table 3.1, the datasets comprise of five large-scale open source projects. All projects are hosted online and are accessible through their respective web interfaces [1–5]. The largest project is OpenStack, a cloud operation system. It has over 3,900,000 reviews and just over 5,000 reviewers. The smallest project collected is LibreOffice, with just over 66,600 reviews and 437 reviewers. We compressed the datasets files to RAR and 7z format for each project. All files are available online for download.

Dataset Schema. We transformed the JSON format into our database schema. Each attribute of the tables can be found in Table 3.2. Our data structure is consistent with the official Gerrit REST API. A full description of the database schema is available¶. We summarize the descriptions of the five tables below:

- **Change** - The change table represents an instance of a code change that is in the review system. The table also contains relevant information such as the author of the code change (`ch_authorId`).
- **Revision** - As a change gets reviewed, it may undergo several revisions of the source code before it is committed. The revision holds information, such as the final commit date of the code change (`rev_committedTime`).
- **People** - The people table was created to store all details of the review members. Each member has a unique id (`p_author Id`).
- **History** - The history table contains all messages or comments related to a review. The history table contains the messages attribute (`hist_message`) that can be used to identify all comments and activities related to the review process.
- **File** - The file table contains the details of the code changes. This table contains information such as the pathname (`f_filename`), and size (`f_linesInserted`, `f_linesDeleted`) of the code change.

§`https://code.google.com/p/gerrit` (Feb 18, 2016)
¶`https://github.com/kin-y/miningReviewRepo/wiki/Database-Schema` (Feb 18, 2016)

Queries example. To utilize the data tables, we need to map the tables to the different aspects of the peer review process. Table 3.3 shows the rationale and hints to which attributes to use when making a query. A sample of other useful queries is available on our website. For example, to get all the core reviewers for a project we need to query the history table (`t_history`) for all people that have 1.) approved or 2.) reject a review or able to provide a score of either 3.) +2 or 4.) -2. This would be in the SQL query:

```
SELECT distinct hist_authorId FROM t_history
WHERE hist_message LIKE '%Looks good to me, approved%'
OR hist_message LIKE '%Code-Review+2%'
OR hist_message LIKE '%Do not submit%'
OR hist_message LIKE '%Code-Review-2%'
ORDER BY hist_createdTime ASC;
```

Table 3.2.: Relational database schema with attributes

Table	Key	**Attribute(Definition)**
`Change` (t_change)	PK	**id**(Unique change id (auto increment))
		ch__Id(Combination of project name, branch name and change id)
		ch__changeId(Change id in Gerrit)
		ch__project(Project name of change)
		ch__branch(Branch name of change)
	FK	**ch__authorId**(Author of change)
		ch__createdTime(Timestamp of when change was created)
		ch__status(Review status of change)
`Revision` (t_revision)	PK	**id**(Unique revision id (auto increment))
		rev__Id(Commit id of revision)
		rev__subject(Subject of revision)
		rev__message(Message of revision)
		rev__authorName(Author of the revision)
		rev__createdTime(Timestamp of when revision was created)
		rev__committerName(Committer of revision)
		rev__committedTime(Timestamp of when revision was committed)
	PK	**rev__patchSetNum**(Revision number in change)
	FK	**rev__changeId**(Change that the revision belongs to)
`People` (t_people)	PK	**id**(Unique people id (auto increment))
		p__authorId(Id of author)
		p__authorName(Name of author)
		p__email(Email address of author)
		p__domain(Domain of email address)
`History` (t_history)	PK	**id**(Unique comment id (auto increment))
		hist__id(Comment id in UUID form)
		hist__message(Comment message)
	FK	**hist__authorId**(Author of comment)
		hist__createdTime(Timestamp of when comment was created)
	FK	**hist__patchSetNum**(Revision number that comment was created for)
	FK	**hist__changeId**(Change that comment was created for)
`File` (t_file)	PK	**id**(Unique file ID (auto increment))
		f__fileName(The path and name of file)
		f__linesInserted(# of inserted lines)
		f__linesDeleted(# of deleted lines)
	FK	**f__revisionId**(Revision that file belongs to)

Table 3.3.: Hints linking review concepts to our database schema

		Linked Tables	**Rationale**
People	id, name, email	People	The people table links to a unique member of the review community
	people roles	History	`History.hist_message` attribute is used to distinguish people types
	commit experience	Change, Revision	The # of `ch_Id` and `rev_Id` linked to a reviewer shows commit experience
	review experience	History, People	Count of `History.hist_changeId` shows review experience
Process	review states	Change	The `Change.ch_status` attribute shows the review states
	review voting	History	Review comments in `History.hist_message` attribute shows the voting results
	review period	Change, History	The difference of commit time (`rev_commit`) and review time (`ch_createdTime`) shows the review period
Product	code changes	Change, Revision, File	How many lines of code has changed in a file, a revision or a change
	revision info	Change, Revision	The revision table links to git commits in code review

Chapter 4.

Peer Review Social Network (PeRSoN)

4.1. Research Questions

We address the following research questions in this chapter.

RQa-1 *Which contributor role is the most important in the code review community?*

RQa-2 *What is the relationship between contributors' activities and their network position?*

Our objective is establishing a model of OSS code review community and a set of quantitative measures to describe the code review process from both technical metrics and non-technical metrics. Another motivation comes from the importance of the human factor in software development. The human factor has been researched from the diversity of different cultures and the rise of globally distributed projects [25].

To understand how contributors work and communicate together, we need to investigate the structure of code review community. In this work, we present Peer Review Social Network (PeRSoN), which is a construction of social networks from code review activities, which is based on our previous work [85] [84]. We categorize contributors into different role groups based on their authorities, and we define the review activity as any contribution in the review process. We use the PeRSoN to evaluate two research questions: **RQa-1.** *Which contributor role*

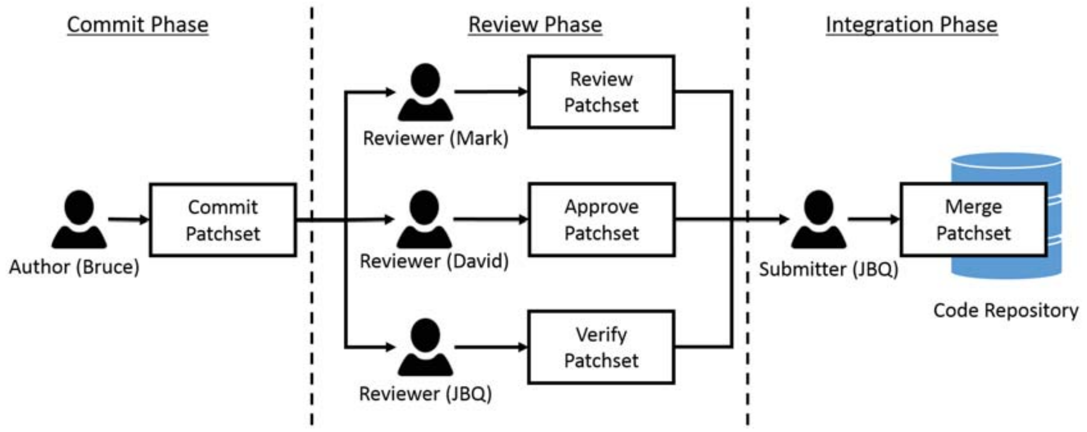


Figure 4.1.: The Review Process of Change #17768 in AOSP

is the most important in the code review community? and **RQa-2.** *What is the relationship between contributors' activities and their network position?*.

We applied PeRSoN to three large-scale OSS projects: Android Open Source Project (AOSP), Qt and OpenStack by case study. The results of analysis addressed our research questions and gave hints about the relationships among OSS code review contributor roles, their activities and their network structure. Our main findings can be summarized as two points. First, the contributors who have the verification authority are the most important (most central) role in the review community (see Table 4.8). Second, a strong linear relationship exists between activities of the contributors who have verification authority and their network positions (see Table 4.9). The main contribution of this work can be summarized as follows:

In the past decade, OSS as a dynamic software development manner has been adopted by many software organizations. Unlike traditional industry projects, OSS code is accessible for patch contributions [28, 53]. With the rise of OSS projects, regular code inspection also has been modified to cater for OSS projects development. It is not feasible for developers to always have communication or collaboration directly. As a result, it is hard to search appropriate reviewers to perform the high-quality review because of lacking the information of contributors. Based on these difficulties, OSS code review applies a broadcasting method to announce and search the appropriate reviewer for particular source code [63].

Table 4.1.: Contributor Roles in the Code Review Process .

Role	Definition
Contributor	A *contributor* represents a participant who takes part in the code review process.
Author	An *author* represents the contributor who submits a patchset and the owner of the review report belonged to this patchse
Reviewer	A *reviewer* represents the contributor who reviews the patchset to find defect and bug inside.
Committer	A *committer* represents the contributor who has the authority to commit the patchset into the code repository.
Approver	A *approver* is an experienced reviewer who reviews and approves the patchsets by checking whether the patchset changes the best practices that have been established by the project fits the project's stated purpose and the existing architecture.
Verifier	A *verifier* handles building, testing and verifying the patchset and decides whether it is suitable for merging into the source code. In many OSS projects, *verifiers* can be automated tools.

To facilitate these OSS code review style, many software projects even industrial projects have adopted code review tools. These companies perform code review instead of face-to-face inspection meeting (e.g., Google uses Gerrit* in AOSP, and Microsoft uses CodeFlow as their review tools†). Many large-scale OSS projects have adopted web-based code review. They also have claimed that code review is an important quality assurance technique in their projects [52] [20] [37]. During the early time of OSS projects, most OSS projects assign review task and retrieve feedback using mailing list [62]. Currently, OSS projects adopt lightweight code review tools instead of using mailing-list. Applying code review tools, the status of code review can be tracked easily and review contributors can participate in code review freely. The mechanism being used in OSS projects for code review provides benefits to communities like sharing knowledge and experience among contributors.

4.2. Peer Review Social Network (PeRSoN)

4.2.1. Process and Networks

Code Review Process.

The *code review process* represents the process to perform code review related activities by code authors and reviewers to guarantee the reliable software. In our study, contribution activity refers to the activities carried out by contributors, which means both authors of code change and reviewers. The primary contribution activities include submission of new patchset, revision of patchset, code review, review approval, review verification, review discussion and others. Based on the activities records in code review system, we calculate the number of activities then separate contributors into different roles by their activities. Also, we introduce the definitions of contributor roles that will be used throughout this study in Table 4.1.

Figure 4.1 is a practical example of review process. This figure represents the review process of Change #17768 of AOSP‡. The first step of review is commit

*`https://code.google.com/p/gerrit/`
†`http://goo.gl/5zk0wF`
‡`https://goo.gl/8uFZiv`

Table 4.2.: Centrality Measures and Social Implication.

Centrality Measures	Social Implication
Degree	Activity
Betweenness	Control
Closeness	Independence

phase that author submit new patchset and the system will notify reviewers by email. Then in review phase, reviewers will perform reviews based on this patchset (Change #17768). Every contributor can perform review in a project, but only those reviewers who have the authority of Approval or Verification(Always chosen from core members, or experienced and active reviewers) can determine whether this change can be merged (Some projects also use bots to build and test the patchset as Verifiers). In this case, three different reviewers have performed reviews. Mark Gross has reviewed but still need someone with approval authority to approve it. As a result, David Turner has approved this patchset, and JBQ (Jean-Baptiste Queru) has verified it. The final step is integration phase, system or particular core members will merge the approved and verified patchset into project code repository. In this example, JBQ has merged this patchset to the repository.

PeRSoN Definition.

Our approach uses social network described as a graph network called PeRSoN. PeRSoN is a social network constructed by code review dataset, which a vertex represents a review contributor and an edge represents a review activity happened between two review contributors (e.g., in Figure 4.1, Mark, David, and JBQ performed code review for Bruce's patchset, and then they left comments as feedback to everyone who have contributed in these reviews). In this study, we define the network model as a undirected and weighted network.

We assume that a contributor c_i has a set of reviews R_{ci}. A review r in a patch set has a set of contributors including both authors and reviewers $\{c_1$, c_2, ... , $c_n\}$. A PeRSoN edge e is formed when two contributors (e.g., c_i and c_j) are members of the same review. Formally, $e(c_i, c_j)$ exists if $c_i \in R_{ci}$ and $c_j \in R_{cj}$ where $R_{ci} \cap R_{cj} \neq \varnothing$.

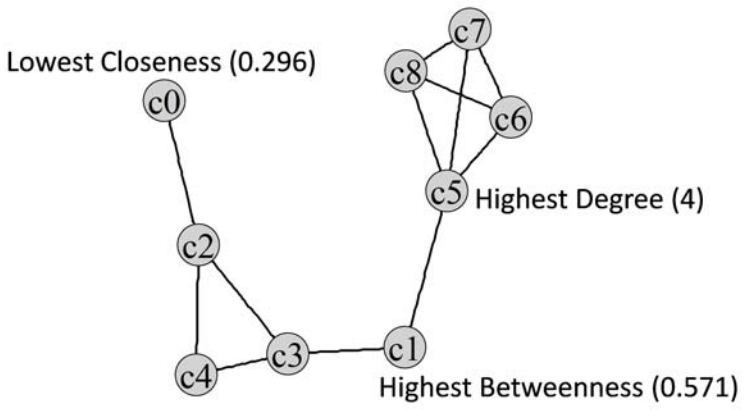


Figure 4.2.: An Example of Contributors Evaluation Using PeRSoN

Network Measures

Our model can be used to describe more complex distribution characteristics of the contributors. In our approach, we evaluate the reviewers network by using the three standard centrality measures of Degree, Betweenness and Closeness based on the definitions from Freeman [29] as below:

- **Degree Centrality.**

  Degree Centrality indicates the number of edges that a vertex has, A vertex (contributor) is defined as c_k, and $a(c_i, c_k) = 1$ if c_i and c_k are connected, otherwise 0. Degree Centrality of c_k is defined as $C_D(c_k)$:

$$C_D(c_k) = \sum_{i=1}^{n} a(c_i, c_k)$$

- **Betweenness Centrality.**

  Betweenness Centrality of a given vertex indicates the number of shortest paths from all vertices to all other vertices that pass through this vertex. We define g_{ij} = the number of edges from vertex c_i to vertex c_j, and $g_{ij}(c_k)$ = the number of edges from vertex c_i to vertex c_j that passing through c_k. Then calculate the probability $b_{ij} = \frac{g_{ij}(c_k)}{g_{ij}}$. Betweenness Centrality of c_k is defined as $C_B(c_k)$:

$$C_B(c_k) = \sum_{i<j}^{n}\sum b_{ij}(c_k)$$

- **Closeness Centrality.**

  Closeness Centrality of one vertex indicates the inverse of its *farness*. *Farness* indicates the sum of the distance between this vertex to all other vertices. We define $d(c_i, c_k)$ = the distance (number of edges) linking c_i and c_k. Then Closeness Centrality of c_k is defined as $C_C(c_k)^{-1}$:

$$C_C(c_k)^{-1} = \sum_{i=1}^{n} d(c_i, c_k)$$

Freeman also suggested that each centrality measures have social implications as shown in Table 4.2 [29]. First, Degree centrality implies activity degree, a vertex with a high degree in the network suggests this person should be active and enthusiastic. Second, betweenness centrality implies control. A vertex with high betweenness centrality acts as a bridge among other vertices in social networks. Third, closeness centrality implies independence of a vertex. A vertex with low closeness represents that this people is independent and far away from all other people.

Figure 4.2 is an example of how we evaluate the performance of contributors from their human factor and social aspect using our approach. Here we simplified the networks by ignoring the weight of edges. However, in the practical experiment, we calculate the weight of edges. The vertices represent contributors and edges represent the review activities between contributors. We perform social network analysis for this network by calculating the centrality measures for each contributor. After we calculated the centrality distribution, some observation can be summarized as follows: **c5** has the highest degree in this network, which indicated he/she is the most active contributors; **c1** has the highest betweenness and it represents that he/she work as a bridge in the community, and he/she could be a potential bottleneck, which may cause problems to the process; **c0** has the lowest closeness which means he/she contribute as individual or rarely collaborate with others.

Table 4.3.: Field and Definition in Comment History.

Field	Definition
reviewId	ID of the review report.
authorId	The author who created the patch.
reviewerId	The contributors who reviewed this patch.
lastUpdate	The timestamp of last update in this review.
writtenOn	the time when this review comment is created.
message	the content of this review comment.

As we introduced in Section 4.2.1, social network analysis provides the data constructed from the social relationship between people. We want to investigate who are the most important contributors and is there a relationship between review authority and social network position in code review community (**RQa-1**). Moreover, whether realistic activities of contributors correlate with their social network position (**RQa-2**). To address these research questions, we introduce the details of our approach in next subsection.

4.2.2. Approach

We now introduce the main steps to our approach as three steps; 1.) Dataset mining, 2.) Network generation and role classification and 3.) Metrics analysis.

Dataset Mining. We used a dataset from our previous work [35][§].

To extract raw code review dataset we apply Gerrit official API to obtain raw dataset. Gerrit code review system provides an REST-like API for users [¶]. Users can access and gain the raw dataset through HTTP for their use. The raw dataset is stored in the form of JSON files. For each review in code review system, the JSON dataset has a unique ChangeID. The JSON dataset includes the following features: The review information with reviewers' comments, the updates history of patches, and the details of when and how they have been merged into the source code or be abandoned. We created scripts using Python to download the raw dataset by using official API. The primary functions of the scripts are

[§] The dataset is available to download at `http://sdlab.naist.jp/reviewmining/`
[¶]`https://goo.gl/PsKwFx`

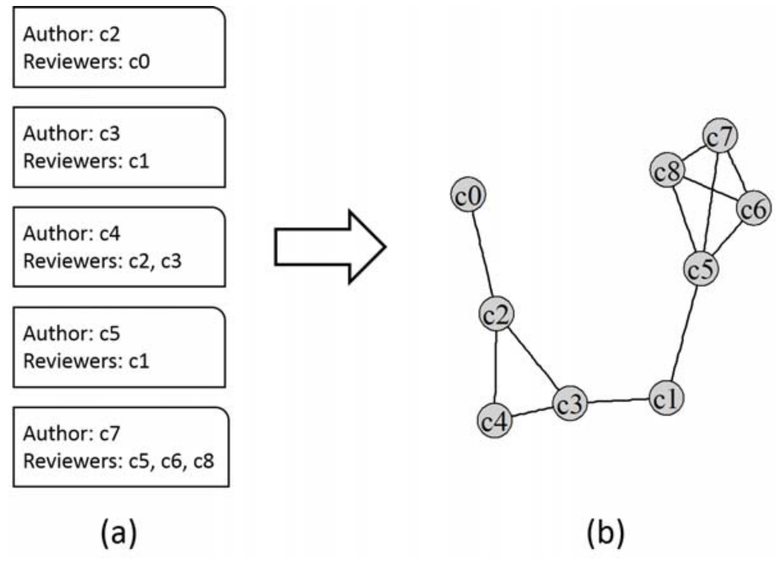


Figure 4.3.: An Illustrative Example of Network Generation

extracting useful data out from raw dataset and store this useful refined dataset into a database. In detail, we extracted the data from the comments history of each review reports in raw dataset. Some important data we used is described as Table 4.3. In next paragraph, we will introduce how we use these data to create the social network and investigate the potential relationship between contributors.

Network Generation and Role Classification. We regard the review as a communication channel between code submitters and reviewers. Every review is a time of discussion or feedback between patch authors and reviewers. Using the history dataset of each review, we extract the connection of all the contributors in code review communities. The steps how we extract the review connection is introduced as follows: First, it was assumed that all the review related activities of the contributors in a review process are recorded in the comments on the review reports, such as an author has submitted a new patch or another reviewer has reviewed a patch and given his/her comments based on the patch change. This step is shown in Figure 4.3(a).

Second, to form the network, all contributors participating in the same review were connected, which indicates the contributors who work in the same review have used the same communication channel as a team work, as shown in Figure

Table 4.4.: Basic Information of AOSP, OpenStack and Qt.

	AOSP	OpenStack	Qt
Period	2009/01∼ 2011/06	2011/07∼ 2012/06	2011/07∼ 2012/06
# of Comments	42449	64793	219044
# of Contributors	1086	426	620
# of Reviewers	451	165	207
# of Approvers	99	86	201
# of Verifiers	111	29	117
# of Vertices	808	379	558
# of Edges	15429	55301	150017

4.3(b).

Finally, we perform role classification by contributors' activities from their comments (e.g., An approval comment from a contributor can be regarded that this contributor has the authority to approve a patch). We separate contributors by their different activities and authorities. More details about classification will be introduced in Section 4.3.3.

Metrics Analysis. The analysis of this study includes analysis of the network metrics. Specifically, we use the standard centrality measures: degree, betweenness and closeness to obtain contributors' network position. Then we compare the centralities between different contributor role groups to investigate which group is most important. We then use statistical analysis to find correlations between the contributors centrality measures and their activities.

4.3. Methodology

We evaluate our approach by applying PeRSoN network to three real world OSS projects that using Gerrit as code review system. These three projects require strict code review mechanism, which means every code change must be reviewed first, then be commit to project code repository. Another reason for choosing these projects is that they adopt code review system instead of a mailing list, which bring more convenience when collecting the data.

4.3.1. Experiment Setup

At first, we studied the code review process using Gerrit environment in three projects. We found that Gerrit system manages contributors by providing different authorities. For example, normal contributors (not core members) can review code, but they have no permission to make the final decision of change. While core members can both review and judge a change. Because high-authority contributors have more responsibility to the quality of change, we first hypothesize the high-authority contributors are more important to the review system as follow.

H1 *The contributors with high authorities and be active in code reviews play the most important role in code review community.*

We address **RQa-1** by accepting **H1**, and we introduce how we compare the difference between high review authorities contributors and other people in statistical analysis in Section 4.3.4. Complementary to **H1**, we add a **H0** as a null hypothesis that indicates *the contributors who have the highest review authorities and other contributors are from the same distribution.*

We address **RQa-2** by calculating the correlation of contributors' activities and their network positions for different contributor groups, which separated by the roles. We use degree, closeness and betweenness to measure contributors' network positions, and then compare their activities.

4.3.2. PeRSoN Generation

Datasets. AOSP (Android Open Source Projects) is an industrial open source project that developing software products for mobile device[||]. OpenStack project provides cloud services platform [**]. Only a few of verifiers are human, but the primary verification works are done by Continuous Integration (CI) tool. Qt Project is an application framework that mainly use to develop graphical user

[||] http://source.android.com/

[**] http://www.openstack.org/

interface [††]. Qt uses the term *Sanity Review* instead of *Verify* in its Gerrit system.

Several necessary data in change information table is needed to generate PeRSoN. e.g., *Change-Id* represents the review report identification, a unique Hash code generated by Git. *Uploaded* and *Updated* represents the timestamps when this report created and when is the latest update.

When a patchset is under review, the author who has submitted this patchset could still update and fix this patchset. A review report could include more than one patchsets if the author upload revisions. In Gerrit, contributors can check the status for current patchset such as who has reviewed or who will verify it. As a result, the comments record all the reviewers who take part in the review are important. In this study, PeRSoN was generated by R [‡‡] and igraph package. The statistical analysis of this study was performed by R.

Generated Networks. Applied the network generating method mentioned in 4.2.2 and in Figure 4.3, the social networks have been generated from these projects separately. The basic information of these social networks is shown as Table 4.4. In this study, the dataset of AOSP covers 2.5 years but OpenStack and Qt have only one year. Because projects have different periods, the review dataset was separated into smaller samples. We split the dataset by one month, three months and six months to observe how the networks of AOSP evolved through time.

In experiment period, we found that results of every six months are most obvious and evident, and then we decided to divide dataset by every six months. We also generate PeRSoN by the whole dataset, and the number of vertices and edges are shown as Table 4.4. We include all the participants in our networks as vertices, and all the review comments between contributors as edges. We can found that each project has a different size in networks.

From the information in Table 4.4, it is easy to find out the common reviewers group is larger than the verifiers group. This observation complies with the Onion Model of OSS development [6, 24] that considers the core members as the most important and the smallest group. It can be supposed that the smallest group of code review, the verifiers should also play the most important roles in code

[††]`http://qt-project.org/`

[‡‡]`http://www.r-project.org/`

Table 4.5.: Summary of Distributions for VAC, VC, AC, and C contributors in AOSP

		VAC	VC	AC	C
Degree	**Min**	5	4	1	1
	1st Qu.	46	15	7	3
	Median	108	29	24	6
	Mean	251	56	25	13
	3rd Qu.	293	49	31	12
	Max	3349	439	79	779
Betweenness	**Min**	0.0000	0.0000	0.0000	0.0000
	1st Qu.	0.0002	0.0000	0.0000	0.0000
	Median	0.0036	0.0000	0.0000	0.0000
	Mean	0.0176	0.0017	0.0006	0.0002
	3rd Qu.	0.0111	0.0024	0.0001	0.0000
	Max	0.6509	0.0155	0.0065	0.0440
Closeness	**Min**	0.3417	0.2837	0.3194	0.0024
	1st Qu.	0.4274	0.3438	0.4089	0.3328
	Median	0.4474	0.4099	0.4199	0.4033
	Mean	0.4455	0.3884	0.4094	0.3708
	3rd Qu.	0.4683	0.4263	0.4226	0.4121
	Max	0.6793	0.4738	0.4473	0.5176

review process. To observe results easily, we separated contributors into different roles from the activities of their review comments that can be regarded as the history of their activities.

4.3.3. Contributor Roles Classifications

We classified the roles of the contributors by the activities they performed during the code review. We extracted contributors' activities by mining their review comments. Administrators or team leaders in a project can set the review rules, such as evaluating code changes by a different score or approving code change with certain review authorities. Each project may have their rules and own authorities assignment. In this study, we investigated the review rules of three projects to extract review activities from the comment history. Moreover, we classify contributors roles more detail to gain a better understanding of the relationship between contributor groups.

As a result, a new classification method using activities are proposed and applied. Based on our previous work related to contributor classifications [85] [42] [73], we apply a classification method based on the activity types. Each contributor is labeled based on their roles, such as V as Verification, A as Approval, C as Code-Review. We investigate contributors from all seven combinations of V, A, C, VA, VC, AC, and VAC. For example, If one contributor has activities records of review and verification, he will be noted as VC. A contributor only contributed by approval without other activities, he will be noted as A. If a contributor has Verification activities also did code review and approved patchsets, who takes part in everything can be noted as VAC.

4.3.4. Results

RQa-1: Most Important Contributors. We address **RQa-1** by comparing the distributions of contributors' network positions. In code review process, verification should be the highest authority in the code review as it was the last stage of change before its merge or abandoned. Moreover, from Freeman's study, we know in a network structure, the centrality measures imply the importance of each node. As a result, the most important contributors indicate they have the highest centralities. In this study, three standard centrality measures have been used, and all of them have different social implications. From the observations of our previous work based on AOSP, which we separated contributor roles into *Verifier* and *Non-Verifier*, we created cumulative graphs of contributors' frequency, and we observed: Verifiers' degree and betweenness increased over time while non-verifiers did not change too much. Verifiers have a relatively greater degree and betweenness than non-verifiers. However, we did not find any relationship between Verifiers and non-verifiers in terms of closeness.

Also, we separate contributors by more detail way as we mentioned in Section 4.1. The classification results like follows: AOSP has four different contributor

Table 4.6.: Summary of Distributions for VAC, AC, and C contributors in OpenStack

		VAC	AC	C
Degree	**Min**	263	9	1
	1st Qu.	995	106	11
	Median	1497	277	29
	Mean	2665	429	69
	3rd Qu.	2604	650	78
	Max	20050	2273	1016
Betweenness	**Min**	0.0000	0.0000	0.0000
	1st Qu.	0.0006	0.0000	0.0000
	Median	0.0033	0.0002	0.0000
	Mean	0.0337	0.0011	0.0000
	3rd Qu.	0.0128	0.0012	0.0000
	Max	0.6290	0.0139	0.0060
Closeness	**Min**	0.5108	0.4725	0.3510
	1st Qu.	0.5602	0.5099	0.4809
	Median	0.5753	0.5232	0.4993
	Mean	0.5966	0.5306	0.4966
	3rd Qu.	0.6082	0.5498	0.5101
	Max	0.8873	0.6087	0.5745

roles as VAC, VC, AC, and C. From the observations we found: VAC group has greater median value than other contributors' groups, and the interquartile range of VAC group is wider than other groups in terms of degree and betweenness but not in closeness (see Table 4.5). OpenStack has three groups: VAC, AC, and C. The results show that the VAC group has greater median than other groups, and same as AOSP, the interquartile range of VAC group is wider than other groups in degree and betweenness but not in closeness (see Table 4.6). Qt has four groups: VAC, AC, A and C. The observation shows, Qt's VAC group has greater median value than other groups in degree and closeness but not obviously in betweenness, and the interquartile range of VAC group is wider than any other

Table 4.7.: Summary of Distributions for VAC, VC, A, and C contributors in Qt

		VAC	VC	AC	C
Degree	**Min**	8	7	2	2
	1st Qu.	363	48	6	6
	Median	776	124	36	20
	Mean	1204	314	45	70
	3rd Qu.	1420	346	82	68
	Max	12010	4194	124	764
Betweenness	**Min**	0.0000	0.0000	0.0000	0.0000
	1st Qu.	0.0000	0.0000	0.0000	0.0000
	Median	0.0000	0.0000	0.0000	0.0000
	Mean	0.0002	0.0000	0.0000	0.0000
	3rd Qu.	0.0000	0.0000	0.0000	0.0000
	Max	0.0053	0.0006	0.0000	0.0001
Closeness	**Min**	0.5014	0.5018	0.5004	0.5009
	1st Qu.	0.5119	0.5053	0.5018	0.5023
	Median	0.5213	0.5082	0.5036	0.5036
	Mean	0.5283	0.5123	0.5039	0.5061
	3rd Qu.	0.5369	0.5134	0.5059	0.5071
	Max	0.6545	0.5919	0.5073	0.5351

group (see Table 4.7).

To compare the different centrality distributions among VAC and other roles, we applied a Wilcoxon-Mann-Whitney test to evaluate **H1** [83]. We adopt Wilcoxon-Mann-Whitney because that we found the population has only one variable (each centrality) but with more than two levels (different roles), and we assume that each role group have independent centrality distribution and not affect each other. All three project have tested by comparing the VAC role with other groups existing in each project. VAC role with VC, AC and C has proved in AOSP. The null hypothesis **H0** that related to **H1** is that VAC and other roles come from the same distribution. **H0** can be rejected, and **H1** can be accepted if VAC > VC, VAC > AC and VAC > C, the p-value of all comparison is below the

Table 4.8.: Comparison of Three Centrality Measures Distributions For Different Role Groups in AOSP, OpenStack and Qt. V Represents Verification, A Represents Approval, C Represents Code-Review, And The Following Number Indicate The Amount Of Reviewers In This Group (e.g., VC (26) represents that 26 contributors have performed Verification and also Code-Review in the current project). Each Row Presents The Hypothesis Being Tested (e.g., VAC ∼ VC), The One-Side Alternative Hypothesis (i.e., >) And The p-Value (With (*) Indicate That The Alternative Hypothesis Is Accepted.)

Projects	Comparison	Degree	Closeness	Betweenness
AOSP	VAC (80) ∼ VC (26)	>, p = 5.091e - 06, (*)	>, p = 8.751e - 07, (*)	>, p = 2.416e - 05, (*)
	VAC (80)∼ AC (16)	>, p = 9.292e - 07, (*)	>, p = 2.897e - 05, (*)	>, p = 2.059e - 05, (*)
	VAC (80)∼ C (451)	>, p < 2.2e - 16, (*)	>, p < 2.2e - 16, (*)	>, p < 2.2e - 16, (*)
OpenStack	VAC (26) ∼ AC (69)	>, p = 9.292e - 07, (*)	>, p = 2.897e - 05, (*)	>, p = 2.059e - 05, (*)
	VAC (26) ∼ C (165)	>, p < 2.2e - 16, (*)	>, p < 2.2e - 16, (*)	>, p < 2.2e - 16, (*)
Qt	VAC (116) ∼ AC (72)	>, p = 1.447e - 14, (*)	>, p = 1.536e - 11, (*)	>, p = 1.096e - 09, (*)
	VAC (116)∼ A (13)	>, p = 1.939e - 08, (*)	>, p = 2.356e - 08, (*)	>, p = 2.645e - 07, (*)
	VAC (116)∼ C (207)	>, p < 2.2e - 16, (*)	>, p < 2.2e - 16, (*)	>, p < 2.2e - 16, (*)

Table 4.9.: Correlation (Spearman) of VAC Activity And Centrality Measure.

Projects	Degree	Betweenness	Closeness
AOSP	0.952	0.789	0.485
OpenStack	0.964	0.992	0.868
Qt	0.953	0.884	0.795

significant threshold of 0.05. The results of p-value are given in Table 4.8 are all below 0.05. The null hypothesis is rejected, and the one-sided alternative hypothesis is accepted, the true location shift is greater than 0. For AOSP, OpenStack and Qt, The most active Verifier (VAC) have significantly higher centrality than other contributors.

As mentioned above, **RQa-1** can be addressed that the most active Verifier (VAC) are the most important (central) role in the review process.

RQa-2: Activities and Network Position. We address **RQa-2** by calculating the correlation of contributors' activities and their network positions. We use Spearman because we take the measurements from ordinal scales, while Pearson correlation is more appropriate for the measurement taken from interval scale. The correlations between the activities of different contributors and their centrality measures in three projects have been calculated. We calculated the correlation for each role group, but we only found a strong relationship in VAC contributors. The results in Table 4.9 shows that in OpenStack and Qt, activities of VAC have a strong linear relationship to all centralities. In AOSP, activities of VAC have a strong linear relationship between degree and betweenness, but not in closeness.

As a result, we addressed **RQa-2** by analyzing contributors' activities and their centrality measures that indicate their network position. We found that most active verifiers (VAC), the relationship between their activities and their network position had a strong positive correlation. Except closeness centrality is unusual in AOSP, all the results show that a strong positive correlation exists between with centralities and contributors' activities. However, we did not found a strong relationship for other contributors' role groups.

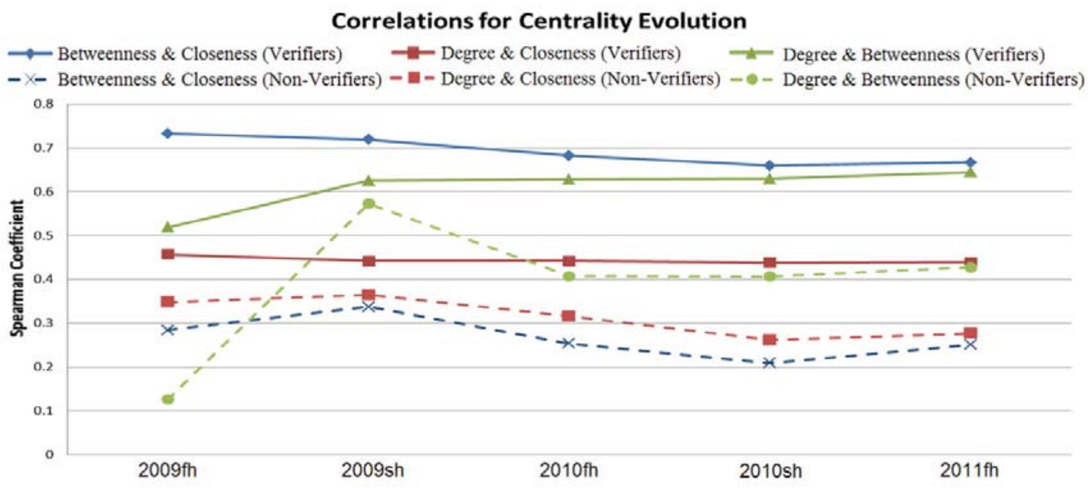


Figure 4.4.: Correlations for Three Centrality Measures Evolution.

4.4. Discussion

4.4.1. Implications

The results of the case study show that the network metrics (centrality) can be used to provide useful information about users roles based on their review activities. In RQa-1, we find that verifiers and not the approvers are the most important roles of a review process. In RQa-2, we find that again contributors that are verifiers (VAC) have a significant correlation with all other network measures.

To further understand the reviewer roles, additionally studied the evolution of role types and network positions over time. As depicted in Figure 4.4, the relationship between the three centrality measures is compared, to estimate correlation coefficients over time. This figure shows that strong positive correlation exists between verifiers' betweenness and closeness, and also between their degree and betweenness. We used this results to categorizing several exception cases of contributors' centralities. Based on the positive correlation between verifiers' betweenness and closeness, it is impossible to find a verifier has high betweenness, but low closeness or a verifier has low betweenness but high closeness. Exception cases have been analyzed in which:

- A verifier had high-degree and low-closeness. A verifier has high-degree means he or she performed more activities than other people, and the low-closeness means he or she is far away from the network center.

- A verifier had low-degree and high-closeness. A verifier has low-degree means he or she performed few activities, and the high-closeness means he or she is close to the network center.

We manually checked dataset to find special cases we described above. We found a core member from AOSP, who has a high degree and a low closeness (comparing with median value). This contributor has contributed a lot in Approvals and Verifications but participated only in *kernel/common* project. While we found another maintainer in AOSP who has low degree but high closeness, we investigated that this contributor contributed very limited times but participated in many projects (e.g., *platform/build*, *platform/external/qemu*, *platform/external/clearsilver*, and *platform/system/core*).

In addition, the correlation results (see Figure 4.4) and the exception cases to create Table 4.10 were combined. From which the following observations can be made: First, verifiers with high-closeness (close to the network center) may have high-betweenness (more control). Second, verifiers with high-degree (more contribution) may have high-betweenness (more control). However, results show that no significant correlation exists between verifiers' degree (contribution activity) and closeness (network position). Finally, from these findings, several suggestions for the exception cases can be provided:

- Verifiers with high-degree and low-closeness may be experts in specialized fields because they perform more activity and have few connections with other members or work teams. Therefore, if they result to be specialists, the important review requests can be suggested to send to them.

- Verifiers with low-degree and high-closeness may contribute less; however, being close to the network center, they may represent key figures tied to many other people. So, verifiers as high authorities can be suggested not to remove them or change their roles thoughtlessly.

Based on the analysis of detecting particular cases, certain contributors can be found in their different behaviors.

Table 4.10.: Exception Cases For Verifiers.

	Low Degree	Low Closeness	Low Betweenness
High Degree	—	Active / Far away	—
High Closeness	Inactive / Central	—	—
High Betweenness	—	—	—

4.4.2. Threats to Validity

This study researched three large-scale OSS projects and proved that the verifiers in code review process are the most important contributors. We discuss the limitations of this study as the following:

- **Dataset Period**. Modern code review process using code review tools is a new technique for OSS projects, different from many bug tracking systems that have been used for more than ten years. In this study, review dataset was extracted from three projects, the longest period (AOSP) are still less than three years. Because AOSP server has shut down for six months, the data structures have been changed after that. As a result, our study based on the dataset has relatively short term period.

- **Community Size and Density**. The community size can be regarded as an important factor in the project's development. Centrality measures are dependent on network size, which is presumably changing over time. The three projects have the different size that may affect the analysis of results. This approach needs to be evaluated by more projects with the different size.

- **Threshold of contributor activities**. One possible threat in our study is the threshold of contributor activities' number. We classify the contributor roles by categories of different activities. However, we did not classify the roles from the number of activities. For example, a VAC contributor who only contribute very few times in Verification but contributed a lot in Code-reviews or Approvals should not be regarded as an active verifier according to his authority and responsibility. In our future study, we need to define a threshold of number of activities according to the case of unbalance activities.

4.5. Related Work

Prior work related to this study could be divided into two aspects: Studies on OSS community and OSS code review; Studies on the social aspect of software engineering. We provide these related work as following two parts.

Raymond referred to the different structures and processes of industry software and OSS as Cathedral and Bazaar [59]. Rigby et al. examined Apache Server Project for two techniques and created several metrics similar to traditional inspection experiments to find an efficient and effective OSS review technique [62]. Rigby et al. also have studied the broadcast nature of OSS code review, which is totally different with traditional method [63]. Balachandran suggested use review-bot to reduce human effort and improve review quality [12].

Bird et al. extracted and studied the potential structure for latent sub-communities in OSS projects using SNA [16]. Zanetti et al. adopted SNA to predict bugs into valid and invalid as the bug triage approach of OSS projects [88]. Kwak et al. have studied social networks based on social networking media such as Twitter [43].

The main difference between our study and related works above is we study code review from social aspect while traditional study are mainly from the technical perspective only. Moreover, based on the prior studies that studied the human and social aspect of software engineering, we found the value and importance to perform study from human and social perspective.

4.6. Conclusion

The motivation of using the approach of Social Network Analysis to research OSS code review process comes from the distributed construction of OSS community and human factors in software development. OSS projects, especially industry-leaded OSS projects need developers to contribute enthusiastically. As proposed human factor should affect OSS review process, SNA approach was applied into this case study has researched three OSS projects. Then reviewers can be classified into several role groups with significant differences. The results show there is a strong correlation between the activities of most important contributors and their network positions. Network measures distributions of contributors can be used for evaluating contributors' activeness. For example, project managements can identify contributors who are enthusiastic but in a specialized field, and contributors who are in important network position but unenthusiastic.

Chapter 5.

Review Fairness

5.1. Research Questions

Since defect resolution cost many resource like time and people, modern code review requires to be performed in a timely manner [30, 36, 62]. Our previous studies about the code review contributors have identified that the roles of contributors (i.e., patch authors, approvers, verifiers) have a lot of overlap [87]. A developers might need to write his or her own patches as a patch author, and also perform code review on other developers' patches as a reviewer. The problem is that if patch authors or reviewers continues focus on their own task, they might have limited time to review other developers' patches (i.e., overload). Other studies about bad smells also reported that the high workloads are more prone to cause bad smells [78]. PostgreSQL project has commitfest managers, who are responsible to make sure that every patch gets review in a timely manner*.

Due to the limited time and workloads, we assume that patches in modern code review may not have the same treatments. In this study, we investigate how developers perceive the treatment of their patches to be equal or not. Moreover, we perform data analysis on the practical patch review histories to identify the difference between developers' perception and the truth. We introduce a concept of fairness to measure the perception of developers. The fairness (or social justice) represent the degree of making judgments that without bias, and it plays an essential role in any social construction that is conducted by human being [44, 45, 48]. Since the code review activities can be regarded as the social interactions

*`http://blog.2ndquadrant.com/managing-a-postgresql-commitfest/`

among the community members (developers), the fairness in software code review should be considered, especially in Open Source Software (OSS) projects. It is a likely case that the OSS developers are discouraged or leave the project if many of their patches do not pass the code review process, which is the final gate to the software repositories of the project. Furthermore, we have asked the OSS developers at the major OSS event of FOSDEM,[†] and two OpenStack Summits about how egalitarian the code review process is perceived.[‡] From the discussion with OSS developers, we find evidences that the OSS developers tend to perceive unfairness in the code reviews. For example, the OSS developers indicated late responses from the reviewers may cause the perceptions of unfairness for them.

Despite the importance of fairness, little is known whether patches are treated fairly in the code review process or not. In the past literature of the code review studies, several studies investigated in human factors and social aspect. For example, Bosu and Carver found that developers' reputation influence their code review outcomes [18]. Yang et al. also found that the social network structure affects the contribution of developers in the code review practices [87].

Thus in this chapter, we address the following research questions:

RQ1 *What is the perception of fairness of developers in code review?*

RQ2 *What is fairness in code review process?*

RQ3 *How do reviewers prioritize code reviews and how do prioritization strategies reflect review fairness?*

RQ4 *What does the data tell us about the practical review fairness?*

In this study, we set out to study the fairness in the code review process. To do so, we first performed a literature survey to formulate the fairness concept for the code review context. Then, we performed an online survey with OSS developers to better understand how do they prioritize a patch to examine, and how they perceive fairness in code review process. We also applied a queuing system to measure the degree of prioritization and performed quantitative analysis

[†]`https://fosdem.org/2016/about/`

[‡]Vancouver (2015):`https://www.openstack.org/summit/vancouver-2015/`
Tokyo (2015):`https://www.openstack.org/summit/tokyo-2015/`

to investigate the impact that prioritization practices have on the reviewing time. Through a case study of the OpenStack project, we address the following research questions:

(RQ1) What is the perception of fairness of developers in code review?

Motivation: Since code review is heavily relied on human involvement and the waiting and service process can be regarded as a social structure, the fairness or social justice in code review process should be investigated. Hence, we asked OpenStack developers about their fairness perceptions in code review practice.

Results: In OpenStack, most developers perceived they perform code review fairly and have been treated well. Most developers perceive the entire code review system is fair, a few of them perceived unfairness.

(RQ2) What is fairness in code review process?

Motivation: Since fairness in code review process should be considered, we have to investigate what review fairness is and why it is important. Hence, we perform literature survey for the concept of fairness.

Results: From the results of the literature survey, we found that the concept of fairness from psychology can be applied to the code review context. Furthermore, we identified that: *Reviewing time*, *Review feedback*, *Communication*, *Review priority*, and *Project policy* can be used to measure the fairness in the code review process.

(RQ3) How do reviewers prioritize code reviews and how do prioritization strategies reflect review fairness?

Motivation: To understand the review fairness in practices, we start our investigation from *Review priority.* Little is known about how reviewers actually prioritize a patch from previous studies. Hence, we asked OpenStack developers about their prioritization strategies.

Results: We found that the OpenStack developers use prioritization strategies to select a patch to examine. In particular, patches are selected based on the related-knowledge of the reviewers, the importance and the difficulty of the patch. Moreover, we found that the OpenStack developers also tend to select the patches from particular patch authors. We found prioritization might lead to potential unfairness perception by some developers.

(RQ4) What does the data tell us about the practical review fairness?

Motivation: In addition to the patch prioritization practices, reviewers should also perform code reviews in a timely manner in a fair review system. In other words, the reviewing time of a patch should not be corresponding to the patch author or the affiliation of the patch author. However, as suggested by the OSS developers, late responses from the reviewers tend to imply a relationship with the unfairness. Furthermore, prior work by [15] found that contributions made by different affiliation can be in unequal practices (i.e., the contributions may not be organizationally distributed). Hence, we investigated whether an author of a patch has an effect on the reviewing time of the reviewers or not. Moreover, we investigated that the perception of fairness from developers, we further discuss that how to measure the fairness in practice.

Results: We found that the patch authors who do not come from the same affiliation as the reviewers are more likely to suffer from a longer review waiting of patch examination than the patch authors who come from the same affiliation as the reviewers. Moreover, we found that a self-approve practice (i.e., patch authors who approve their own patches during the code review) affects the reviewing time. From the statistical analysis, we found that priority of a patch and time-consuming affect the practical fairness, which have similar results as the perceived fairness.

The study is organized as follows: Section 5.2 presents the approaches and the results of three research questions. Section 5.3 discusses the broader implications of our observations. Section 5.4 discloses the threats to the validity. Section 5.5 lists prior related work. Finally, Section 5.6 concludes our works.

The *code review process* represents the process that patch reviewers and patch authors perform code review related activities. The mechanism of MCR provides additional benefits to the development community such as knowledge sharing among the developers and establishing social connections through the code review interactions.

A typical MCR process in OSS projects includes the following steps: (1) the patch author must submit his or her patch to the code review system before merging it to the code repositories (patches with the same goal can be organized as a change with an unique Change-id). (2) reviewers perform reviews

Table 5.1.: Review Score and Definition in OpenStack.

Category	Score	Definition
Code Review	+2	Looks good to me, (approved)
	+1	Looks good to me, but someone else must approve
	0	No score
	-1	I would prefer that you didn't submit this
	-2	Do not submit
Verify	+2	Succeeded (gate test)
	+1	Succeeded (check test)
	-1	Failed (check test)
	-2	Failed (gate test)

on this patch and discuss with the patch author through comments. In most OSS projects, every contributor can perform review but only the core reviewers have the authorities to approve or verify a patch, which determines whether this change can be merged or not. Some projects also apply automated review tools (e.g., Continuous Integration tools) to build and test patches for repeated tasks. (3) the patch author obtain the feedback from the reviewers and tries to revise the original patch. MCR system regards the revised patches as the revisions of the original patch and organize them under the same Change-id. (4) if the core reviewers who participate in this patch reviews consider that this patch has achieved the practices of product quality in this project, they will allow this patch to be merged into the code repositories.

In this section, we examine the code review process in OpenStack. We retrieve part of the dataset from the code review repositories of OpenStack Gerrit. OpenStack is a large, industrial lead OSS project that provides a set of software tools for building and managing cloud computing platforms, for both public and private clouds. OpenStack project started in 2010. Over 200 companies are involved by the time of March 2015 in its development. However, not all the companies have the same authorities to determine the particular development tasks or set the entire business strategies.

OpenStack uses Gerrit to manage the code review tasks.[§] Gerrit is a code review tool that enables developers to perform code reviews and track the status of the reviews through web browser. In addition, OpenStack uses review bots integrated with the CI tools to build and test the patches instead of a human verifier. Gerrit in OpenStack has two kinds of review categories: *Code-Review* and *Verify*. Reviewers vote the patches based on their judgements. Table 5.1 shows the voting score and their definitions in OpenStack. In the *Code-Review* category, the score varies from -2 to +2. (Note that two +1 votes do not make a +2 vote). In the *Code-Review* category, *Approve* means the core members have approved this patch and it can be merged.

In April 2014, OpenStack added a new category *Workflow*, where a patch can be marked as a work-in-progress (WIP) or uncompleted. The purpose of this new category is to avoid unnecessary costs of the code review. There is an additional category *Verify* which is used by Jenkins, the CI tool used in OpenStack. Jenkins runs the "check" tests and returns the results as a +1 or a -1, depending on if the tests are passed or not. If a patch is approved and pass the "check" tests, Jenkins will run the "gate" tests before merging this patch, and return the result as a +2 or a -2. The code is merged only after the "gate" tests are passed successfully.

Note that in this process, every time the patch author updates and resubmits the patch, all the tests and the review tasks must be performed again, except when the updates do not include any change (as it might happen when a patch is *rebased* in Git).

5.2. Case Study

In this section, we present the results of our case study with respect to our four research questions. For each research question, we present our approach and results.

[§]`https://review.openstack.org/`

(RQ1) What is the perception of fairness of developers in code review?

Since code review is heavily relied on human involvement and the process can be regarded as a social structure, the fairness or social justice in code review process should be investigated. Hence, we asked OpenStack developers about their fairness perceptions in code review practice.

Approach

To address RQ1, we conduct an online survey with OpenStack developers. We asked the developers what do they perceive their contributions to be treated. In the survey, we provided three selection questions about how developer perceive fairness in code reviews practice as follows.

- *According to your experience as a contributor, have your contributions been treated unfairly? (never, rarely, occasionally, often, always)*
- *According to your experience as a reviewer, do you perform code reviews unfairly? (never, rarely, occasionally, often, always)*
- *In general, the code review process in OpenStack is fair. (strongly agree, agree, neutral, disagree, strongly disagree)*

For each question, we provided either 5-point unipolar scale or 5-point likert scale. Since there is not clear definition about fairness in code review, we included an opened-ended question after each question to collect the explanation or evidence of the answers of respondents. We provided a free-text box for this open-ended question as below.

- *Feel free to explain or provide evidence for your answer (optional)*

This survey is focused on the OpenStack code reviewers. Therefore, we selected only the OpenStack developers who participated in code reviews in the last 4 years. We acquired a list of developers and their email addresses that are recorded in the code reviews of the OpenStack project (i.e., Gerrit). Finally, we invite the

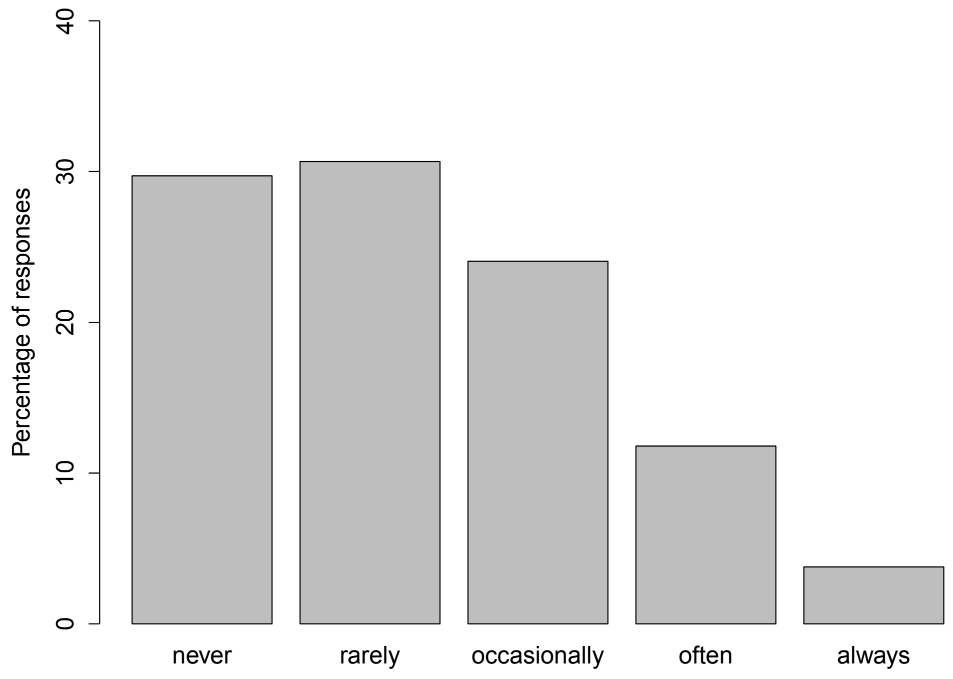


Figure 5.1.: Survey responses of fairness perceptions as patch authors.

2,870 OpenStack developers to participate in the survey by emails and the survey was open for two weeks.

Once we received the responses, we analyzed descriptive statistics on the responses. Furthermore, we analyzed the responses of the open-ended question by using Grounded Theory, which is based on the concept of coding [23]. To do so, we carefully read the responses of the open-ended question and categorized them into several groups. To mitigate the bias of the categorization results, two authors of this study performed the coding separately, then their results were cross-validated.

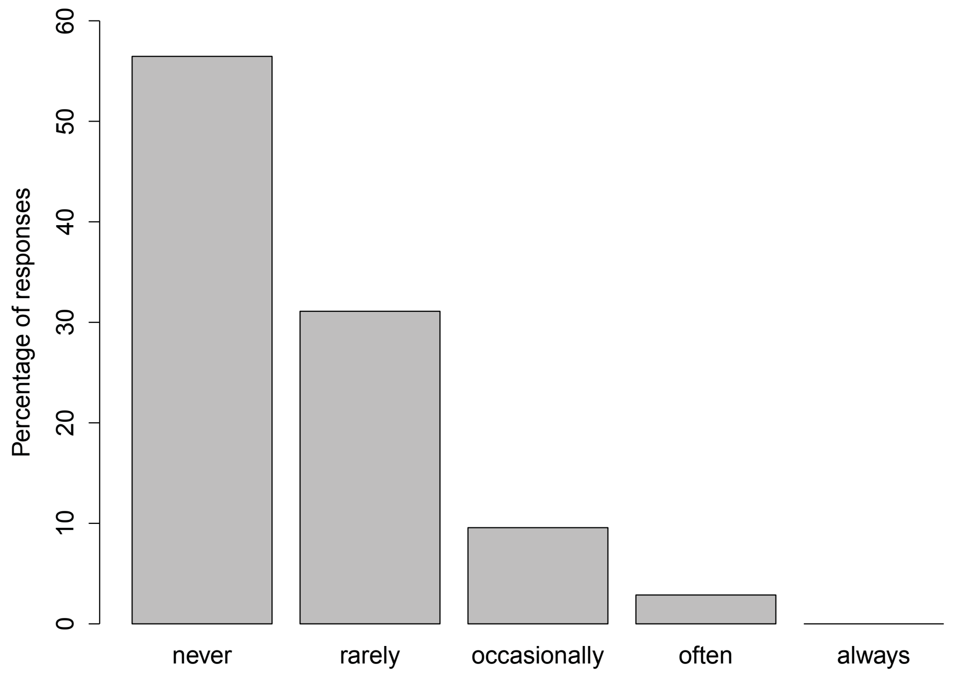


Figure 5.2.: Survey responses of fairness perceptions as patch reviewers.

Results

Perception as Developers:
(have your contributions been treated unfairly?)

We are interested to investigate the “perception of fairness from the perspective of developers”. After the memoing phase, we group the responses of this questions into seven categories. Each category indicate a reason why respondents perceive unfairness from their experience in code review processes.

1. Delay and Neglect

Respondents stated that review delay or neglect for a long time to their contributions can be regarded as a phenomenon of unfair treatment.

It largely depends on the project. [Project A] is the absolute worst. Easy patches for bug fixes can go 9 months without approval.

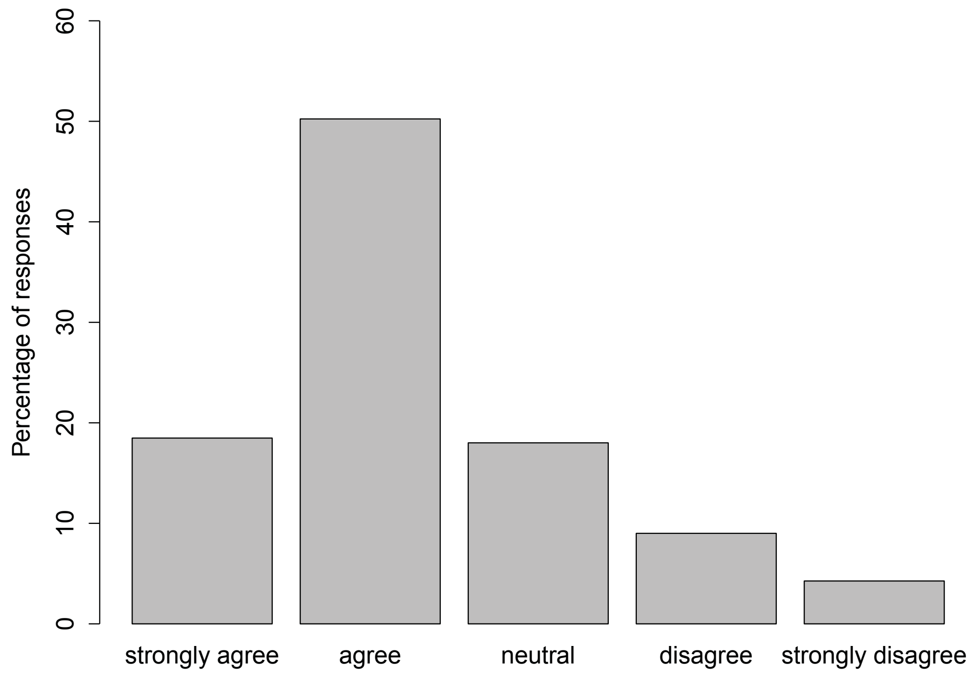


Figure 5.3.: Survey responses of fairness perceptions as patch authors.

Reasons: Developers' mistake (e.g., use Old API) geographical or cultural issue (e.g., patch authors and reviewers from different countries) Contribution with less importance

2. Nitpicky Reviews

Respondents stated that they perceive unfairness from the reviewers who are overly picky to their contributions, especially in detailed parts like spelling, grammar or code style. Some respondents mentioned that they do not regard picky as unfairness but a quite unpleasant behavior.

> *I don't think I'd categorize reviews on my contributions as 'unfair' but rather 'nitpicky'. I think there are many reviewers in OpenStack who will give a negative review not for functionality but rather for a typo or an extra space...*

Influence: It discourages people

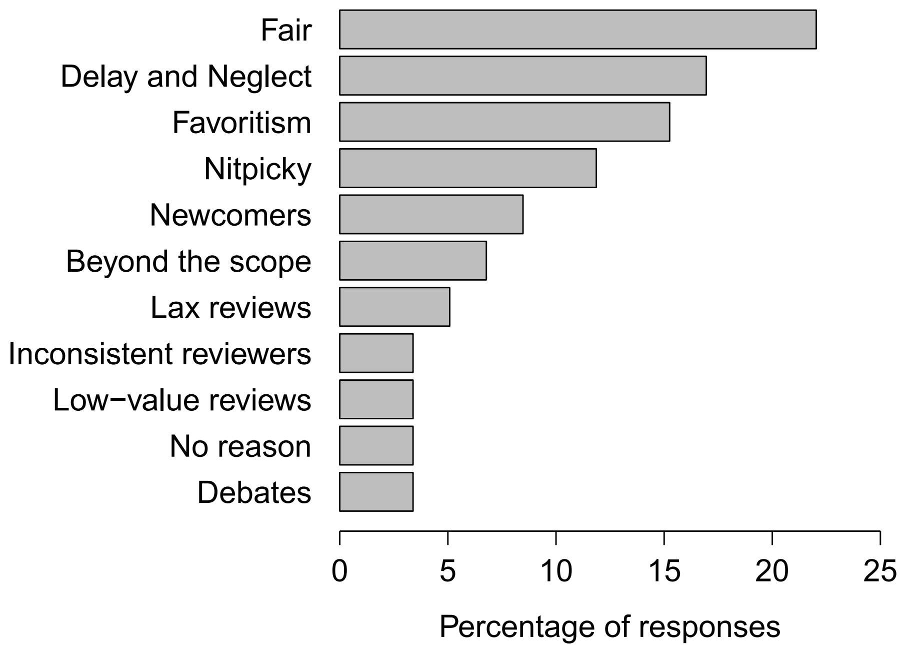


Figure 5.4.: Explanations about fairness perceptions to the entire code review system.

3. Reputation

Respondents mentioned that the reputation of a contributor or his/her affiliation in community can affect his/her perception about review fairness.

It helps to be a member of a corporation. I would imagine the experience of anyone contributing on one's own would be very difficult for many projects.

It usually takes a bit longer to get a reviewer to look at your code when they don't know you. Once you've built some trust, it's easier.

4. Confused Reviews

Respondents stated that sometimes they obtain the confused review feedback,

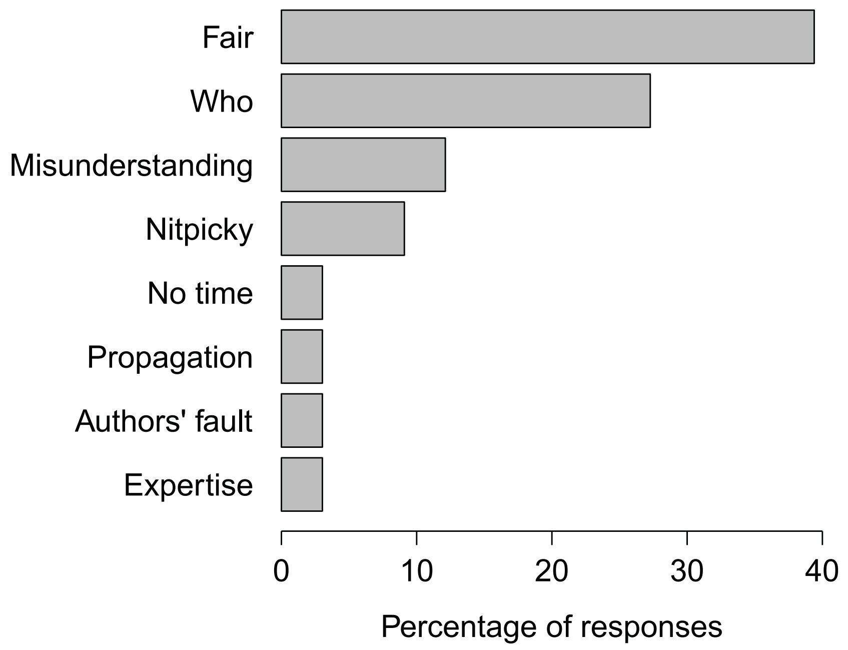


Figure 5.5.: Explanations about fairness perceptions as patch reviewers.

such as beyond the scope of their submitted changes, or conflicts and arguments among reviewers.

Something that usually happens is that you spend a lot of time modifying your patches to fit the current reviewer's view, and after a couple months, a new reviewer takes over and makes you change another bunch of things, I've got patches hanging around for more than a year due to that.

5. Low-value Reviews

low quality feedback

Respondents stated that they perceive unfairness when their contributions receive low-value reviews, or be rejected without good reasons.

Frequent reviews where reviewers clearly hadn't even read the code.

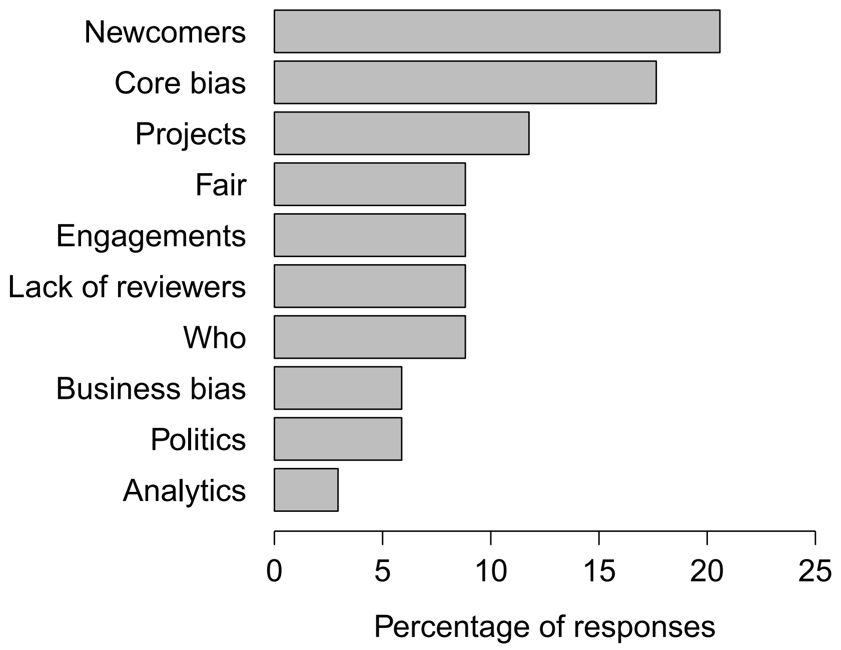


Figure 5.6.: Explanations about fairness perceptions to the entire code review system.

Recently i had 5 +1 and 1 +2 and a core reviewer gave me a -1 which was not understandable. I asked him 4 times on IRC to discuss with me and he ignored it.

6. As a Newcomer

Respondents stated that newcomer may suffer difficulties when contribute, and they may perceive discouragement or bias from the review feedback.

I have only tried to make one contribution. I was so discouraged by the response from the reviewer I have not tried again.

Reason: New contributors know few about the project and community.

7. Prioritization

A few of respondents mentioned that they perceive unfairness from the prioritization strategies of core reviewers.

Leads in [Project A] have indicated that their believe the group of core reviewers focus on reviews they care about primarily (e.g. changes they want in the project) rather than treating the whole community equally and fairly.

Perception as Reviewers:
(have your performed review unfairly?)

We are interested to investigate the "perception of fairness from the perspective of reviewers". After the memoing phase, we group the responses of this questions into five categories. Each category indicate a reason why respondents perceive unfairness from their experience when performing code reviews.

1. Human aspect (WHO)

Respondents stated that human aspect cause a lot of influence when they perform code review. They express that a review can be affected by who the author is and which company the author belongs to.

I try to treat all code review as equal as possible. Often I try to avoid looking at the name of the contributor so that I am not accidentally biased.

2. Delay

Respondents stated that the delay of reviews may cause unfair perceptions.

There are case that people are trying to delay other's contributions.

Reasons: Miss the posting order Not enough reviewers (bandwidth)

3. Nitpicky

Respondents stated that they are more strict to the newcomers. However, they do not treat this behaviors as unfair. The respondents tend to check the small mistakes such as typo or spelling error, which may cause newcomers perceive unfairness. Another respondent expressed that he/she is more picky to his/her colleagues than contributors from other companies.

... I think I do tend to be tougher on a new contributor (or perhaps I'm just more prone to looking very deeply at their code and as a result, catch more spelling mistakes and other small errors)

put into WHO *I often am much more strict when reviewing code from my [Company] colleagues than I am from other companies. It's not that I am being unfair; it's more that I am just extra picky in how I treat the code.*

4. Author's Fault

Respondents stated that they may perform unfairly because of the problem of authors. A respondent expressed he/she will stop to review the patches from particular patch contributors because his/her comments have been ignored by these contributors.

If I comment on a review several times and the committer totally ignores the comments, and does not respond, then I'll stop future review of that commit. Since I'm under no obligation to review code, I'll sometimes be reluctant to select that person's code reviews in the future.

5. Misunderstanding

Respondents stated that the misunderstanding of reviewed code can be regarded as an issue.

Sometimes reviews may be missed or the patch may be misunderstood, causing the patch owner to appear to have been treated poorly

Perception to the entire project

(In general, the code review process in OpenStack is fair?)

We are interested to investigate the "perception to the entire project" from a wider viewpoint. After the memoing phase, we group the responses of this questions into six categories. Each category indicate a reason why respondents perceive fair or unfairness in the entire project.

1. Core contributors (WHO)

Most respondents stated that core (or high reputation) contributors play important roles in the review process. Some of the respondents claimed the core contributors deserve more resource and better treatments in code review. Other respondents expressed that they perceive unfairness from the bias treatments from/to core contributors.

*OpenStack is a *community*. Contributing members in good standing that regularly and unselfishly carry their share of the workload can and should expect to receive higher priority for reviews.*

It's strongly biased towards existing core contributors, whose patches are often reviewed faster. Tiny issues are allowed to hold up patches, and delay in review often translates to missing a feature freeze or release window, and forcing the code to wait for another 6 months to go through the process again.

2. lack of reviewers

Respondents stated that some projects have few core members, which may delay the entire work progress and cause unfairness perception.

As the OpenStack projects are becoming bigger, the issue about the number of core reviewers to review all the changes was raised. This is discussed regularly in the OpenStack mailing list and there is no easy solution. So today, the review process is slowed down by the small number of core reviewers.

3. Politics of reviews

Some respondents stated that the review process is politicized.

Cores review other cores patches, need to get all political to have a decent chance for a review every 2 to 3 weeks.

I believe the process is fair, but politicized. A submitter is presented with the illusion that all submissions are equal, when in reality the community prioritizes PR's from knows submitters on current topics of discussion.

4. Importance of projects and affiliations

Respondents stated that the importance of projects and affiliations cause different results in the code review process.

... If you spend a lot of time on IRC and the mailing list and work on areas considered to be important, then you're more likely to get your patches reviewed quickly...

Though I've not have much issues aside from the delays, I have a lot of colleagues that have been turned down just because the company that had the most members in the specific project's committee did not benefit from that contribution.

5. Statistics-driven reviewers (WHO)

A respondent stated that he/she found some “Statistics-driven” reviewers are unfair. Those reviewers are motivate to generate reviews by the desire of increasing the amount of contributions.

> *I have contributed and reviewed in 3 projects: A, B, and C. I found that A and B to be quite reasonable and fair. As stated above, C is a disaster. But in general I have found that the review process and the metrics gathered by stackalaytics to generally favor negative reviews, especially to contributions by non-cores. Reviewers have an incentive to increase their review count on stackalytics, and it is *much* easier to do that by giving a -1 to someone's change for some trivial or minor stylistic change than to actually +1 it.*

Most developers perceived they perform code review fairly and have been treated fairly. Moreover, most developers perceive the entire code review system is fair, a few of them perceived unfairness.

(RQ2) What is fairness in code review process?

In order to better understand what is the fairness in the code review process, we perform a literature survey of the concept of fairness [46]. In particular, our goals of RQ2 are to investigate (1) What is fairness in other areas (e.g., psychology)?, (2) What is fairness in the context of code review process?, and (3) Can the concept of fairness be applied to the context of the code review process?

Below, we present our approach of literature survey and discussion of applying the results of literature survey to the code review context, followed by a general conclusion.

Literature Survey

We focus on the studies that have been published in the past 10 years (2006-2016). To ensure the coverage of the survey, we select the prior studies that have been published in several areas such as computer engineering, psychology, and management. Therefore, our approach of the literature survey consists of two steps: searching and filtering. We now describe each step of our approach.

Searching: To discover the related works on fairness, we search for publications in Google Scholar using "fairness" as a main keyword. We also use the "code review" and "computer" keywords for the searching. However, there is no publications related to fairness in neither the code reviews nor software engineering studies. We only find several studies address the fairness in terms of resource allocation in the computer system and computer network. Yet, these studies are not related to the fairness in terms of human and social aspect. Therefore, we mainly focus on the literature in psychology studies.

Once we retrieve a set of publications from the discovering step, we expand this set of publications by searching their references. To determine whether the publication is related to the fairness, we check the title and the abstract of each publication. Finally, we identified 17 related publications.

Filtering: In the filtering step, we carefully read these 17 papers. We found that 7 papers are applicable to investigate our proposed fairness issues occurring in code review practices. We found that some work mention fairness but the contents are diverse. For example, some papers apply the concept of fairness to explain the phenomena and issues in the area of law, politics, or economics. Due to our limit knowledge, we only keep the papers that mainly introduce the fundamental concepts of fairness.

Results

From the literature survey, we find that the concept of fairness is derived from the service fairness and organization justice [34]. These studies are focused on how do recipients perceive the degree of fairness in the behaviors of service providers [66] and employees perceive the fairness from employers [34]. Moreover, we can identify three main types of the fairness: (1) *distributive* [17, 22], (2) *procedural* [9, 39, 79], and (3) *interactional* [17] fairness. Table 5.2 lists the basic rules and their examples for each type of fairness, which we describe in detail below.

Distributive Fairness: The distributive fairness refers to the fairness in the outcome of a dispute, a negotiation, or a decision [17, 22]. Distributive fairness is focused on the fair exchange between a contribution and an outcome. There is three applicable rules for distributive fairness, i.e.,(1) Equity, (2) Equality, and

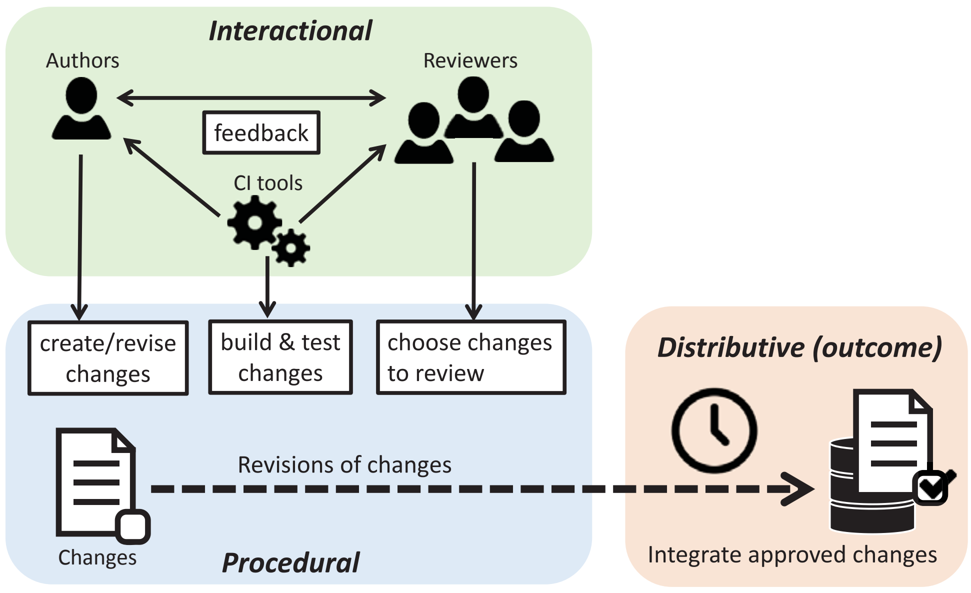


Figure 5.7.: Three Perspectives of Fairness in the Review Process

(3) Need. *Equity* refers to the distribution of rewards and resources that should correspond to one's contribution. In contrast of *equity*, *equality* requests everyone to receive the same outcome regardless the contributions. *Need* refers to whether the outcome meets the requirements of the recipient.

Procedural Fairness: The procedural fairness refers to the fairness in the procedures of deciding the outcome [79]. The rules of procedural fairness include *control (including process-control and decision-control)*, *consistency*, *bias suppression*, *information accuracy*, *correctability*, and *ethicality*. Procedural fairness is related to timeliness, responsiveness, and convenience of the issues handling process. For example, several studies have found that the long-term waiting causes participants' negative emotion, which leads to unfairness perception [9, 39].

Interactional Fairness: The interactional fairness refers to the interpersonal treatments between recipients and decision makers during the process. In particular, the interactional fairness requires a good social manner such as *politeness*, *respect* and *dignity*. Prior studies have shown that the fair interactional practices bring better overall evaluations of complaint handling process in service fairness [17].

Applying Fairness Theory to Code Review Context

From our results of the literature survey, we apply the concept of fairness to the context of the code review process. Thus, we separate a typical code review process into three main phases corresponding to the three categories of fairness theory, i.e., interactional fairness, procedural fairness, and distributive fairness. Figure 5.7 provides an overview of the three perspectives of fairness in the code review process. Below, we discuss how the fairness theory (see Table 5.2) can be applied to the code review process as shown in Table 5.3.

Distributive Review Fairness:
Distributive review fairness refers to the fairness in the outcome of the code review process. The review outcome includes a decision (i.e., whether a patch should be integrated into the software repositories or not) and reviewing time (e.g., when to review a patch and how long). Since the patches in the code review process are diverse, one should not expect that every outcome will be equal. However, since the long-term code reviews can disappoint developers, the review outcomes of patches should close to an mean value or a similar result. Hence, we suggest that the *equity* and *equality* rules should be applied in the code review process. Furthermore, a project should avoid merging the patches with low quality. Thus, the *need* rule should be applied in code reviews.

We propose that the *reviewing time* and *review feedback* can be used to measure the distributive review fairness in code review practices. The *reviewing time* refers to the timestamp of each review feedback , while the *review feedback* presents the reviewers' opinion towards a patch or revisions.

Procedural Review Fairness:
Procedural review fairness refers to the fairness that exists in the procedures of code reviews.

Since the procedures of code review are involved by developers, we suspect that the behaviors of the participants can influence the fairness in the code review process. In the code review process, developers can act as both a patch author and a reviewer, i.e., creating a patch and selecting a patch to review. To yield a fair review system, when developers act as reviewers, they should apply the code review process consistently to every patch without bias. In other words, the *consistency* rule and *bias suppression* rule should be applied in the code review process. Furthermore, developers should able to express their opinion about code review practices and get responses from others (i.e., *control* rule). For example, when a developer is searching for a reviewer for his or her patch in case of emergency, the review system or project should allow the developer to express his or her request.

Since the review procedures are designed by projects, the policy of a project is essential to the fairness of code reviews. On the granularity level of system (project), a fair code review process should: (1) provide accurate information to developers (i.e., *Information accuracy* rule), (2) prepare for correcting the mistakes in review process (i.e., *Correctability* rule), and (3) adhere to ethical and moral principles in practices (i.e., *Ethicality* rule).

We propose that the *review priority* of patches, the *communication* between review participants, and the *project policies* can be considered to measure the procedural review fairness in code review practices. The *review priority* refers to the priority of a patch based on the potential prioritization strategies of reviewers. From mining the information of *communication*, we can identify that whether individuals can control the review process. The *project policy* refers to the policies established by project managements to protect the interests of developers.

Interactional Review Fairness:
Interactional review fairness refers to the fairness that exists in the interpersonal treatments in the code review process. In particular, during code review, reviewers examine a patch and provide feedback to the patch author. The patch author can either revise the patch to address the reviewer feedback or discuss with the reviewers in order to find a better solutions. According to such review practices, the quality of code review is heavily relied on this social interactions between patch authors and reviewers. Hence, the *politeness*, *respect*, and *dignity* rules should be applied in the code review process. Therefore, we assume that interpersonal treatments during interactions can influence the fairness in the code review process.

We propose that the *communication* can be considered to measure the procedural review fairness in code review practices. In a modern code review process, many communication channels exist to support developers. Developers can use the internet relay chat (IRC), email or comment in the code review system directly to communicate.

The concept of fairness can be applied to the code review context. Furthermore, we suspect that: (1) the review outcomes in the distributive review, (2) the behaviors of participants and the policies of projects in the procedural review, and (3) the interpersonal treatments in the interactional review, can influence the fairness in the code review process.

(RQ3) How do reviewers prioritize code reviews and how do prioritization strategies reflect review fairness?

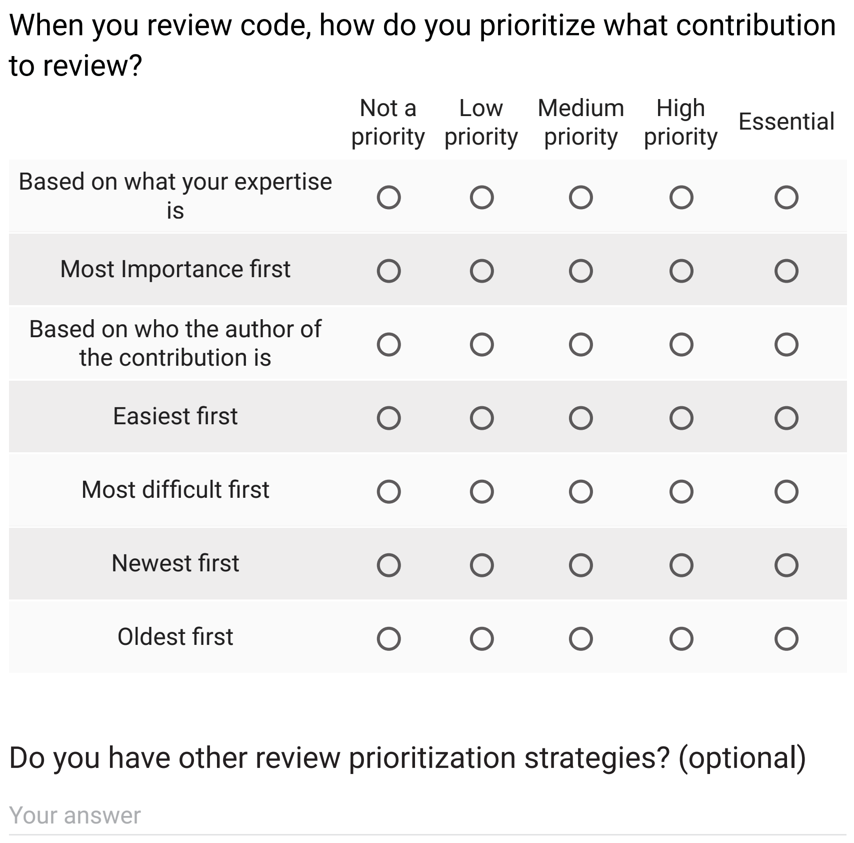
Our RQ2 has shown the concept of fairness in the code review context. Moreover, it is a likely case that the behaviors of participants in the procedural review can reflect the fairness in the code review process. Hence in RQ3 we further investigate how do reviewers select and prioritize a patch to examine.

Approach

To address RQ2, we conduct an online survey with OpenStack developers. We asked the developers how they prioritize a patch to review. Figure 5.8 shows the overview of the survey. In the survey, we provided five possible prioritization strategies, i.e., a patch is selected based on (1) the developer's expertise, (2) the importance, (3) the author of the patch, (4) the difficulty (the easiest or the most difficult patch first), or (5) the freshness (the newest or the oldest patch first). For each strategy, we provided a 5-point unipolar scale of the priority. To complement our predefined prioritization strategies, we also included an opened-ended question to collect the additional responses if developers have other prioritization strategies. We provided a free-text box for this open-ended question.

Results

Among 2,870 OpenStack developers, we received 213 responses, with a response rate of 7.4%. We also received 48 open-ended responses from the free-text ques-


When you review code, how do you prioritize what contribution to review?

	Not a priority	Low priority	Medium priority	High priority	Essential
Based on what your expertise is	○	○	○	○	○
Most Importance first	○	○	○	○	○
Based on who the author of the contribution is	○	○	○	○	○
Easiest first	○	○	○	○	○
Most difficult first	○	○	○	○	○
Newest first	○	○	○	○	○
Oldest first	○	○	○	○	○

Do you have other review prioritization strategies? (optional)

Your answer

Figure 5.8.: An overview of the developer survey for patch prioritization strategies.

tion of the survey. Base on the responses, we are able to obtain a brief understanding on how OpenStack reviewers prioritize patches and how these prioritization strategies affect the unfairness issues during the code review process. Figure 5.9 shows the summary of the survey responses using stacked bar chart. Table 5.4 shows the categories of patch prioritization and their respective frequencies, which are based on the open-ended responses. In below, we present our findings according to our prioritization strategies.

Expertise: We found that patches where the reviewers have related expertise tend to have higher priority. Figure 5.9 shows that 43% of the respondents prioritize the patches where they have the related expertise as *essential*. Moreover, 10% and 41% of the respondents prioritized such patches as medium and high,

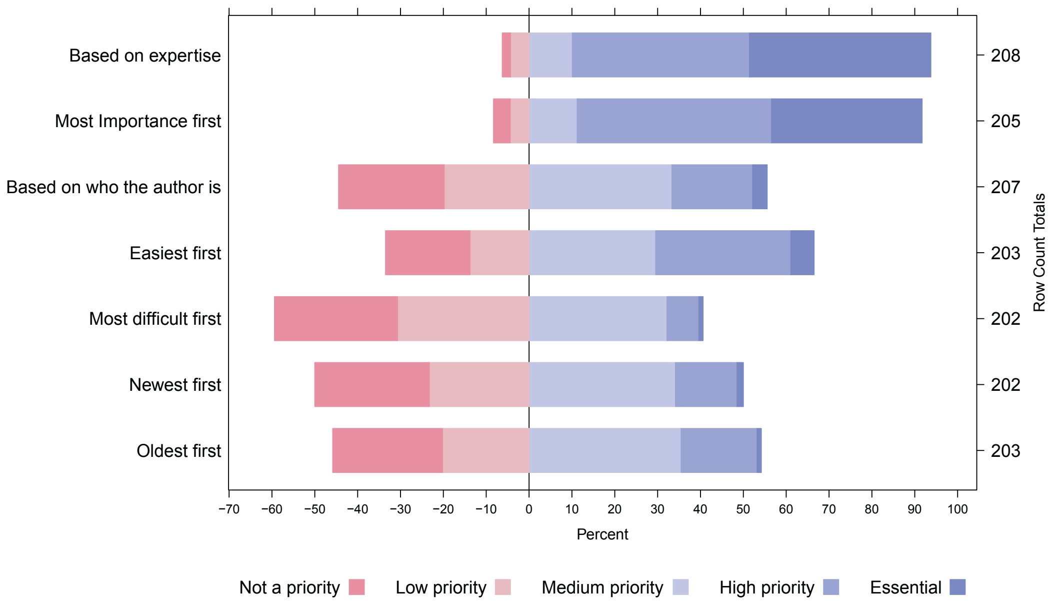


Figure 5.9.: Survey responses of patch prioritization strategies.

respectively. Only 6% of the respondents prioritize such patches as no or low priority. Table 5.4 also shows that the respondents tend to select a patch based on their expertise (12 of 48 responses). From these responses of the open-ended question, we found two main reasons that lead the respondents to select a patch based on their expertise. One reason is to increase the productivity of reviewing code:

"Usually, I try to look for places where I have experience so that I can maximize the time I have available to contribute."

Another reason is to ensure the quality of reviewing code, i.e., reducing the likelihood of misunderstanding the effects of the code to be reviewed and of making mistakes during the review:

"I focus on the projects that I have experience. To increase the quality of code review, I am always reviewing very carefully."

Importance: Similar to the expertise, we find that the important patches tend to have higher priority. Figure 5.9 shows that 35% of the respondents prioritize the important patches as essential, 46% of them as high, and 11% of them as medium. On the other hand, only 8% of the respondents prioritize the important patches as no or low priority. From the open-ended question, the respondents stated that they prioritize the patches with high impacts or the patches that may block other patches, e.g., patches with dependency on others (5 of 48 responses), security-related issues (3 of 48 responses), and bugs (2 of 48 responses):

"security requirements are on essential priority" and *"dependencies: reviews which are dependencies to other reviews are on higher priority."*

Who: We found that there is priorization based on who the patch author is. Figure 5.9 shows that 56% of the respondents gave a medium to essential priority to the patches that are made by particular developers. Table 5.4 shows that 10 of 48 respondents prefer to review the patches made by the patch authors who the respondents know or those who come from the same affiliation:

"I respond to requests to review code from the (patch) author or team I'm working with." and *"Those who have tagged me as a reviewer, or that I have reviewed a previous patchset of first."*

Difficulty: We found that easy patches have a higher chance to be reviewed before more difficult patches. Figure 5.9 shows that 67% of the respondents prefer to review the easiest patches first with medium to essential priority, while only 40% of the respondents prefer to review the most difficult ones. From the open-ended question, 6 of 48 respondents confirmed that they prioritize the patches that are perceived as easy to review. However, no response describes any reason why some respondents prioritize the complex patches to review first:

"I often try to review the changes that are most likely to get merged quickly first. Often this is the code reviews that are easy or do not contain a lot of changes in the same patch."

Besides the difficulty, we also found that another prioritization strategy is selecting patches that already have positive feedback from other reviewers or that have passed the integration tests (9 of 48 responses):

"I try to prioritize patches with positive feedback from reviewers (first) and CI (second)."

Freshness: Though we designed the freshness as two possible strategies (the newest or the oldest patch first) in closed-ended questions, only one respondent stated that he or she prefers to prioritize the patches from the freshness from the open-ended responses. This respondent expressed that he or she equalizes the review opportunities for all the patches in order to reduce the number of ignored patches (i.e., old patches first).

"I sometimes pick patchsets mostly at random that have been around for a bit, just to try to reduce the problem of patchsets that sit around without getting sufficient review."

Others: Besides our predefined prioritization strategies, we obtain the other strategies that were reported by some respondents. Several respondents (4 of 48 responses) also stated that they prioritized a patch that benefits them or aligns their interests (i.e., egocentric):

"The ones that are most critical for my solution." and *"Area of codebase, where I have my interest - to figure out the changes in the landscape, which may potentially impact my work"*

Furthermore, 2 of 48 respondents reported that they prefer to review the patches without prioritization strategies:

"Random: something catches my eye when it's committed...."

One respondent described that he or she prioritizes patches based on the business requirements of the OpenStack system, rather than the needs of individuals or organizations:

"....Look for features or bugs that have cross-company consensus (again based upon shared customer requirements, real-world use cases). I lower the priority of things where it is apparent a single company is trying to force in proprietary code...."

Although the goals for most of the prioritization strategies tend to increase the productivity of reviewing code (i.e., expertise, importance, difficulty), there also is a likely case that patches are prioritized based on the author of the patches. Such practices can indicate potential unfairness in the code reviews process.

(RQ4) What does the data tell us about the practical review fairness?

We want to further investigate whether the patch authors related bias has an effect on the distributive review fairness. In a fair review system, there should not be an association between the reviewing time and the affiliation of the patch author.

Human factors Influence

Approach

To address our RQ3, we performed a quantitative analysis to investigate the effect of patch author's affiliation on the reviewing time. In particular, we studied the patch author's affiliation from three perspectives: (1) *same affiliation*, i.e., a patch author who comes from the same affiliation as the reviewers, (2) *core affiliation*, i.e., a patch author who comes from the core affiliation (i.e., the affiliation that has OpenStack core members), and (3) *self-approved*, i.e., a patch author who is one of the reviewers. Furthermore, we studied the reviewing time in terms of the *feedback delay*, i.e., the time from a patch submission to its initial feedback and the *review length*, i.e., the time from a patch submission to its integration. Below, we describe our data preparation and analysis approaches.

Data preparation:

We used the review dataset from our data study [86] which describes the patch information, the personnel involved, and review discussion history.¶ This dataset captures review history from July 2011 to March 2015. Furthermore, we used the dataset of Gonzalez-Barahona et al. [32] to include the affiliation information of the personnel involved.‖ We then linked the information between our dataset and the affiliation dataset using the `username` field in the affiliation dataset and the `user-id` field in our dataset. For the developers that we cannot identify their affiliation (221 developers in total), we manually examined their personal websites and social media (e.g., GitHub, Twitter, and LinkedIn) to identify their

¶`http://kin-y.github.io/miningReviewRepo/`

‖`http://gsyc.es/~jgb/repro/2015-msr-grimoire-data/`

affiliations.

Once the review datasets are linked, we cleaned the data in order to ensure the accuracy of the study results. Since the OpenStack project is composed of several sub-projects, we selected only the official sub-projects for our study.** Then, we selected the patches where their reviews are closed (i.e., a review that is marked as merged or abandoned). We also removed the patches that are generated by automatic tools, since our study is focused on human involvements. Table 5.5 provides an overview of the dataset that we used for this study.

Finally, we identified a patch corresponding to the three perspectives of patch author's affiliation, i.e., same affiliation, core affiliation, and self-approved. We identified a patch as a same affiliation patch if the author of the patch has the same affiliation as one of the reviewers of the patch. Otherwise, we identified the patch as a different affiliation patch. We identified a patch as a core affiliation patch if the affiliation of the patch author has at least one core member. Otherwise, we identified the patch as a peripheral affiliation patch. We identified a patch as a self-approved patch if the patch author has approved the patch for integration. Otherwise, we identify the patch as a peer-approved patch. Table 5.6 shows the results of our classifications.

Data analysis:
Since several studies have found that the reviewing time and the patch size often related to each other [14,38], we also check the correlation between the reviewing time and the patch size using Spearman's correlation [41].

Then, we used the one-tailed Mann-Whitney U tests to statistically confirm the differences between the reviewing time (i.e., the feedback delay and the review length) of the patches that are identified as same affiliation (core affiliation or self-review) and the reviewing time of the patches that are classified as different affiliation (peripheral affiliation or peer-review) ($\alpha = 0.05$). We also used Cliff's δ [47] to measure the effect size, i.e., magnitude of the differences. Cliff's δ is considered as negligible for $\delta < 0.147$, small for $0.147 \leq \delta < 0.33$, medium for $0.33 \leq \delta < 0.474$, and large for $\delta \geq 0.474$.

**`http://git.openstack.org/cgit/openstack/governance/tree/reference/projects.yaml`

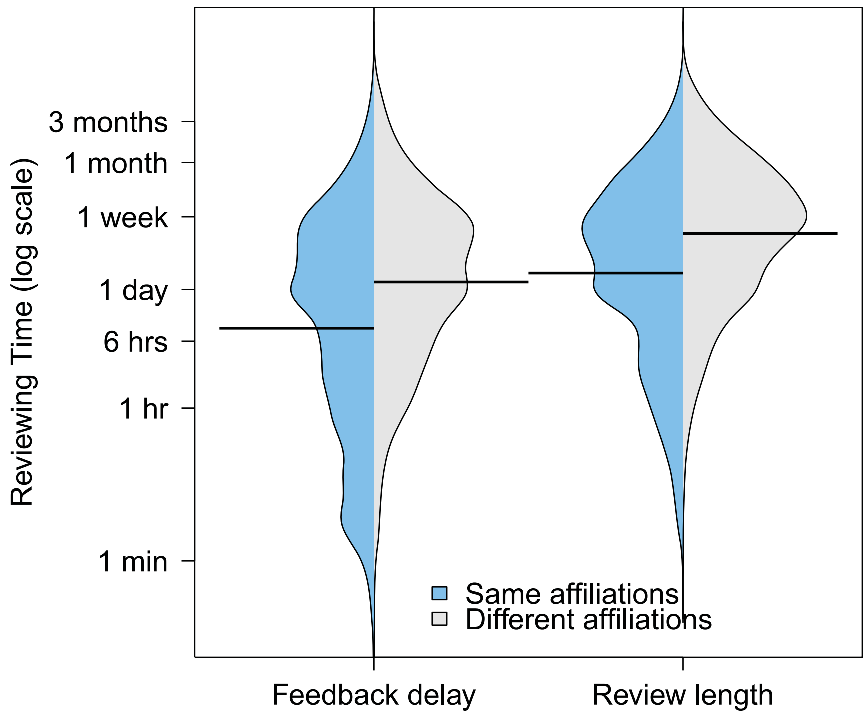


Figure 5.10.: A comparison of reviewing time between patches where the patch authors come from the same affiliations (blue) and the different affiliations (gray) as the reviewers of the patches. The horizontal lines show the median values of the distributions.

Results

From the correlation analysis, we find that the correlation between the reviewing time and the patch size is relatively small, i.e., Spearman's correlation of 0.19 for feedback delay and Spearman's correlation of 0.32 for review length [41].

The same affiliation patches tend to undergo reviews with shorter reviewing time than the different affiliation patches do. Figure 5.10 shows that the feedback delay of the same affiliation patches is shorter than that of the different affiliation patches. The median values of the feedback delay are 15 hrs for the same affiliation patches and 36 hrs for the different affiliation patches. Similarly, Figure 5.10 also shows that the review length of the same affiliation patches is shorter than that of the different affiliation patches. The median values of the review length are 2.1 days for the same affiliation patches and and 5.7 days for the different affiliation patches. Mann-Whitney U tests confirm that the differences are statistically significant (p-value < 0.001 for both feedback delay and review length), with small effect size ($\delta = 0.23$ for feedback delay and $\delta = 0.25$ for review length).

The patch authors who do not come from the same affiliation as the reviewers are more likely to suffer from a longer (small effect) review waiting of patch examination than the patch authors who come from the same affiliation as the reviewers.

The core affiliation patches tend to undergo reviews with shorter reviewing time than the peripheral affiliation patches do. Figure 5.11 shows that the feedback delay and the review length of the core affiliation patches are shorter than those of the peripheral affiliation patches. The median values of feedback delay are 1.1 days for the core affiliation patches and 1.8 days for the peripheral affiliation patches. The median values of the review length are 4.2 days for the core affiliation patches and 7 days for the peripheral affiliation patches. Mann-Whitney U tests confirm that the differences are statistically significant (p-value < 0.001 for both the feedback delay and the review length). However, we find that the effect size is negligible with a δ value of 0.11 for the feedback delay and a δ value of 0.14 for the review length.

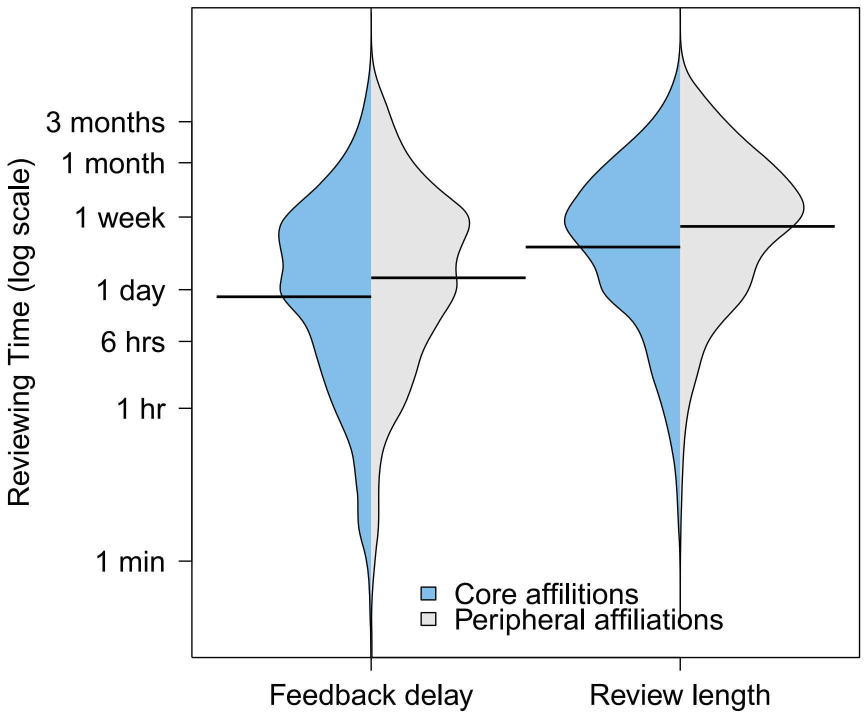


Figure 5.11.: A comparison of reviewing time between patches where the patch authors come from the core affiliations (blue) and the peripheral affiliations (gray). The horizontal lines show the median values of the distributions.

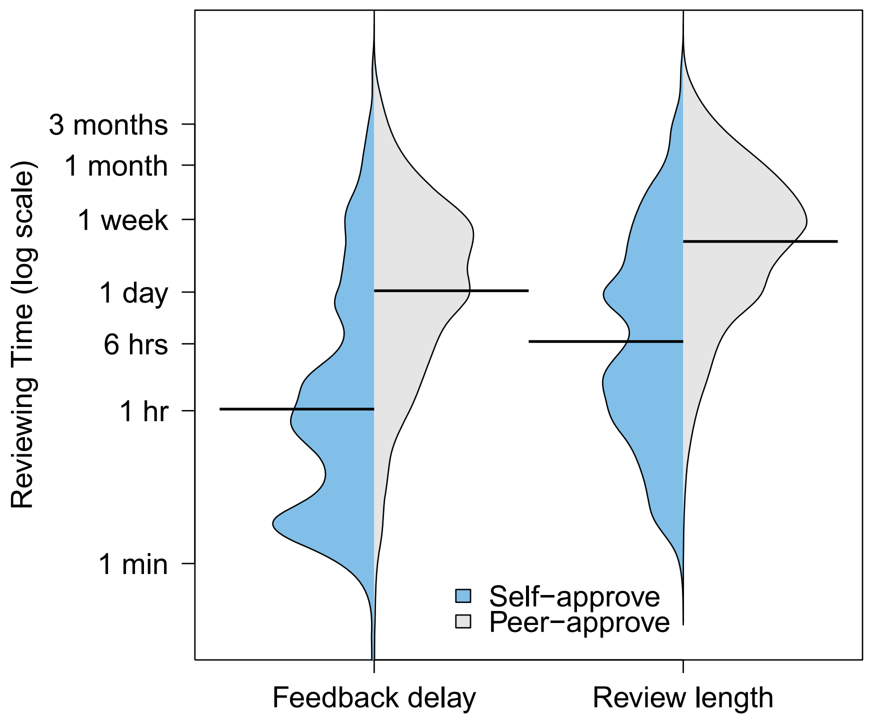


Figure 5.12.: A comparison of reviewing time between patches where the patch authors are one of the reviewers (blue) and those where the patch authors are not one of the reviewers (gray) as the reviewers of the patches. The horizontal lines show the median values of the distributions.

Although the differences are statistically significant, there is a less likely case that the patch authors come from peripheral affiliations will suffer from a long review waiting.

The self-approved patches tend to undergo reviews with shorter reviewing time than the peer-approved patches do. Figure 5.12 shows that the feedback delay and the review length of the self-approved patch are shorter than those of the peer-approved patches. As for the median values, the feedback delay of the self-approved patches is one hour, while the feedback delay of the peer-approved patches is one day. Similarly, the median values of the reviewing time are 6 hrs and 5 days for the self-approved patches and the peer-approved patches, respectively. Mann-Whitney U tests also confirm that the differences are statistically significant (p-value < 0.001 for both feedback delay and review length). Moreover, the effect size of the differences are large for both the feedback delay ($\delta = 0.57$) and the review length ($\delta = 0.56$).

The self-approved practices significantly affect the reviewing time.

Prioritization Influence

We further investigate that how we can measure the review fairness in a prioritization review system. Below, we present our quantitative approach of adopting a queuing system model to measure fairness in a code review system, followed by the results of statistical analysis.

Approach

Since we found that prioritization behaviors strongly affect the reviewing time in code review processes, we regard a code review process as an time ordering process, which represents every patch in the code review process has an ordinal position according to the time (e.g., the arrival time of a patch). Thus, we apply a queuing system model from the perspective of social justice (fairness) [9, 10] to address RQ4. Ordered queues have been used in many scenarios of real life, such as supermarket, hospitals, airports, computer (network) systems and many other systems [58]. A queue conducted by people can be regarded as a miniature of social construct, and the main reason for using this formation is to maintain the social fairness of participants.

In this study, we use two disciplines of queuing system that have been regarded as most fair from the perspective of service providers. The most common, natural, and traditional queuing disciplines is first-in-first-out (FIFO), or first-come-first-served (FCFS). The FIFO discipline comes from the practices of queues where the total amount of service is limited by the resource of service system. For example, the queuing system in a supermarket is a typical FIFO discipline: customers are lining up in front of several counters, and the cashiers will serve the customers one by one. However, FIFO is not the only fair queue discipline, prioritizing short jobs ahead of long jobs can also be treated as fair, which is known as shortest-job-first (SJF) or shortest-job-next (SJN). Prior study has proved that waiting in a queue can cause emotion changes [81]. Avi-Itzhak et al. simulate a scenario happened in a supermarket [9]: a customer holding only one item lines behind another customer carrying a fully loaded cart of items. In the viewpoint of a cashier, would it be more fair to serve the customer with only one item ahead of the customer with a fully loaded cart? This case may be fair, if the cashier regards fairness from the service time principle: *the one who demands the least of the server's time should be served first.* Thus, we use both the FIFO and SJF disciplines to measure fairness in the code review queuing system.

A code review process can be regarded as a *blind* queuing system. OpenStack adopts Gerrit as the review tracking system, every patch (or revision) must be committed and stored in the pending repositories of Gerrit. Despite the review system in OpenStack is not designed as a typical queuing system that follows the FIFO discipline, the OpenStack project managements still suggest that reviewers should prioritize the patches by their arrival time (i.e., the FIFO discipline)[††]. Moreover, from the observation of Figure 5.9, we found that 67% of the respondents choose easiest patches as priority, which implies the SJF discipline. Thus, we apply a queuing system mode based on the code review practices. In the code review queuing system, we assume that all patches can be ordered by different factors, such as the arrival times of patches or the difficulties of patch reviews. However, a code review queuing system is different from the queuing systems in supermarket or hospital. In code review process, a patch author submits a new patch and he/she may expect to get the review feedback in a certain time frame

[††]`https://wiki.openstack.org/wiki/Nova/CoreTeam#Review_Prioritization` see "Review Expectations"

(e.g., seven working days). On the other hands, a reviewer is able to choose any patch from the queue to perform reviews according to his/her prioritization strategies. However, the patch author may never know the actual position of his/her patch in the code review queue, which is prioritized by the reviewer. For example, this patch has been treated as a lower priority from reviewers, the patch author may only notice that the review cost relatively longer time than normal.

We define a code review timeline to present when code review related activities have occurred in each patch. Figure 5.13 is an example of the code review timeline. A patch in code review timeline includes two main timeframes: waiting and reviewing. Waiting timeframe presents the period from a patch first arrives to code review system until the first core-review happens to this patch. Reviewing timeframe presents the period from the first core-review of a patch, until the close time of this patch. We assume that three patches exist in a project as shown in Figure 5.13. After the prioritization of reviewers, we observe the following results:

Three patches P_1, P_2, and P_3 that arrived with different arrival times (i.e., $A_1 < A_2$ and $A_2 < A_3$. Observe the scenario in this example, we found that: (1) P_3 has been core-reviewed earlier than P_1, and P_1 has been core-reviewed earlier than P_2 (i.e., $F_3 < F_1$ and $F_1 < F_2$), (2) P_3 has been closed earlier than P_2, and P_2 has been closed earlier than P_1 (i.e., $C_3 < C_2$ and $C_2 < C_1$).

Results

From the correlation analysis, we found that **the correlation between reviewing time and skip metrics is strong negative**, which implies the prioritization activities significantly affect the review outcomes. From this result, we identify the prioritization strategies are essential to maintain the fairness perceptions of review participants.

The target dataset is retrieved from the projects (the sub-project, a.k.a. components) in OpenStack. We detail the analysis of the entire OpenStack into several projects because the different project have different reviewer teams with different expertise as well. Finally, we choose most contributed six projects sorted by the amount of patches: Nova, Neutron, Heat, Keystone, Horizon, and Cinder. In our selected six projects, Nova project includes 14,296 changes, while Cinder includes 3,540 changes. We then calculate the orders of happening times for each

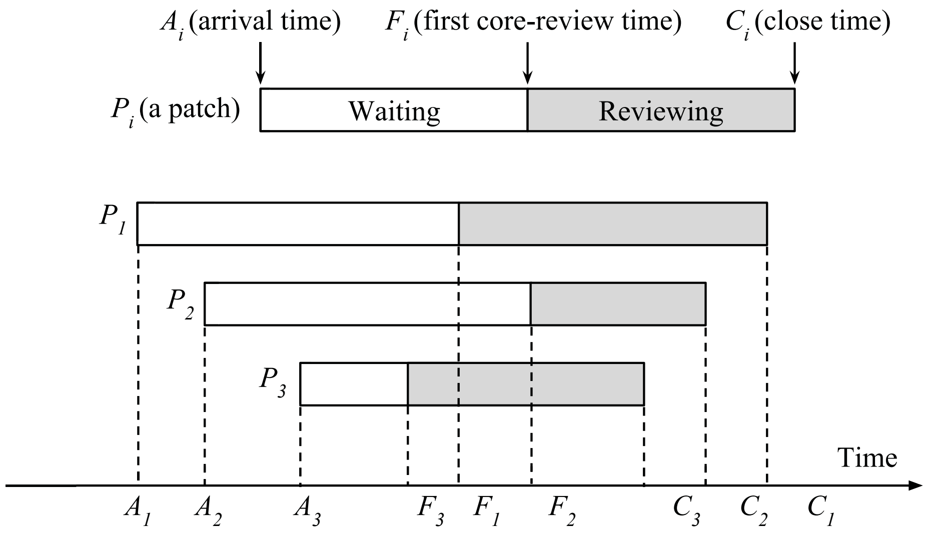


Figure 5.13.: An example of code reviews timeline. There are three patches with different review waiting time and reviewing time. The timeline presents the arrival time, first core-review time, and close time of each patch.

review activities such as submitted order, first core-review order, and close order. Finally, the skip metrics that measures the degree of prioritization activities can be calculated.

We calculate the correlation using Pearson method since the distribution of skip metrics is normal distribution in a review queuing system. We found strong correlation between the skip metrics and reviewing time, both first core-review (-0.90) and close review (-0.88). The results identify that the prioritization significantly influence the reviewing time. Moreover, since reviewing time can be a factor of fairness perceptions, the overuse of prioritization might bring unfair perceptions to developers and entire project.

5.3. Discussion

In this section, we discuss the implications of the results.

Why is fairness important for code review practices?

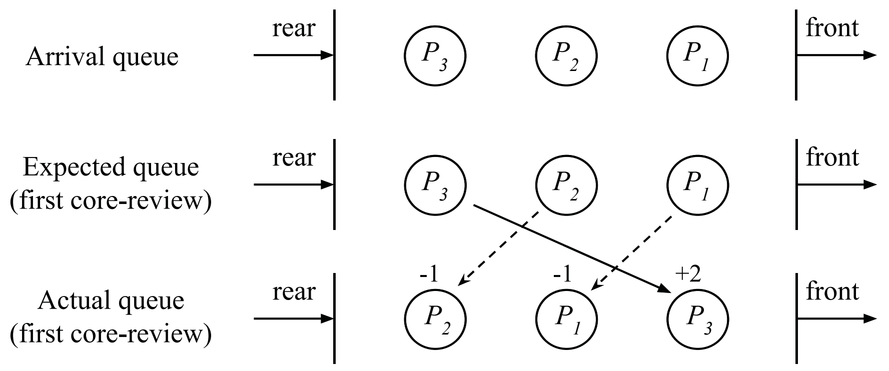


Figure 5.14.: An model of the code review queuing system. This case assumes that the arrival queue and the expected queue have the same order following the FIFO discipline. However, the actual queue is different with the expected queue: P3 skipped two positions (i.e., +2) and became the first patch to be reviewed by core reviewers. P1 and P2 kept the same order but all of them slipped one position (i.e., -1) from the expected queue.

The results of our RQ1 show that the concept of fairness can be applied to the code review context. From Table 5.2, we understand the importance of fairness and the rules that help to make a fair community. Fairness is important as it assures people to be treated fairly in a community. Fairness is also important for decision makers, employers, or the entire community. For example, in a fair system or community, participants are required to follow the rules, but they are also able to express their views when they have the needs.

A fair environment helps to establish a friendly, strong, and healthy community among people. Similarly, a project using code reviews requires a fair community since the code review activities highly depend on human involvement and social interactions. Therefore, to perform code reviews more efficiently, we suggest the practitioners to consider the potential fairness issues in the project communities.

How can we measure the code review practices by using the concept of fairness?

Here we do the complementary discussion of the results of RQ2 and RQ3: From the results of RQ2, we found that more than half of the reviewers prioritize the patches that are created by particular patch authors. According to the *consistency* rule in procedural fairness in Table 5.2, this prioritization practices do not meet the *consistency* rule in procedural fairness. Similarly, the *bias suppression* rule in procedural fairness requires the decision makers to be neutral. However, some reviewers report that they tend to select the patches that already have positive feedback.

From the results of RQ3, we found that there is an association between the reviewing time and the affiliations of a patch author. According to the *equality* rule in distributive fairness, reviewers should treat every patch author equally in the code review outcomes. However, reviewers spend different reviewing time for different person.

How can we improve the code review process by using the concept of fairness?

Since we already found that fairness plays an important role to make stable, and healthy code review process, we wonder how we can improve the review practices using the concept of fairness. Based on the findings of this study, we make several suggestions to practitioners:

(1) The development communities should make prioritization practices more open and transparent. The openness and transparency help to increase interactional fairness and distributive fairness in the code review process. In particular, since some prioritization strategies influence the fairness in the code review process, the communities can suggest the reviewers to use the strategies with less bias.

(2) Since the patch authors' background information (e.g., affiliation) significantly affects the code review outcomes, we suggest that communities can anonymize the user names of patch authors when reviewers perform the code review practices. Despite some reviewers may guess the patch authors correctly from their source code, or the project needs a "shortcut" to merge particular patches, we still provide this suggestion because it reduces the impact of first impressions to the patch authors.

The perspective of reviewers From the results of our survey in RQ2, we observed that OpenStack reviewers neither followed the FIFO discipline (i.e., Newest first), nor the last-in-first-out (LIFO) discipline (i.e., Oldest first) as shown in Figure 5.9.

5.4. Threats to Validity

External validity

One potential threat to external validity is that the number of projects that we studied. However, we focused on the concept of review fairness and this concept has not been defined in software engineering. Thus, we applied a exploratory research method and we designed three different approaches for three research questions. In the data collection, we collected the data from both developers and software repositories by using survey and data mining. As for the analysis, we applied a mixed method which combine both qualitative and quantitative approaches. Therefore, our main goal in this study is to establish a new concept of review fairness and discuss the best method to define and measure it. Nonetheless, further replication studies on other projects are needed to refine and generalize our results.

Construct validity

We mainly collected the data from online survey and code review system. One of the potential threats to construct validity is that we may not capture all the affiliation information for every developer. We can only include the people who have recognizable email domain, or free accessible personal information on the internet. It is difficult to gather information from some individual developers who use emails from public mailbox providers and have no personal information on the internet. However, the developers whose lack information of affiliation is less than 7% of the total number. Considering that OpenStack is an industrial leading OSS project, we had to ignore the individuals without public personal information. In addition, we examined that this type of developers contributed significant fewer than the median.

Internal validity

Our code review data is collected from the Gerrit repositories of the OpenStack project. However, Gerrit in OpenStack has several big changes on the API, the user interface and even the review criteria. These history events of OpenStack caused some inconsistencies of the dataset. For example, we discovered 3,087 patches (around 5.7% of all patches) that the patch's submit time is later than the patch's closed time. Similarly, the changes on the rules of the code review process might influence the reviewing time. We need more experiment to determine and solve this threats.

5.5. Related work

Prior work related to this study can be divided into two main aspects: studies on code review and studies on fairness theory.

Code Review

We introduce the most important related work with respect to code review studies. Bacchelli and Bird studied the motivations, the expectations, and the outcomes of modern code review [11]. Rigby et al. studied the review policies and examined which metrics have the largest impact on review efficacy in OSS projects [61]. Balachandran suggested using review-bot to reduce human effort and improve code review quality [12]. Bosu et al. investigated the factor of useful reviews to improve the effectiveness of code reviews [19]. Thongtanunam et al. studied traditional code ownership heuristics using code review activities [72]. Baysal et al. found the non-technical factors of code review can significantly influence the code review outcomes [13]. McIntosh et al. found that there exists a negative influence on software quality when the poorly-reviewed code is merged [49]. Jiang et al. found the experiences of the developers impact the patch acceptance and the reviewing time [38]. Tsay et al. found that in some case, even the submitter's contribution is rejected, the core team still fulfill the submitter's technical goals by implementing an alternative solution [77]. Tourani and Adams studied the impact of human discussions, which is related to the interactional fairness [75]. Yang et al. studied the social relationships among the patch authors and the reviewers [85].

Fairness Theory

We look into the concept of fairness in psychology field and we list the related work with respect to fairness theory research. Leventhal studied the equity is one of the rules of distributive justice, which implies that the rewards and resources should be distributed in accordance with people's contribution [21, 44, 45] Some studies have been done in the research of the procedural justice [69, 79]. Blodgett et al. examined the influences of distributive justice, interactional justice, and procedural justice on complainants' repatronage [17]. Colquitt studied distributive justice, interactional justice, and procedural justice in organizational justice [22]. Sindhav et al. performed survey to test the satisfaction of airport security from the perspective of perceived fairness [67]. Avi-Itzhak et al. studied the measurement of fairness in queuing system [9, 10].

5.6. Conclusions

A code review can be regarded as social interactions between a patch author and a reviewer of this patch. Though many studies have been done to understand code review techniques, little research has been directed towards the developers' behaviors and perceptions in code review practices.

In this study, we are interested in how the developers' behaviors influence the code review process, and whether the code reviews are performed fairly among the developers. We apply an exploratory research method since the concept of fairness has never been defined in the context of code review. First, we apply the concept of fairness in the context of code review through a literature survey. Second, we start an investigation on the behaviors of developers which have a potential impact on fairness. Final, we investigated whether this unfairness can influence the review practices.

We made the following main contributions in this study:

- Most participants in OpenStack perceived the whole system is fair, a few of them perceived unfairness.

- We proposed a fairness theory in the context of the code review process to study the fairness issues in review practices.
- Though projects have principles about prioritizing patches in code review practice, some reviewers use different methods. Prioritization might lead to potential unfairness perception by some developers.
- We are able to measure fairness metrics and discussed the differences between perceived fairness and practical fairness.

Our study shed light on the existence of unfairness in the code review process, which implies potential risks of the development communities. Our findings also suggest that the code fairness issues exist, even in a system that most developers perceived to be fair. However, practitioners should consider to reduce the unfairness perception of developers when design or improve the code review process.

Table 5.2.: Types of Fairness from The Results of The Literature Survey.

Categories	Rules	Description	Literature
Distributive Fairness	Equity	Rewards and resources are distributed in accordance with one's contribution	[17, 22, 44
	Equality	All parties receive the same outcome regardless of contribution	[17, 22]
	Need	The outcome meets the requirements of the recipient	[17, 22]
Procedural Fairness	Control	The abilities to (1) voice one's views and (2) influence the outcomes	[17, 22, 67
	Consistency	The process is applied consistently across time and persons	[17, 22, 67
	Bias suppression	Decision makers are neutral	[17, 22, 67
	Information Accuracy	Information are not based on inaccurate information	[17, 22, 67
	Correctability	Appeal procedures exist for correcting bad outcomes	[17, 22]
	Ethicality	The process upholds personal standards of ethics and morality	[17, 22]
Interactional Fairness	Politeness	Whether one has been treated with politeness	[17, 22]
	Respect	Whether one has been treated with respect	[17, 22, 67
	Dignity	Whether one has been treated with dignity	[17, 22]

Table 5.3.: Review Fairness in Code Review Practices.

Categories	Rules	Description	Measurement
Distributive Review Fairness	Equity	the review outcome corresponds to developer's contribution	Reviewing time Review feedback
	Equality	Every patch receives the same outcome regardless the contributions	
	Need	the review outcome meets the requirements of projects	
Procedural Review Fairness	Control	The abilities to (1) voice one's views (2) influence the outcomes	Communication
	Consistency	Review process is applied consistently across time and persons	Review priority
	Bias Suppression	Reviewers are neutral	
	Information Accuracy	Information are not based on inaccurate information	Project policy
	Correctability	Appeal procedures exist for correcting bad outcomes	
	Ethicality	The process upholds personal standards of ethics and morality	
Interactional Review Fairness	Politeness	One has been treated with politeness	Communication
	Respect	One has been treated with respect	
	Dignity	One has been treated with dignity	

Table 5.4.: Categories and subcategories of review prioritization strategies that emerged from the open-ended responses.

Category	Sub Category	Frequency*
Expertise		12
Importance	Dependency	5
	Security issues	3
	Bugs	2
Who		10
Difficulty	Easy patches first	6
	Positive feedback	9
Freshness		1
Others	Egocentric	4
	Random	2
	Business	1

*Since one responses can have several prioritization strategies, the sum of the frequency is higher than the total number of the responses.

Table 5.5.: An Overview of the OpenStack Review Dataset

Review data		Personnel	
# Patches	49,886	# Authors	2,250
Median of Churn	30	# Reviewers	2,870
Median of Reviewers	3	# Core Members	462
Median of Comments	4	# Affiliations	229

Table 5.6.: Patch Classification Results

	#Patch		#Patch
Same affiliation	11,805 (24%)	Different affiliation	38,081 (76%)
Core affiliation	40,478 (81%)	Peripheral affiliation	9,408 (19%)
Self-review	2,030 (4%)	Peer-review	47,856 (96%)

Chapter 6.

Conclusion

The main contribution of this dissertation is understanding the human factors and social aspects in modern code review. Though many studies have been done from the technical parts like source code quality, post-release bugs in OSS code reviews and traditional industry code inspections, there are few study focus on the code review participants: their roles, their behaviors, their perceptions, and the relationship among them. We explore these human and social related questions and suggest practitioners that the software developments is a social collaboration and we should consider the human factors and social aspects of it.

The primary contribution of this work is list as following parts.

(1) Social structure of code review community is essential to evaluate the performance of developers.

Our findings identify that modern code review methods and techniques changed with the work flow of development life cycle. Findings also present that human factor significantly influences code review process and outcomes such as behaviors. Furthermore, we find interactions among developers such as treatments to patch contribution affect the perceptions of people and it influences the fairness perceptions.

As proposed human factor should affect OSS review process, SNA approach was applied into this case study has researched three OSS projects. Then reviewers can be classified into several role groups with significant differences. The results show there is a strong correlation between the activities of most important contributors and their network positions. Network measures distributions of contributors can be used for evaluating contributors' activeness. For example, project managements can identify contributors who are enthusiastic but in a specialized field, and contributors who are in important network position but unenthusiastic.

(2) Reviewers' behaviors strongly affect the review process and outcomes.

A code review can be regarded as social interactions between a patch author and a reviewer of this patch. Though many studies have been done to understand code review techniques, little research has been directed towards the developers' behaviors and perceptions in code review practices.

We are interested in how the developers' behaviors influence the code review process. We apply an exploratory research method to study the prioritization of reviewers. We found that even a project have principles about how to perform code review, the actual review prioritization strategies vary among reviewers and projects. From the results, we found that prioritization activities significantly influence the review outcomes like reviewing time. Furthermore, the affiliations of authors, self reviews strongly affect the reviewing time as well.

(3) Fairness issues exist in code review process and should not be ignored.

Since we already know review prioritization might affect code review outcome strongly, we further investigate what is the influence and what is the degree of this influence. We introduce the concept of fairness into code review practices. Since the concept of fairness has never been defined in the context of code review. First, we apply the concept of fairness in the context of code review through a literature survey. Second, we start an investigation on the behaviors of developers which have a potential impact on fairness. Final, we investigated whether this unfairness can influence the review practices. The results identify that review fairness is based on perceptions of developers more than the results of review. A system that is perceived to be fair by most participants might be perceived as unfair by some developers. Practitioners should consider the existence of fairness when they design code review processes and policies, and care more about the potential negative influence to other developers.

In our future plan, we plan to replicate our approaches to more software projects. Since different projects have different review processes and different review techniques, we might need more experiments to evaluate the usefulness of our approaches. We believe our study can help to identify the human factors and social aspects that influence the review communities, and find out the weak points for the potential improvements in processes or community structures.

Appendix A.

Mining Repositories

A.1. Database Schema

Fig A.1 represents the data schema of the code review repositories we mined.

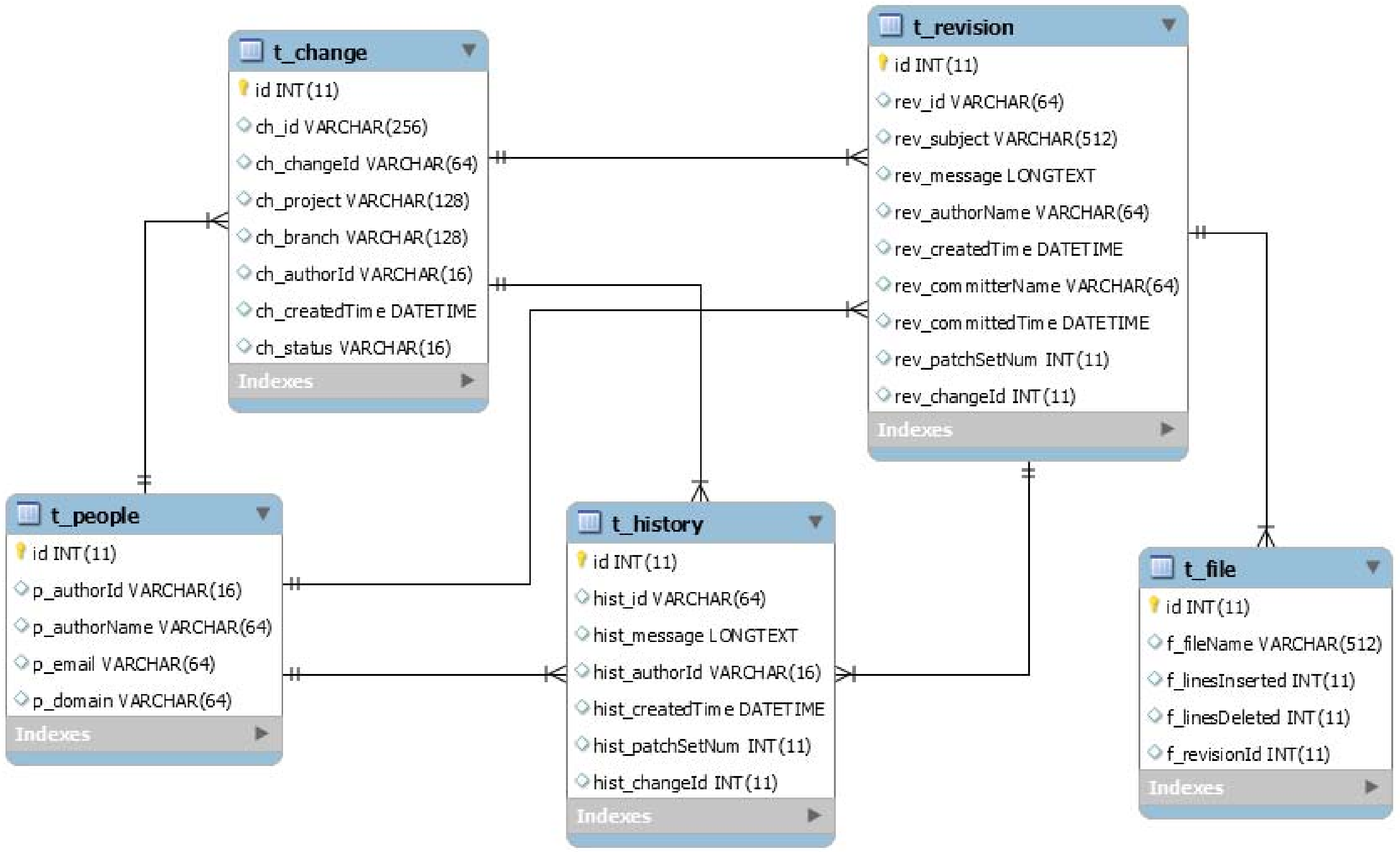


Figure A.1.: Schema of code review database

A.2. Details of Tables

Here we introduce the fields of each table.

Change (t_change)

The change table contains the information of changes. In Gerrit, a change represents a set of patches related to the same topic (e.g., fix a bug or add a new feature). A change is unique, but it can include more than one revision, which means the patch author can repeatedly update the change by committing new revisions with the same change-id.

Table A.1.: Table t_change and field details

PK/FK	Field Name	Description
Primary Key	id	Unique id of changes in database (auto increment)
	ch_Id	A combination of project name, branch name and change id
	ch_changeId	Change id in Gerrit
	ch_project	Project name (repository name) of change
	ch_branch	Branch name of change
Foreign Key	ch_authorId	Author's id of change
	ch_createdTime	Timestamp of when change was created
	ch_status	Review status of change, i.e., NEW, MERGED and ABANDONED

Revision (t_revision)

The revision table contains the information of revisions, which refer to the patch commit history. Notice that you cannot get people's id from revision table directly, because username and email address are the only information stored in Git.

People (t_people)

The people table contain the developers information who participate in Gerrit code review. We have hashed the names and email of developers using SHA-1 to protect the privacy of developers. However, we remain the domain of email address for each developer.

History (t_history)

Table A.2.: Table t_change and field details

PK/FK	Field Name	Description
Primary Key	id	Unique revision id in database (auto increment)
	rev_Id	Commit id of revision
	rev_subject	Subject of revision
	rev_message	Message of revision
	rev_authorName	Author's name in the revision (SHA-1)
	rev_createdTime	Timestamp of when revision was created
	rev_committerName	Committer's name in the revision (SHA-1)
	rev_committedTime	Timestamp of when revision was committed
Primary Key	rev_patchSetNum	Revision number in change
Foreign Key	rev_changeId	Change that the revision belongs to

Table A.3.: Table t_change and field details

PK/FK	Field Name	Description
Primary Key	id	Unique people id in database (auto increment)
	p_id	Id of developer in Gerrit
	p_name	Name of developer in Git (SHA-1)
	p_email	Email address of developer (SHA-1)
	p_domain	Domain of email address

The history table records the review comment history. Every review comment can be mapped to certain change and revision.

File (t_file)

The file table contains the files and churn (lines of code that added and deleted) information. One file can be modified in various propose and involved in different revisions and changes.

Table A.4.: Table t_change and field details

PK/FK	Field Name	Description
Primary Key	id	Unique comment id in database (auto increment)
	hist_id	Comment id in UUID form
	hist_message	Review comment message
Foreign Key	hist_authorId	Author id of review comment
	hist_createdTime	Timestamp of when review comment was created
Foreign Key	hist_patchSetNum	Revision number that review comment was created for
Foreign Key	hist_changeId	Change id that review comment was created for

Table A.5.: Table t_change and field details

PK/FK	Field Name	Description
Primary Key	id	Unique file id in database (auto increment)
	f_fileName	The path and name of file
	f_linesInserted	Number of inserted lines
	f_linesDeleted	Number of deleted lines
Foreign Key	f_revisionId	Revision id that file belongs to

A.3. Example of query

Here are some examples about how to query our database.

Query the number of changes by status

Default status in Gerrit includes Merged, Abandoned and New

```
SELECT COUNT(*), ch_status FROM t_change
GROUP BY ch_status;
```

Query all the review with -2 vote

```
SELECT * FROM t_history
WHERE hist_message LIKE '%Do not submit%'
OR hist_message LIKE '%Code-Review-2%'
ORDER BY hist_createdTime ASC;
```

Query all the review with -1 vote

```
SELECT * FROM t_history
WHERE hist_message LIKE '%I would prefer that
you didn\'t submit this%'
OR hist_message LIKE '%Code-Review-1%'
ORDER BY hist_createdTime ASC;
```

Query all the review with +1 vote

```
SELECT * FROM t_history
WHERE hist_message LIKE '%Looks good to me,
but someone else must approve%'
OR hist_message LIKE '%Code-Review+1%'
ORDER BY hist_createdTime ASC;
```

Query all the review with +2 vote

Note that some projects have different rules about patch approvals.

```
SELECT * FROM t_history
WHERE hist_message LIKE '%Looks good to me, approved%'
OR hist_message LIKE '%Code-Review+2%'
ORDER BY hist_createdTime ASC;
```

Query all the core reviewers

Here we define the core reviewers as the reviewers who voted +2 or -2

```
SELECT distinct hist_authorId FROM t_history
WHERE hist_message LIKE '%Looks good to me, approved%'
OR hist_message LIKE '%Code-Review+2%'
OR hist_message LIKE '%Do not submit%'
OR hist_message LIKE '%Code-Review-2%'
ORDER BY hist_createdTime ASC;
```

A.4. Obtain the source code

In this example, we obtain the diff of commits (revisions) in the source code repositories of OpenStack Nova Project.

1) Clone the repositories.

```
git clone https://git.openstack.org/openstack/nova
```

2) Check the patchsets (revisions) of changes. For example, we examine a patchset (change-id = 176805). You will get a list of patchsets of this change.

```
git ls-remote | grep /176805/
```

3) Get the revisions and create new branches. Here we create two new branches based on patchset 1 and patchset 2.

```
git fetch https://review.openstack.org/openstack/nova
refs/changes/05/176805/1
git branch change/176805/1 FETCH_HEAD

git fetch https://review.openstack.org/openstack/nova
refs/changes/05/176805/2
git branch change/176805/2 FETCH_HEAD
```

4) Compare the diff file. You can also save the diff to local.

```
git diff change/176805/1 change/176805/2
```

or

```
git diff change/176805/1 change/176805/2 >
diff-176805-1-2.txt
```

(Similar shell script to fetch all the diff, please check https://github.com/saper/gerrit-fetch-all/blob/master/gerrit-fetch-all)

Appendix B. Online Survey

B.1. Request Email

We sent personalized email to each potential participate in OpenStack Project for the survey of code review fairness. The goal of this request was to show investigate the factors that influence developers to perceive fairness or bias in peer review processes.

```
Dear {PARTICIPANT NAME},

I am a PhD student at Nara Institute of Science and
Technology, Japan, and I work with a group of researchers
from Canada and Spain. We are researching the code review
 process and effectiveness in OpenStack.

We are interested in your perception of the code review
process. Therefore we have built a small survey composed
of 5 questions. This survey should take 3-10 minutes to
answer. The results of this research will be shared with
the OpenStack community.

This survey can be accessed at
http://sdlab.naist.jp/members/kin-y/survey.html
```

```
For more information about our research group, please
visit http://kin-y.github.io/groupCodeReview/
Furthermore, it will be very appreciating if you can
provide us with your feedback.

Thank you in advance.

Sincerely,

Xin
```

B.2. Survey Information

We present the details of our online survey as follows:

Title

OpenStack Survey on Fairness in Code Reviewing

Subtitle

A survey on how fair code reviewing process is in Free/Open Source Projects.

Research Goal

We want to investigate the factors that influence developers to perceive fairness or bias in peer review processes. As perception is very subjective, we have launched a survey and are asking OpenStack developers to give us their input.

Survey Period

1st Announcement: 22 Feb 2016
2nd Announcement: 29 Feb 2016
Closing: 7 Mar 2016

Details

Both industrial software projects and Free and Open Source Software (FOSS) projects have adopted peer review as an important quality assurance technique in their software development process.

Many studies have been done to understand the peer review techniques, but little research has been directed towards the developers' personal understanding of the peer review process. In this paper, we propose a novel approach to investigate the factors that affect developers to perceive fairness or bias in peer reviews. We define review fairness metrics to measure the degree of fairness perceived by developers who participated in the peer review process.

Statistical models are used to determine the factors that affect the perception of fairness and help in predicting if a review is fair or not. These metrics are evaluated in an empirical study of the peer review process of OpenStack, a large FOSS project mostly contributed and managed by several large commercial organizations.

In addition, we ask developers about how they perceive the code reviewing process in regard to its fairness. Therefore, a survey has been created and OpenStack developers have been kindly invited to participate.

Research Team

This survey is carried out by an international group of researchers with ample experience in the study of Free/Open Source Software projects.

Xin Yang
PhD student, Nara Institute of Science and Technology, Japan.

Germán Poo Camaño
PhD student / developer, University of Victoria, Canada.

Daniel M. Germán
Professor, University of Victoria, Canada.

Gregorio Robles
Associate Professor, Universidad Rey Juan Carlos, Spain.

B.3. Survey Questions

Description We are researching the code review process in OpenStack and its effectiveness. We are interested in the developers' perception of the code review process, either as a reviewer, patch contributor or both.

This survey should take around 3-10 minutes to answer 5 short questions.

The results of this research will be shared with the OpenStack community.

For more information (goals, research team, etc.), please go to http://kin-y.github.io/groupCodeReview

Questions

1. How many hours a week do you usually spend reviewing code (single choice)

- less than 1 hour
- 1 $\sim$ 2 hours
- 2 $\sim$ 4 hours
- 4 $\sim$ 8 hours
- 8 $\sim$ 15 hours
- 15 $\sim$ 20 hours
- 20 $\sim$ 40 hours
- more than 40 hours

2. When you review code, how do you prioritize what contribution to review?

Do you have other review prioritization strategies? (optional)

3. According to your experience as a contributor, have your contributions been treated unfairly?

	Not a priority	Low priority	Medium priority	High priority	Essential priority
Oldest first					
Newest first					
Most Importance first					
Most difficult first					
Easiest first					
Based on what your expertise is					
Based on who the author of the contribution is					

- never
- rarely
- occasionally
- often
- always

Feel free to explain or provide evidence for your answer (optional)

4. According to your experience as a reviewer, do you perform code reviews unfairly?

- never
- rarely
- occasionally
- often
- always

Feel free to explain or provide evidence for your answer (optional)

5. In general, the code review process in OpenStack is fair.

- strongly agree
- agree
- neutral
- disagree
- strongly disagree

Feel free to explain or provide evidence for your answer (optional)

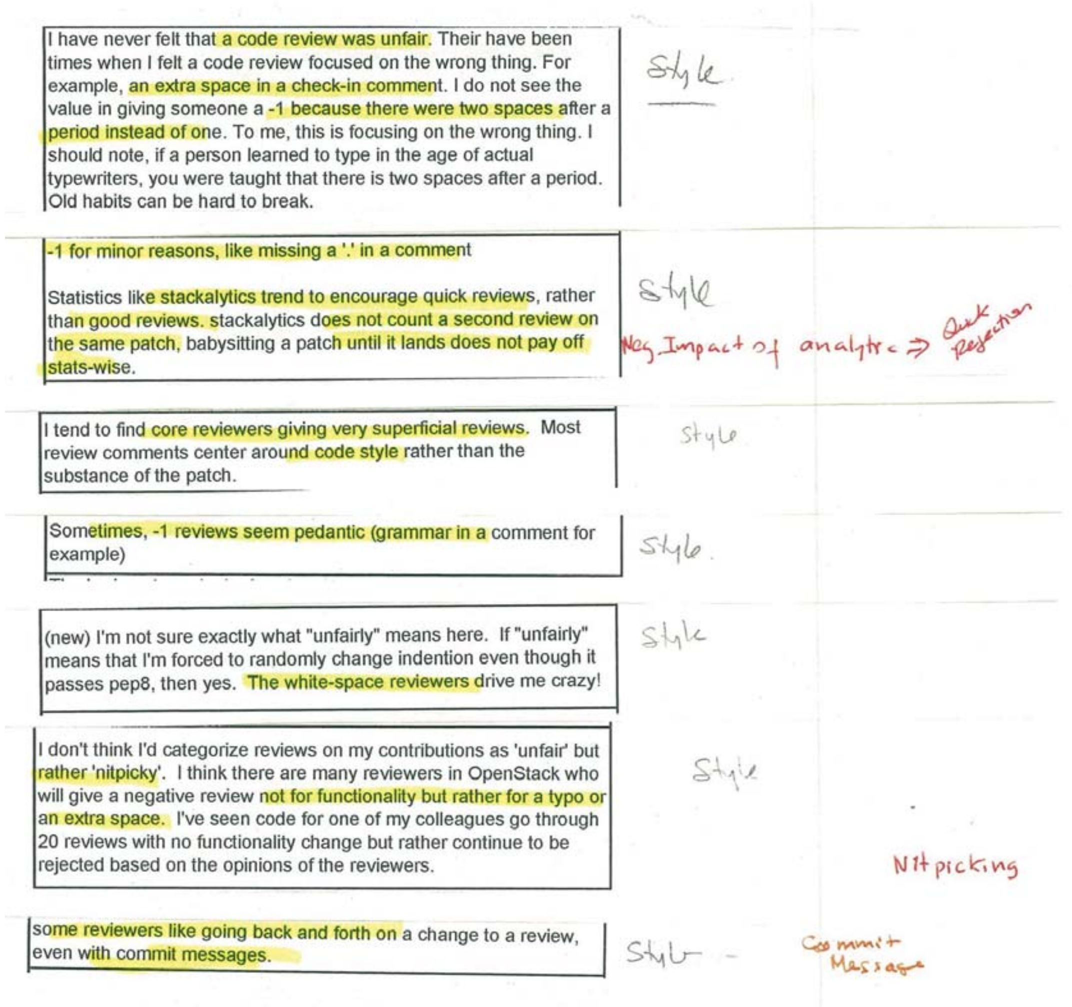
6. If you have interests in this survey or our research, If so, fill out your username, e-mail or openstack ID

Appendix C.

Manual Analysis of Survey

C.1. Examples of coding and memoing

The following images represent the open coding and memoing procedures. The response are listed on the left and we performed memoing on the right.



I have never felt that a code review was unfair. Their have been times when I felt a code review focused on the wrong thing. For example, an extra space in a check-in comment. I do not see the value in giving someone a -1 because there were two spaces after a period instead of one. To me, this is focusing on the wrong thing. I should note, if a person learned to type in the age of actual typewriters, you were taught that there is two spaces after a period. Old habits can be hard to break.

Style.

-1 for minor reasons, like missing a '.' in a comment

Statistics like stackalytics trend to encourage quick reviews, rather than good reviews. stackalytics does not count a second review on the same patch, babysitting a patch until it lands does not pay off stats-wise.

Style

Neg. Impact of analytic ⇒ Quick Rejection

I tend to find core reviewers giving very superficial reviews. Most review comments center around code style rather than the substance of the patch.

Style

Sometimes, -1 reviews seem pedantic (grammar in a comment for example)

Style.

(new) I'm not sure exactly what "unfairly" means here. If "unfairly" means that I'm forced to randomly change indention even though it passes pep8, then yes. The white-space reviewers drive me crazy!

Style

I don't think I'd categorize reviews on my contributions as 'unfair' but rather 'nitpicky'. I think there are many reviewers in OpenStack who will give a negative review not for functionality but rather for a typo or an extra space. I've seen code for one of my colleagues go through 20 reviews with no functionality change but rather continue to be rejected based on the opinions of the reviewers.

Style

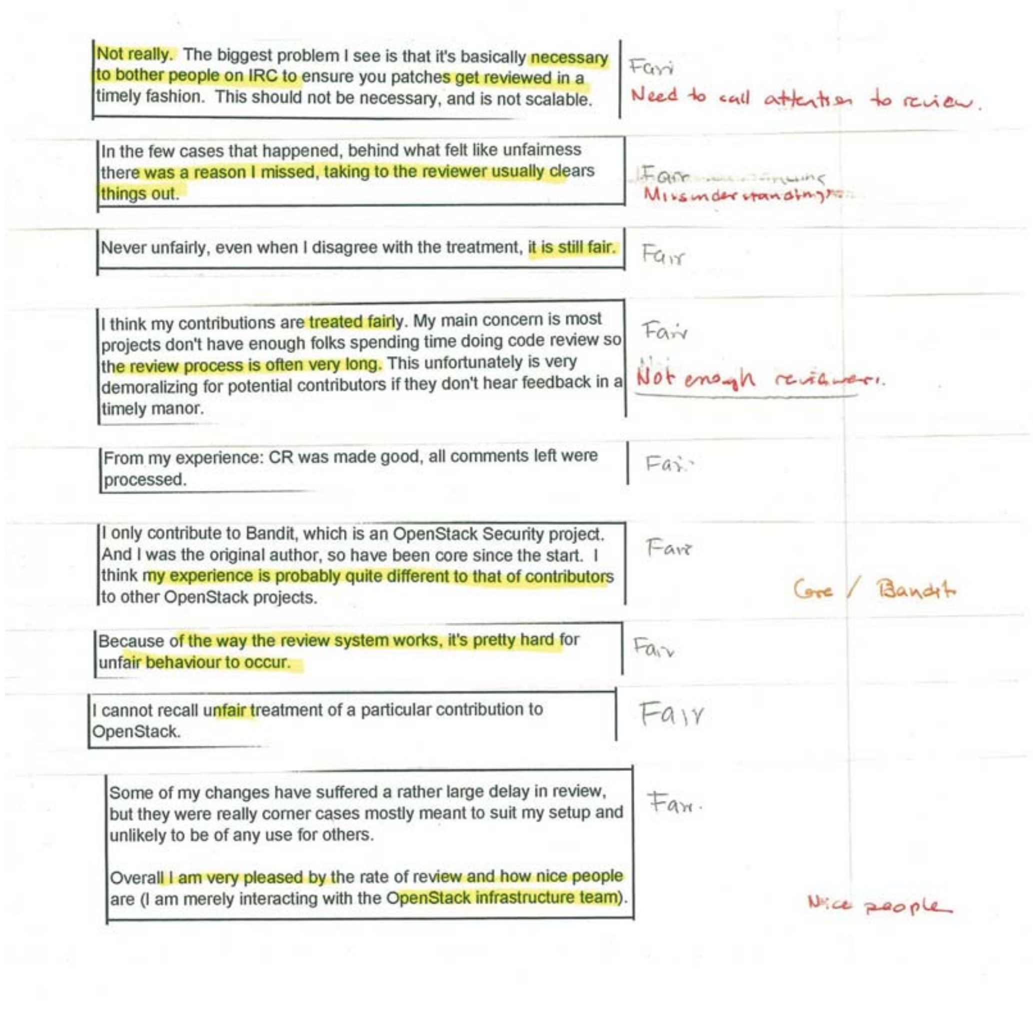
Nit picking

some reviewers like going back and forth on a change to a review, even with commit messages.

Style -

Commit Message

Figure C.1.: Survey responses and coding for filtering theme



Not really. The biggest problem I see is that it's basically necessary to bother people on IRC to ensure you patches get reviewed in a timely fashion. This should not be necessary, and is not scalable.

Fair
Need to call attention to review.

In the few cases that happened, behind what felt like unfairness there was a reason I missed, taking to the reviewer usually clears things out.

Fair
Misunderstandings

Never unfairly, even when I disagree with the treatment, it is still fair.

Fair

I think my contributions are treated fairly. My main concern is most projects don't have enough folks spending time doing code review so the review process is often very long. This unfortunately is very demoralizing for potential contributors if they don't hear feedback in a timely manor.

Fair
Not enough reviewers.

From my experience: CR was made good, all comments left were processed.

Fair

I only contribute to Bandit, which is an OpenStack Security project. And I was the original author, so have been core since the start. I think my experience is probably quite different to that of contributors to other OpenStack projects.

Fair
Core / Bandit

Because of the way the review system works, it's pretty hard for unfair behaviour to occur.

Fair

I cannot recall unfair treatment of a particular contribution to OpenStack.

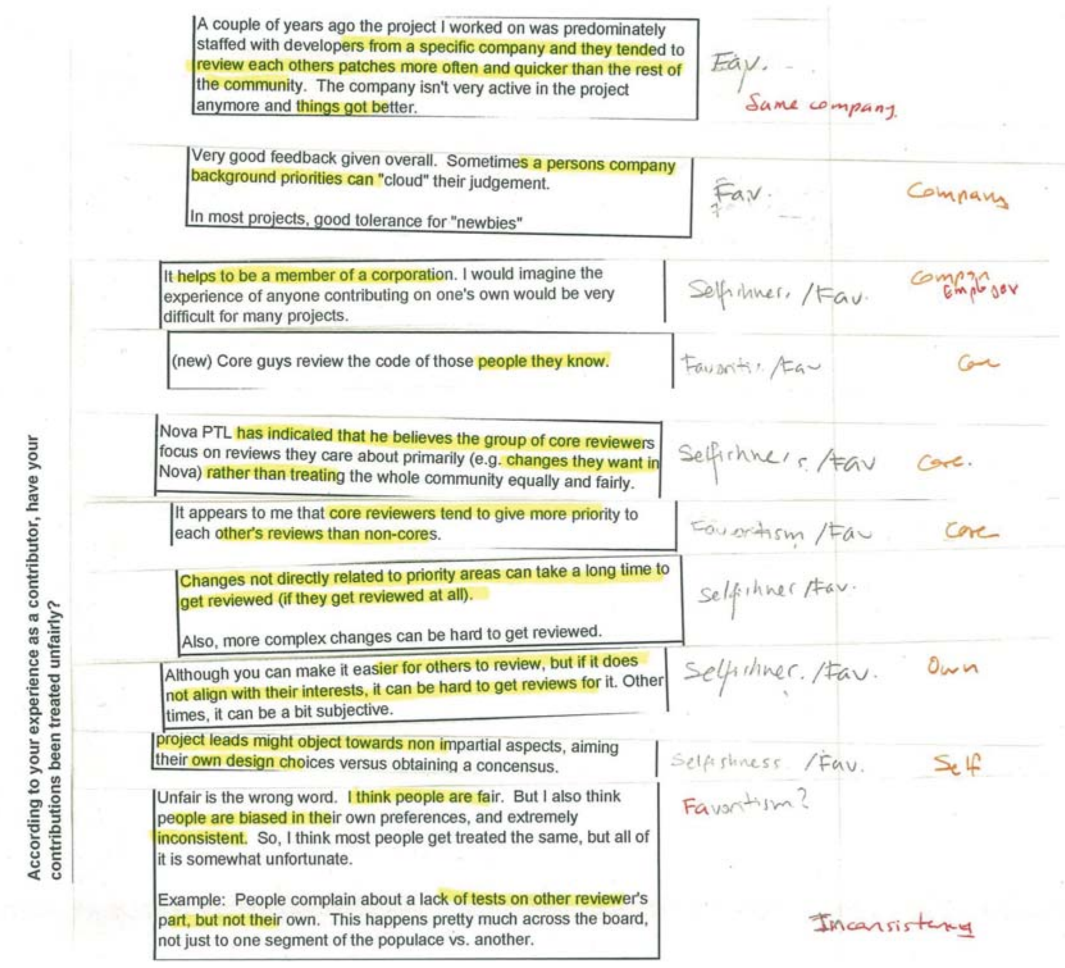
Fair

Some of my changes have suffered a rather large delay in review, but they were really corner cases mostly meant to suit my setup and unlikely to be of any use for others.

Overall I am very pleased by the rate of review and how nice people are (I am merely interacting with the OpenStack infrastructure team).

Fair.
Nice people

Figure C.2.: Survey responses and coding for filtering theme


According to your experience as a contributor, have your contributions been treated unfairly?

A couple of years ago the project I worked on was predominately staffed with developers from a specific company and they tended to review each others patches more often and quicker than the rest of the community. The company isn't very active in the project anymore and things got better.

Fav.

Same company

Very good feedback given overall. Sometimes a persons company background priorities can "cloud" their judgement.

In most projects, good tolerance for "newbies"

Fav.

Company

It helps to be a member of a corporation. I would imagine the experience of anyone contributing on one's own would be very difficult for many projects.

Selfishness /Fav.

Company Employee

(new) Core guys review the code of those people they know.

Favoritism /Fav

Core

Nova PTL has indicated that he believes the group of core reviewers focus on reviews they care about primarily (e.g. changes they want in Nova) rather than treating the whole community equally and fairly.

Selfishness /Fav

Core.

It appears to me that core reviewers tend to give more priority to each other's reviews than non-cores.

Favoritism /Fav

Core

Changes not directly related to priority areas can take a long time to get reviewed (if they get reviewed at all).

Also, more complex changes can be hard to get reviewed.

Selfishness /Fav.

Although you can make it easier for others to review, but if it does not align with their interests, it can be hard to get reviews for it. Other times, it can be a bit subjective.

Selfishness. /Fav.

Own

project leads might object towards non impartial aspects, aiming their own design choices versus obtaining a concensus.

Selfishness /Fav.

Self

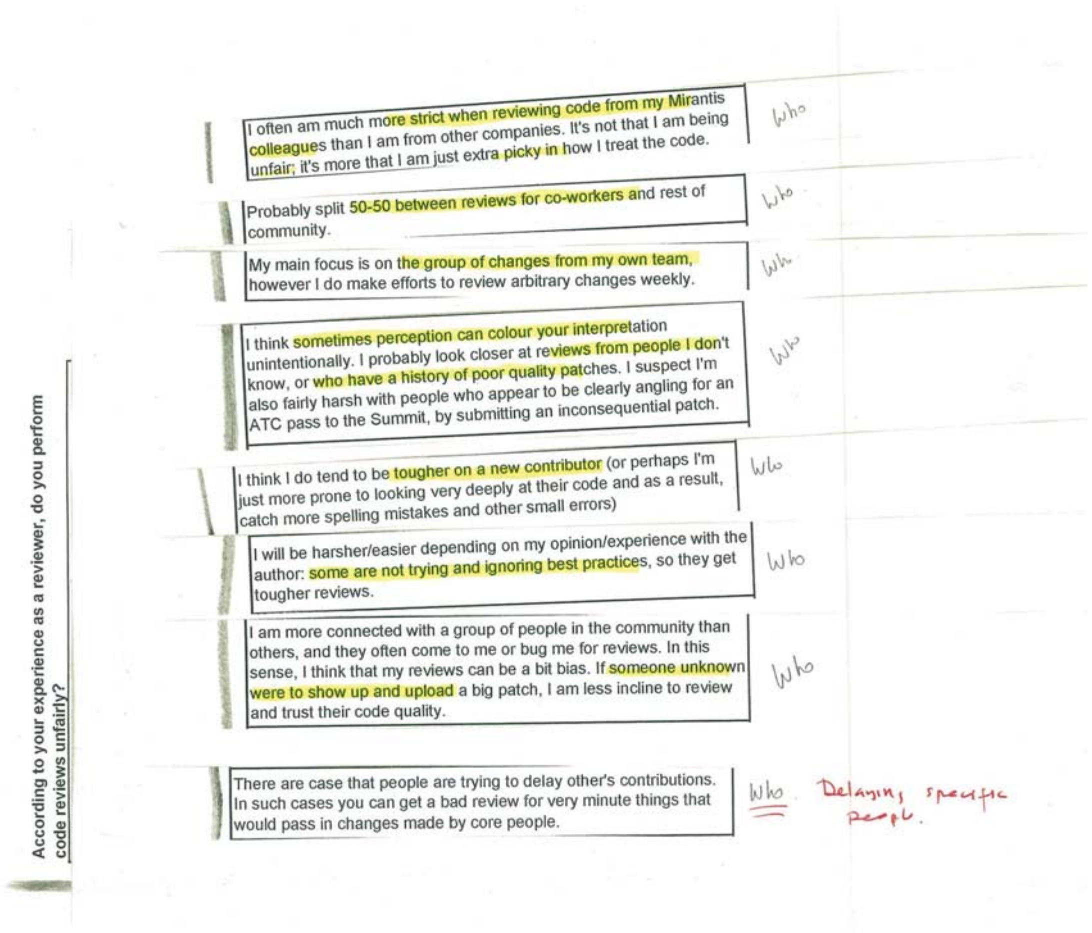
Unfair is the wrong word. I think people are fair. But I also think people are biased in their own preferences, and extremely inconsistent. So, I think most people get treated the same, but all of it is somewhat unfortunate.

Example: People complain about a lack of tests on other reviewer's part, but not their own. This happens pretty much across the board, not just to one segment of the populace vs. another.

Favoritism?

Inconsistency

Figure C.3.: Survey responses and coding for filtering theme



According to your experience as a reviewer, do you perform code reviews unfairly?

I often am much more strict when reviewing code from my Mirantis colleagues than I am from other companies. It's not that I am being unfair; it's more that I am just extra picky in how I treat the code.

Who

Probably split 50-50 between reviews for co-workers and rest of community.

Who

My main focus is on the group of changes from my own team, however I do make efforts to review arbitrary changes weekly.

Who

I think sometimes perception can colour your interpretation unintentionally. I probably look closer at reviews from people I don't know, or who have a history of poor quality patches. I suspect I'm also fairly harsh with people who appear to be clearly angling for an ATC pass to the Summit, by submitting an inconsequential patch.

Who

I think I do tend to be tougher on a new contributor (or perhaps I'm just more prone to looking very deeply at their code and as a result, catch more spelling mistakes and other small errors)

Who

I will be harsher/easier depending on my opinion/experience with the author: some are not trying and ignoring best practices, so they get tougher reviews.

Who

I am more connected with a group of people in the community than others, and they often come to me or bug me for reviews. In this sense, I think that my reviews can be a bit bias. If someone unknown were to show up and upload a big patch, I am less incline to review and trust their code quality.

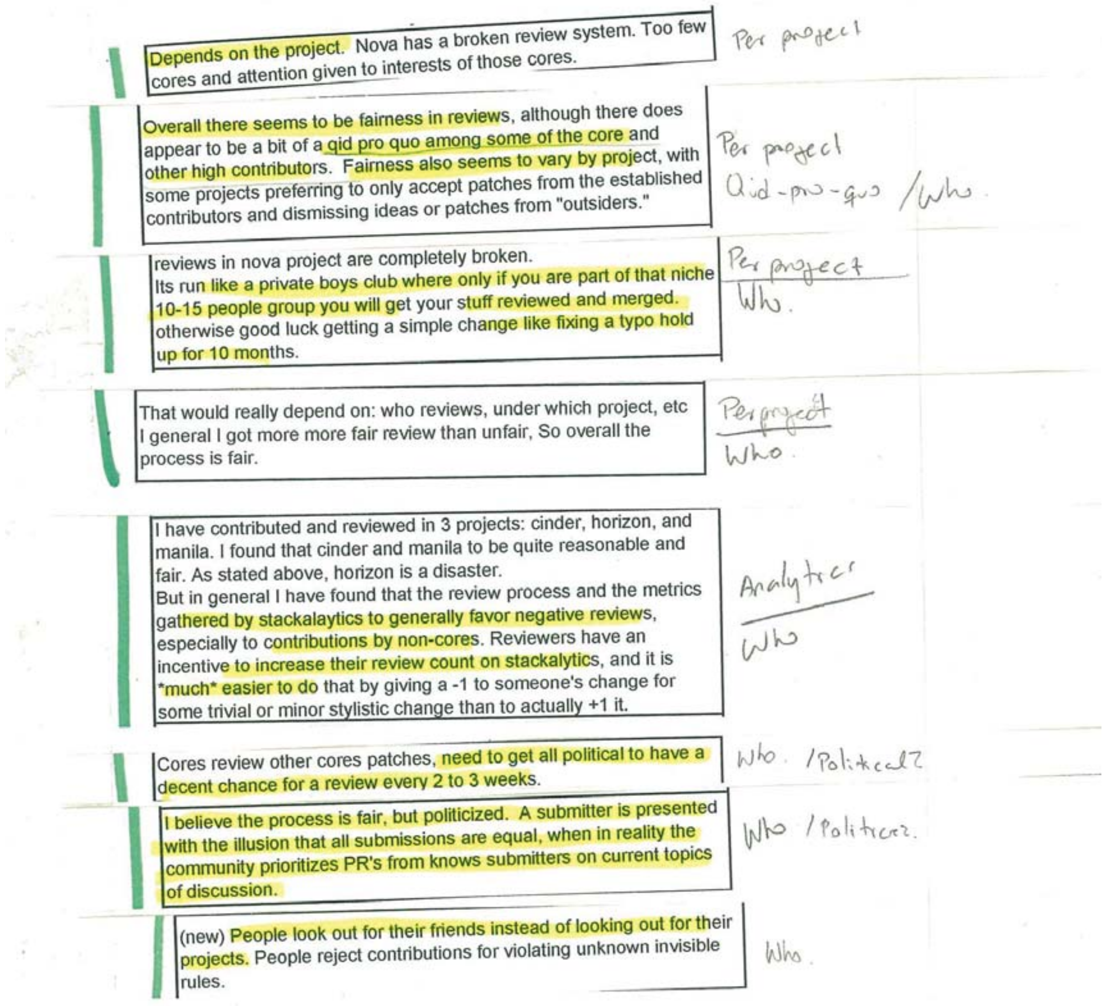
Who

There are case that people are trying to delay other's contributions. In such cases you can get a bad review for very minute things that would pass in changes made by core people.

Who

Delaying specific people.

Figure C.4.: Survey responses and coding for filtering theme



Depends on the project. Nova has a broken review system. Too few cores and attention given to interests of those cores.

Per project

Overall there seems to be fairness in reviews, although there does appear to be a bit of a qid pro quo among some of the core and other high contributors. Fairness also seems to vary by project, with some projects preferring to only accept patches from the established contributors and dismissing ideas or patches from "outsiders."

Per project
Qid-pro-quo /Who.

reviews in nova project are completely broken.
Its run like a private boys club where only if you are part of that niche 10-15 people group you will get your stuff reviewed and merged.
otherwise good luck getting a simple change like fixing a typo hold up for 10 months.

Per project
Who.

That would really depend on: who reviews, under which project, etc
I general I got more more fair review than unfair, So overall the process is fair.

~~Per project~~
Who.

I have contributed and reviewed in 3 projects: cinder, horizon, and manila. I found that cinder and manila to be quite reasonable and fair. As stated above, horizon is a disaster.
But in general I have found that the review process and the metrics gathered by stackalaytics to generally favor negative reviews, especially to contributions by non-cores. Reviewers have an incentive to increase their review count on stackalytics, and it is *much* easier to do that by giving a -1 to someone's change for some trivial or minor stylistic change than to actually +1 it.

~~Analytics~~
Who

Cores review other cores patches, need to get all political to have a decent chance for a review every 2 to 3 weeks.

Who. /Political?

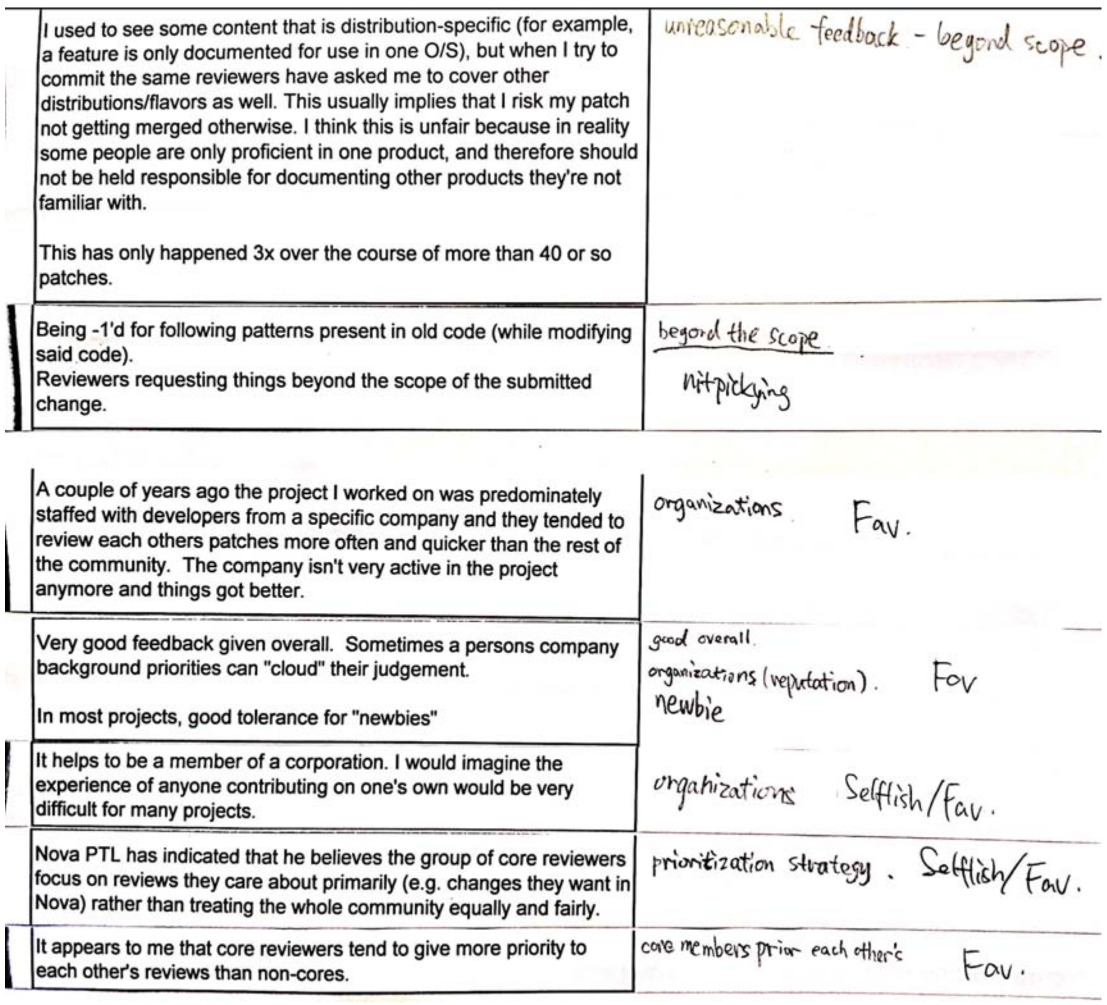
I believe the process is fair, but politicized. A submitter is presented with the illusion that all submissions are equal, when in reality the community prioritizes PR's from knows submitters on current topics of discussion.

Who /Political.

(new) People look out for their friends instead of looking out for their projects. People reject contributions for violating unknown invisible rules.

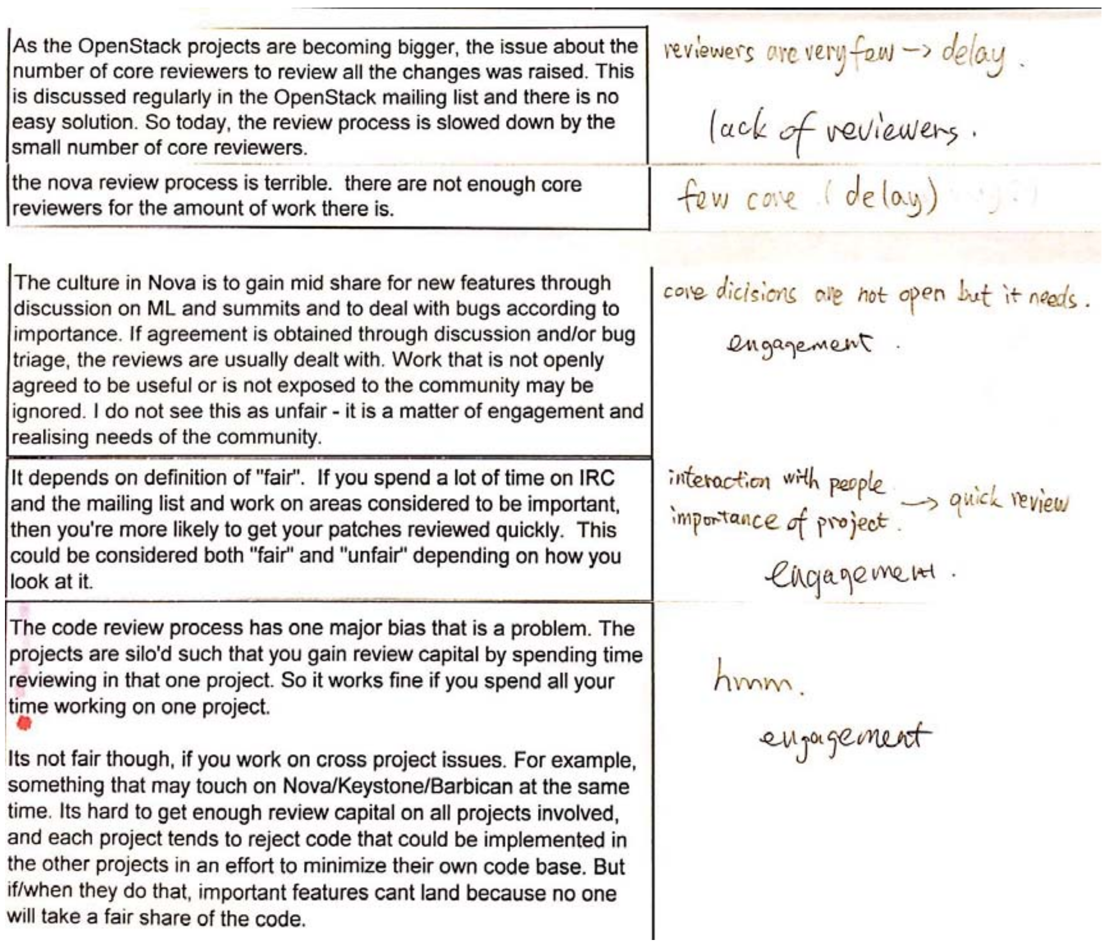
Who.

Figure C.5.: Survey responses and coding for filtering theme



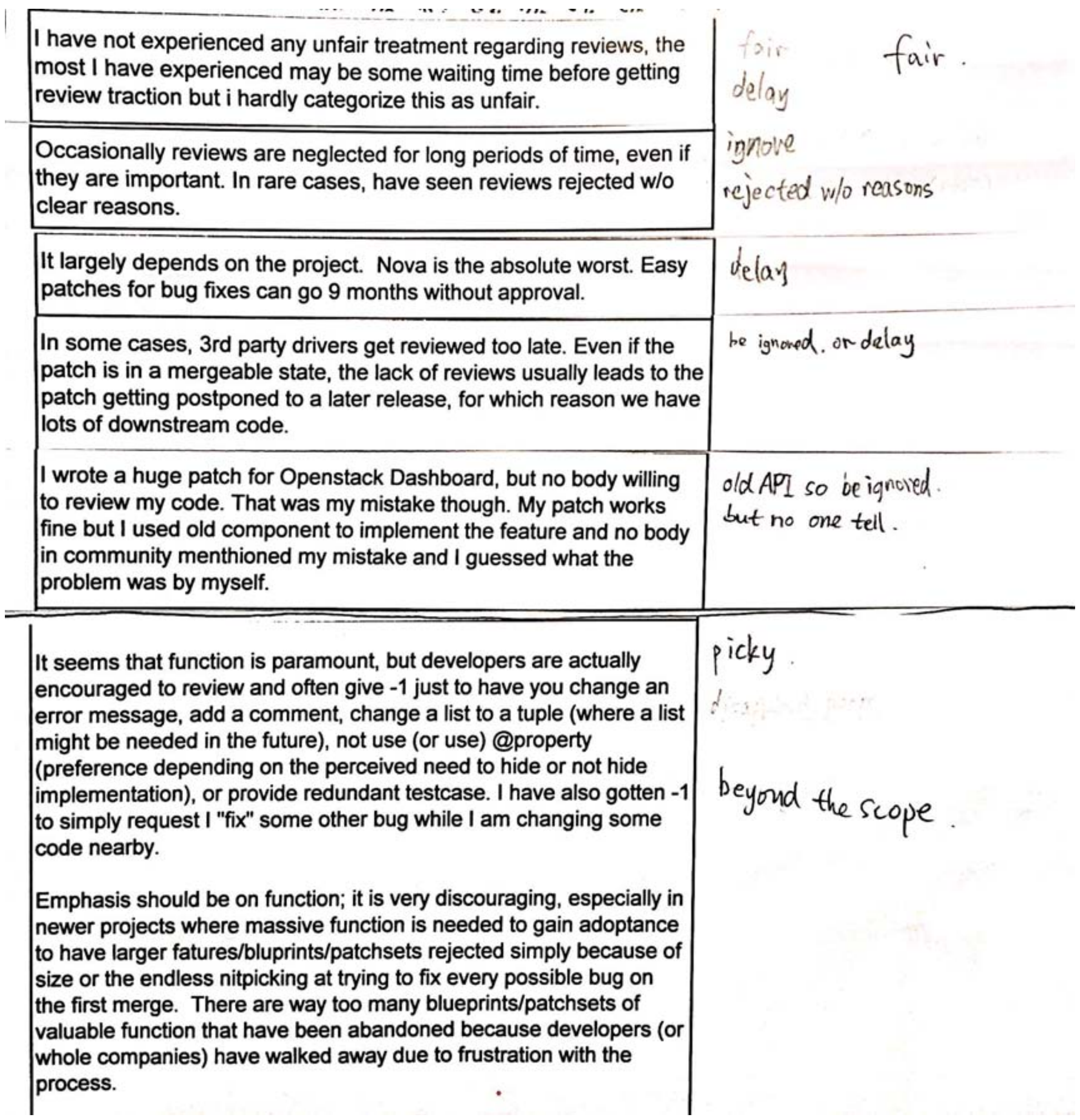
I used to see some content that is distribution-specific (for example, a feature is only documented for use in one O/S), but when I try to commit the same reviewers have asked me to cover other distributions/flavors as well. This usually implies that I risk my patch not getting merged otherwise. I think this is unfair because in reality some people are only proficient in one product, and therefore should not be held responsible for documenting other products they're not familiar with. This has only happened 3x over the course of more than 40 or so patches.	unreasonable feedback - beyond scope.
Being -1'd for following patterns present in old code (while modifying said code). Reviewers requesting things beyond the scope of the submitted change.	beyond the scope nitpickying
A couple of years ago the project I worked on was predominately staffed with developers from a specific company and they tended to review each others patches more often and quicker than the rest of the community. The company isn't very active in the project anymore and things got better.	organizations Fav.
Very good feedback given overall. Sometimes a persons company background priorities can "cloud" their judgement. In most projects, good tolerance for "newbies"	good overall. organizations (reputation). Fov newbie
It helps to be a member of a corporation. I would imagine the experience of anyone contributing on one's own would be very difficult for many projects.	organizations Selflish/Fav.
Nova PTL has indicated that he believes the group of core reviewers focus on reviews they care about primarily (e.g. changes they want in Nova) rather than treating the whole community equally and fairly.	prioritization strategy. Selflish/Fav.
It appears to me that core reviewers tend to give more priority to each other's reviews than non-cores.	core members prior each other's Fav.

Figure C.6.: Survey responses and coding for filtering theme



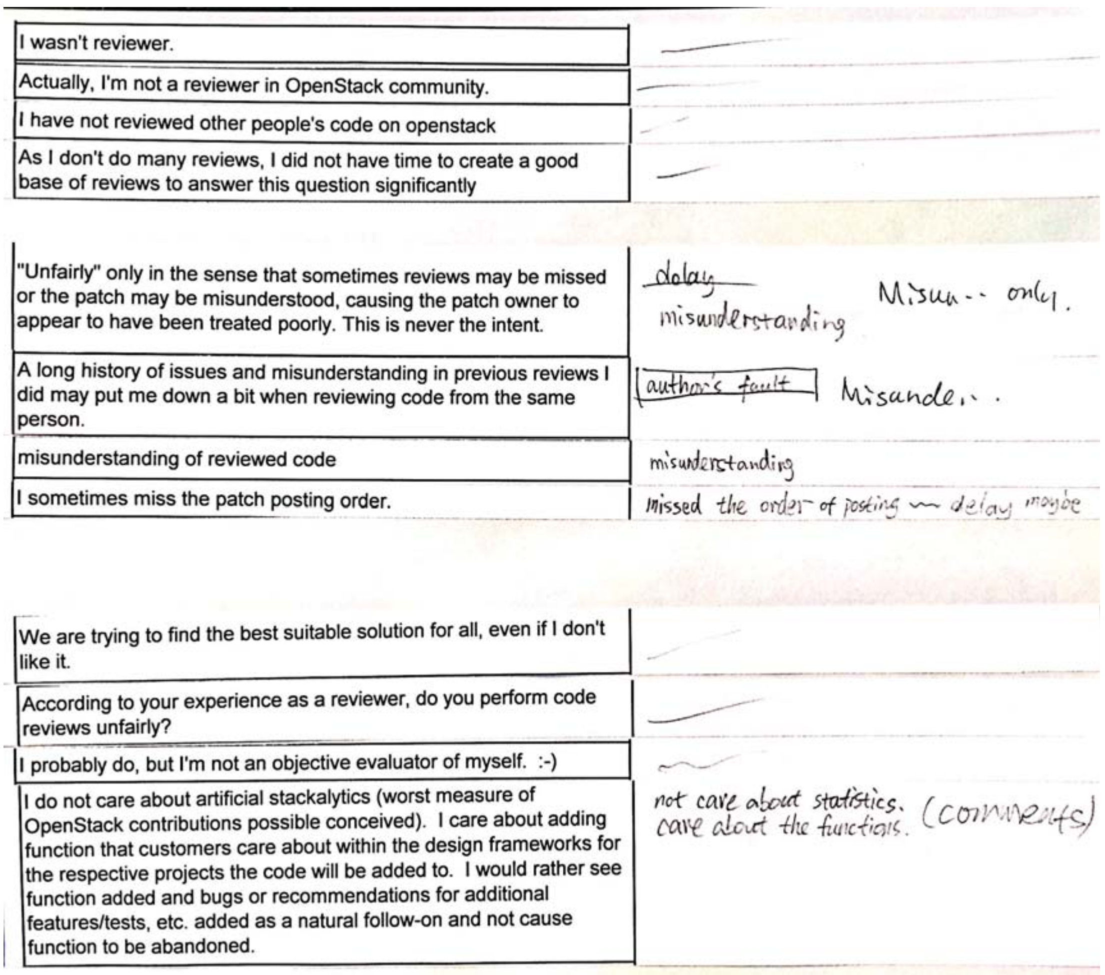
As the OpenStack projects are becoming bigger, the issue about the number of core reviewers to review all the changes was raised. This is discussed regularly in the OpenStack mailing list and there is no easy solution. So today, the review process is slowed down by the small number of core reviewers.	reviewers are very few → delay. lack of reviewers.
the nova review process is terrible. there are not enough core reviewers for the amount of work there is.	few core (delay)
The culture in Nova is to gain mid share for new features through discussion on ML and summits and to deal with bugs according to importance. If agreement is obtained through discussion and/or bug triage, the reviews are usually dealt with. Work that is not openly agreed to be useful or is not exposed to the community may be ignored. I do not see this as unfair - it is a matter of engagement and realising needs of the community.	core dicisions are not open but it needs. engagement.
It depends on definition of "fair". If you spend a lot of time on IRC and the mailing list and work on areas considered to be important, then you're more likely to get your patches reviewed quickly. This could be considered both "fair" and "unfair" depending on how you look at it.	interaction with people importance of project. → quick review engagement.
The code review process has one major bias that is a problem. The projects are silo'd such that you gain review capital by spending time reviewing in that one project. So it works fine if you spend all your time working on one project. Its not fair though, if you work on cross project issues. For example, something that may touch on Nova/Keystone/Barbican at the same time. Its hard to get enough review capital on all projects involved, and each project tends to reject code that could be implemented in the other projects in an effort to minimize their own code base. But if/when they do that, important features cant land because no one will take a fair share of the code.	hmm. engagement

Figure C.7.: Survey responses and coding for filtering theme



I have not experienced any unfair treatment regarding reviews, the most I have experienced may be some waiting time before getting review traction but i hardly categorize this as unfair.	fair fair. delay
Occasionally reviews are neglected for long periods of time, even if they are important. In rare cases, have seen reviews rejected w/o clear reasons.	ignore rejected w/o reasons
It largely depends on the project. Nova is the absolute worst. Easy patches for bug fixes can go 9 months without approval.	delay
In some cases, 3rd party drivers get reviewed too late. Even if the patch is in a mergeable state, the lack of reviews usually leads to the patch getting postponed to a later release, for which reason we have lots of downstream code.	be ignored. or delay
I wrote a huge patch for Openstack Dashboard, but no body willing to review my code. That was my mistake though. My patch works fine but I used old component to implement the feature and no body in community menthioned my mistake and I guessed what the problem was by myself.	old API so be ignored. but no one tell.
It seems that function is paramount, but developers are actually encouraged to review and often give -1 just to have you change an error message, add a comment, change a list to a tuple (where a list might be needed in the future), not use (or use) @property (preference depending on the perceived need to hide or not hide implementation), or provide redundant testcase. I have also gotten -1 to simply request I "fix" some other bug while I am changing some code nearby. Emphasis should be on function; it is very discouraging, especially in newer projects where massive function is needed to gain adoptance to have larger fatures/bluprints/patchsets rejected simply because of size or the endless nitpicking at trying to fix every possible bug on the first merge. There are way too many blueprints/patchsets of valuable function that have been abandoned because developers (or whole companies) have walked away due to frustration with the process.	picky. [illegible] beyond the scope.

Figure C.8.: Survey responses and coding for filtering theme



I wasn't reviewer.	—
Actually, I'm not a reviewer in OpenStack community.	—
I have not reviewed other people's code on openstack	—
As I don't do many reviews, I did not have time to create a good base of reviews to answer this question significantly	—

"Unfairly" only in the sense that sometimes reviews may be missed or the patch may be misunderstood, causing the patch owner to appear to have been treated poorly. This is never the intent.	delay misunderstanding Misun.. only.
A long history of issues and misunderstanding in previous reviews I did may put me down a bit when reviewing code from the same person.	$\boxed{\text{author's fault}}$ Misande...
misunderstanding of reviewed code	misunderstanding
I sometimes miss the patch posting order.	missed the order of posting ~ delay maybe

We are trying to find the best suitable solution for all, even if I don't like it.	—
According to your experience as a reviewer, do you perform code reviews unfairly?	—
I probably do, but I'm not an objective evaluator of myself. :-)	~
I do not care about artificial stackalytics (worst measure of OpenStack contributions possible conceived). I care about adding function that customers care about within the design frameworks for the respective projects the code will be added to. I would rather see function added and bugs or recommendations for additional features/tests, etc. added as a natural follow-on and not cause function to be abandoned.	not care about statistics. care about the functions. (comments)

Figure C.9.: Survey responses and coding for filtering theme

Major Publication List

1. **Xin Yang**, Norihiro Yoshida, Raula Gaikovina Kula and Hajimu Iida, Peer Review Social Network (PeRSoN) in Open Source Projects, *IEICE Transactions on Information and Systems, volume E99-D, number 3, pages 661-670*, 2016.

2. **Xin Yang**, Raula Gaikovina Kula, Norihiro Yoshida and Hajimu Iida, Mining the Modern Code Review Repositories: A Dataset of People, Process and Product, *In Proceedings of the 13th Working Conference on Mining Software Repositories, MSR 2016 (Austin, TX, US)*, pages 460-463, May 2016.

3. **Xin Yang**, Social Network Analysis in Open Source Software Peer Review, *In Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2014 (Hong Kong, China)*, pages 820-822, 2014.

4. **Xin Yang**, Norihiro Yoshida, Kenji Fujiwara, Yong Jin and Hajimu Iida, Categorizing Code Review Result with Social Networks Analysis: A Case Study on Three OSS Projects, *In Proceedings of IPSJ/SIGSE Software Engineering Symposium 2014, SES 2014 (Tokyo, Japan)*, pages, 200-201, 2014.

5. **Xin Yang**, Raula Gaikovina Kula, Camargo Cruz Ana Erika, Norihiro Yoshida, Kazuki Hamasaki, Kenji Fujiwara, and Hajimu Iida, Understanding OSS Peer Review Roles in Peer Review Social Network (PeRSoN), *In Proceedings of the 19th Asia-Pacific Software Engineering Conference, APSEC 2012 (Hong Kong, China)*, pages 709-712, 2012.

Related Publication List

1. **Xin Yang**, Kar-Long Chan, Papon Yongpisanpop, Hideaki Hata, Hajimu Iida and Kenichi Matsumoto, Human Software Interaction in Software Development Community, *In Proceedings of Winter Workshop 2015 in Ginowan, WWS 2015 (Okinawa, Japan)*, pages 5-6, 2015.

2. Thunyathon Jaruchotrattanasakul, **Xin Yang**, Erina Makihara, Kenji Fujiwara and Hajimu Iida, Open Source Resume (OSR): A Visualization Tool for Presenting OSS Biographies of Developers, *In the 7th International Workshop on Empirical Software Engineering in Practice, IWESEP 2016 (Osaka, Japan)*, pages 57-62, March 2016.

3. Yong Jin, **Xin Yang**, Raula Gaikovina Kula, Eunjong Choi, Hajimu Iida and Katsuro Inoue, Quick Trigger on Stack Overflow: A Study of Gamification-Influenced Member Tendencies, *In Proceedings of the 12th Working Conference on Mining Software Repositories, MSR 2015 (Florence, Italy)*, pages 434-437, May 2015.

4. Patanamon Thongtanunam, **Xin Yang**, Norihiro Yoshida, Kenji Fujiwara, Raula Gaikovina Kula, Ana Erika Camargo Cruz, Hajimu Iida, ReDA: A Web-based Visualization Tool for Analyzing Modern Code Review Dataset, *In proceeding of the 30th International Conference on Software Maintenance and Evolution, ICSME 2014 (Victoria, BC, Canada*, pages 605-608, 2014.

5. Raula Gaikovina Kula, Ana Erika Camargo Cruz, Norihiro Yoshida, Kazuki Hamasaki, Kenji Fujiwara, **Xin Yang**, Hajimu Iida: Using Profiling Metrics to Categorise Peer Review Types in the Android Project, *In supplemental proceedings of the IEEE 23rd International Symposium on Software Reliability Engineering, ISSRE 2012 (Dallas, TX, US)*, pages 146-151, 2012.

References

[1] `https://android-review.googlesource.com` (Feb 18, 2016).

[2] `https://git.eclipse.org/r/` (Feb 18, 2016).

[3] `https://gerrit.libreoffice.org` (Feb 18, 2016).

[4] `https://review.openstack.org` (Feb 18, 2016).

[5] `https://codereview.qt-project.org` (Feb 18, 2016).

[6] M. Aberdour. Achieving quality in open-source software. *Software, IEEE*, pages 58–64, 2007.

[7] A. Frank Ackerman, Priscilla J. Fowler, and Robert G. Ebenau. Software inspections and the industrial production of software. In *Proc. of a symposium on Software validation: inspection-testing-verification-alternatives*, pages 13–40, 1984.

[8] A.F. Ackerman, L.S. Buchwald, and F.H. Lewski. Software inspections: an effective verification process. *Software, IEEE*, 6:31–36, 1989.

[9] Benjamin Avi-Itzhak, Hanoch Levy, and David Raz. A resource allocation queueing fairness measure: Properties and bounds. *Queueing Systems*, 56(2):65–71, 2007.

[10] Benjamin Avi-Itzhak, Hanoch Levy, and David Raz. Quantifying Fairness in Queuing Systems. *Probability in the Engineering and Informational Sciences*, 22(04):495–517, 2008.

[11] Alberto Bacchelli and Christian Bird. Expectations, outcomes, and challenges of modern code review. In *Proc. ICSE '13*, pages 712–721, 2013.

[12] Vipin Balachandran. Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation. In *Proc. of ICSE 2013*, pages 931–940, 2013.

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

[14] Olga Baysal, Oleksii Kononenko, Reid Holmes, and Michael W. Godfrey. Investigating technical and non-technical factors influencing modern code review. *Empirical Software Engineering (EMSE)*, pages 1–28, 2015.

[15] Christian Bird and Nachiappan Nagappan. Who? where? what?: examining distributed development in two large open source projects. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, pages 237–246. IEEE Press, 2012.

[16] Christian Bird, David Pattison, Raissa D'Souza, Vladimir Filkov, and Premkumar Devanbu. Latent social structure in open source projects. In *Proc. of FSE 2008*, pages 24–35, 2008.

[17] Jeffrey G Blodgett, Donna J Hill, and Stephen S Tax. The effects of distributive, procedural, and interactional justice on postcomplaint behavior. *Journal of retailing*, 73(2):185–210, 1997.

[18] Amiangshu Bosu and Jeffrey C. Carver. Impact of developer reputation on code review outcomes in oss projects: An empirical investigation. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '14, pages 33:1–33:10, 2014.

[19] Amiangshu Bosu, Michaela Greiler, and Christian Bird. Characteristics of useful code reviews: An empirical study at microsoft. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, pages 146–156, 2015.

[20] Silvia Breu, Rahul Premraj, Jonathan Sillito, and Thomas Zimmermann. Information needs in bug reports: improving cooperation between develop-

ers and users. In *Proceedings of the 2010 ACM conference on Computer supported cooperative work*, pages 301–310, 2010.

[21] Ronald L Cohen. Distributive justice: Theory and research. *Social Justice Research*, 1(1):19–40, 1987.

[22] Jason A Colquitt. On the dimensionality of organizational justice: a construct validation of a measure. *Journal of applied psychology*, 86(3):386, 2001.

[23] Juliet M Corbin. *Grounded theory in practice.* Sage, 1997.

[24] Kevin Crowston and James Howison. The social structure of free and open source software development. *First Monday*, 10(2), 2005.

[25] Daniela E Damian and Didar Zowghi. An insight into the interplay between culture, conflict and distance in globally distributed requirements negotiations. In *System Sciences, 2003. Proceedings of the 36th Annual Hawaii International Conference on*, pages 10–pp. IEEE, 2003.

[26] Michael Fagan. Advances in software inspections. *IEEE Transactions on Software Engineering*, 12(7):744–751, 1986.

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

[28] Brian Fitzgerald. The transformation of open source software. *Mis Quarterly*, pages 587–598, 2006.

[29] Linton C. Freeman. Centrality in social networks conceptual clarification. *Social Networks*, 1(3):215–239, 1978-1979.

[30] S. S. Gokhale and R. E. Mullen. Queuing models for field defect resolution process. In *Software Reliability Engineering, 2006. ISSRE'06. 17th International Symposium On*, pages 353–362, Nov 2006.

[31] Jesús M. González-Barahona, Daniel Izquierdo-Cortázar, Gregorio Robles, and Mario Gallegos. Code review analytics: WebKit as case study. In

Luis Corral, Alberto Sillitti, Giancarlo Succi, Jelena Vlasenko, and AnthonyI. Wasserman, editors, *Open Source Software: Mobile Open Source Technologies*, volume 427 of *IFIP Advances in Information and Communication Technology*, pages 1–10. Springer Berlin Heidelberg, 2014.

[32] Jesus M. Gonzalez-Barahona, Gregorio Robles, and Daniel Izquierdo-Cortazar. The MetricsGrimoire Database Collection. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 478–481. IEEE, may 2015.

[33] Georgios Gousios, Andy Zaidman, Margaret-anne Storey, and Arie Van Deursen. Work Practices and Challenges in Pull-Based Development: The Integrator's Perspective. In *37th International Conference on Software Engineering*, Florence, Italy, 2015.

[34] Jerald Greenberg. A taxonomy of organizational justice theories. *Academy of Management review*, 12(1):9–22, 1987.

[35] Kazuki Hamasaki, Raula Gaikovina Kula, Norihiro Yoshida, A. E. Camargo Cruz, Kenji Fujiwara, and Hajimu Iida. Who does what during a code review? Datasets of OSS peer review repositories. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 49–52. IEEE, may 2013.

[36] Rattikorn Hewett and Phongphun Kijsanayothin. On modeling software defect repair time. *Empirical Software Engineering*, 14(2):165–186, 2009.

[37] Gaeul Jeong, Sunghun Kim, Thomas Zimmermann, and Kwangkeun Yi. Improving code review by predicting reviewers and acceptance of patches. *Research on Software Analysis for Error-free Computing Center Tech-Memo (ROSAEC MEMO 2009-006)*, 2009.

[38] Yujuan Jiang, Bram Adams, and Daniel M. German. Will My Patch Make It? And How Fast? Case Study on the Linux Kernel. In *Proceeding of the 10th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 101–110, 2013.

[39] Karen L Katz, Blaire M Larson, and Richard C Larson. Prescription for the waiting-in-line blues: Entertain, enlighten, and engage. *MIT Sloan Management Review*, 32(2):44, 1991.

[40] Oleksii Kononenko, Olga Baysal, and Michael W Godfrey. Code review quality: How developers see it. In *Proc. of ICSE '16*, page to appear, 2016.

[41] Helena Chmura Kraemer, George A. Morgan, Nancy L. Leech, Jeffrey A. Gliner, Jerry J. Vaske, and Robert J. Harmon. Measures of Clinical Significance. *Journal of the American Academy of Child & Adolescent Psychiatry*, 42(12):1534–1529, 2003.

[42] Raula Gaikovina Kula, Ana Erika Camargo Cruz, Norihiro Yoshida, Kazuki Hamasaki, Koji Fujiwara, Xu Yang, and Hiroyuki Iida. Using profiling metrics to categorise peer review types in the android project. In *Proc. of ISSRE 2012*, pages 146–151, 2012.

[43] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is twitter, a social network or a news media? In *Proceedings of the 19th International Conference on World Wide Web*, pages 591–600, 2010.

[44] Gerald S Leventhal. The distribution of rewards and resources in groups and organizations. *Advances in experimental social psychology*, 9:91–131, 1976.

[45] Gerald S Leventhal. *What should be done with equity theory?* Springer, 1980.

[46] Thomas R. Lindlof and Bryan C. Taylor. *Qualitative communication research methods*. SAGE Publications, 2nd edition, 2002.

[47] Guillermo Macbeth, Eugenia Razumiejczyk, and Rub{\'e}n Daniel Ledesma. Cliff's Delta Calculator: A Non-parametric Effect Size Program for Two Groups of Observations. *Universitas Psychologica*, 10:545–555, 2011.

[48] Christina Maslach and Michael P Leiter. Early predictors of job burnout and engagement. *Journal of applied psychology*, 93(3):498, 2008.

[49] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassa. An Empirical Study of the Impact of Modern Code Review Practices on Software Quality. *Empirical Software Engineering (EMSE)*, pages 1–44, 2015.

[50] 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 *Proc. of MSR '14*, pages 192–201, 2014.

[51] Andrew Meneely, Laurie Williams, Will Snipes, and Jason Osborne. Predicting failures with developer networks and social network analysis. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '08/FSE-16, pages 13–23, New York, NY, USA, 2008. ACM.

[52] Audris Mockus, Roy T. Fielding, and James Herbsleb. A case study of open source software development: the apache server. In *Proc. of ICSE 2000*, pages 263–272, 2000.

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

[54] Murtuza Mukadam, Christian Bird, and Peter C. Rigby. Gerrit software code review data from android. In *Proc. of MSR '13*, pages 45–48, 2013.

[55] Mark Newman. *Networks: An Introduction.* Oxford University Press, Inc., 2010.

[56] Peter Axel Nielsen and Gitte Tjørnehøj. Social networks in software process improvement. *Software Process: Improvement and Practice*, 2009.

[57] David L. Parnas and Mark Lawford. The role of inspection in software quality assurance. *IEEE Trans. Softw. Eng.*, 29(8):674–676, 2003.

[58] Anat Rafaeli, Greg Barron, and Keren Haber. The Effects of Queue Structure on Attitudes. *Journal of Service Research*, 5(2):125–139, 2002.

[59] Eric S. Raymond. *The Cathedral and the Bazaar.* O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1st edition, 1999.

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

[61] Peter C. Rigby, Daniel M. German, Laura Cowen, and Margaret-Anne Storey. Peer Review on Open-Source Software Projects. *ACM Transactions on Software Engineering and Methodology*, 23(4):1–33, 2014.

[62] Peter C Rigby, Daniel M German, and Margaret-Anne Storey. Open source software peer review practices: a case study of the apache server. In *Proceedings of the 30th international conference on Software engineering*, pages 541–550. ACM, 2008.

[63] Peter C Rigby and Margaret-Anne Storey. Understanding broadcast based peer review on open source software projects. In *Proc. of ICSE 2011*, pages 541–550, 2011.

[64] Jason Robbins. Adopting open source software engineering (osse) practices by adopting osse tools. *Perspectives on free and open source software*, pages 245–264, 2005.

[65] Giedre Sabaliauskaite, Fumikazu Matsukawa, Shinji Kusumoto, and Katsuro Inoue. Further investigations of reading techniques for object-oriented design inspection. *Information and Software Technology*, 45(9):571–585, 2003.

[66] Kathleen Seiders and Leonard L Berry. Service fairness: What it is and why it matters. *The Academy of Management Executive*, 12(2):8–20, 1998.

[67] Birud Sindhav, Jonna Holland, Amy Risch Rodie, Phani Tej Adidam, and Louis G Pol. The impact of perceived fairness on satisfaction: are airport security measures fair? does it matter? *Journal of Marketing Theory and Practice*, 14(4):323–335, 2006.

[68] Jose Teixeira, Gregorio Robles, and Jesús M González-Barahona. Lessons learned from applying social network analysis on an industrial free/libre/open source software ecosystem. *Journal of Internet Services and Applications*, 6(1):1–27, 2015.

[69] John W Thibaut and Laurens Walker. *Procedural justice: A psychological analysis.* L. Erlbaum Associates, 1975.

[70] P. Thongtanunam, C. Tantithamthavorn, R.G. Kula, N. Yoshida, H. Iida, and K. Matsumoto. Who should review my code? a file location-based code-reviewer recommendation approach for modern code review. In *Proc. of SANER 2015*, 2015.

[71] Patanamon Thongtanunam, Shane McIntosh, Ahmed E. Hassan, and Hajimu Iida. Investigating code review practices in defective files: An empirical study of the qt system. In *Proc of MSR '15*, pages 168–179, 2015.

[72] Patanamon Thongtanunam, Shane McIntosh, Ahmed E. Hassan, and Hajimu Iida. Revisiting Code Ownership and its Relationship with Software Quality in the Scope of Modern Code Review. In *Proceedings of the 38th international conference on Software engineering - ICSE '16*, pages 1039–1050, 2016.

[73] Patanamon Thongtanunam, Xin Yang, Norihiro Yoshida, Raula Gaikovina Kula, Ana Erika Camargo Cruz, Koji Fujiwara, and Hiroyuki Iida. Reda: A web-based visualization tool for analyzing modern code review dataset. In *Proc. of ICSME 2014*, pages 605–608, 2014.

[74] Koji Toda, Yasutaka Kamei, Kazuki Hamasaki, and Norihiro Yoshida. Effect of review and patch development experience in the Chromium project's patch review time. *Computer Software*, 32(1):227–233, 2015.

[75] Parastou Tourani and Bram Adams. The impact of human discussions on just-in-time quality assurance. In *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering - SANER '16*, 2016.

[76] Guilherme Travassos, Forrest Shull, Michael Fredericks, and Victor R. Basili. Detecting defects in object-oriented designs: Using reading techniques to increase software quality. In *Proc. of OOPSLA '99*, pages 47–56, 1999.

[77] Jason Tsay, Laura Dabbish, and James Herbsleb. Let's talk about it: evaluating contributions through discussion in GitHub. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 144–154. ACM, 2014.

[78] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. When and why your code starts to smell bad. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 403–414, Piscataway, NJ, USA, 2015. IEEE Press.

[79] Tom Tyler, Jonathan Jackson, and Ben Bradford. Psychology of procedural justice and cooperation. *ENCYCLOPEDIA OF CRIMINOLOGY AND CRIMINAL JUSTICE, edited by G. Bruinsma and D. Weisburd, Springer-Verlag, Forthcoming*, 2013.

[80] Erik Van Der Veen, Georgios Gousios, and Andy Zaidman. Automatically Prioritizing Pull Requests. In *12th International Conference on Mining Software Repositories*, Florence, Italy, 2015.

[81] Clay M. Voorhees, Julie Baker, Brian L. Bourdeau, E. Deanne Brocato, and J. Joseph Cronin. It Depends: Moderating the Relationships Among Perceived Waiting Time, Anger, and Regret, 2009.

[82] David A Wheeler, Bill Brykczynski, and Reginald N Meeson Jr. *Software Inspection: An Industry Best Practice for Defect Detection and Removal.* IEEE Computer Society Press, 1996.

[83] Douglas A Wolfe and Myles Hollander. Nonparametric statistical methods. *Nonparametric statistical methods*, 1973.

[84] Xin Yang. Social network analysis in open source software peer review. In *Proc. of FSE 2014*, pages 820–822, 2014.

[85] Xin Yang, Raula Gaikovina Kula, Camargo Cruz Ana Erika, Norihiro Yoshida, Kazuki Hamasaki, Kenji Fujiwara, and Hajimu Iida. Understanding oss peer review roles in peer review social network (PeRSoN). In *Proceedings of the 19th Asia-Pacific Software Engineering Conference - APSEC '12*, volume 1, pages 709–712. IEEE, 2012.

[86] Xin Yang, Raula Gaikovina Kula, Norihiro Yoshida, and Hajimu Iida. Mining the Modern Code Review Repositories: A Dataset of People, Process and Product. In *Proceedings of the 13th International Conference on Mining Software Repositories*, pages 460–463, 2016.

[87] Xin Yang, Norihiro Yoshida, Raula Gaikovina Kula, and Hajimu Iida. Peer review social network (PeRSoN) in open source projects. *IEICE Transactions on Information and Systems*, E99-D(3):661–670, 2016.

[88] Marcelo Serrano Zanetti, Ingo Scholtes, Claudio Juan Tessone, and Frank Schweitzer. Categorizing bugs with social networks: a case study on four open source software communities. In *Proc. of ICSE 2013*, pages 1032–1041, 2013.