The Pennsylvania State University
The Graduate School

College of Information Sciences and Technology

BEHIND LINUS’S LAW:
INVESTIGATING PEER REVIEW PROCESSES IN OPEN SOURCE

A Dissertation in
Information Sciences and Technology

By
Jing Wang

© 2013 Jing Wang

Submitted in Partial Fulfillment
of the Requirements
for the Degree of
Doctor of Philosophy

December 2013
The dissertation of Jing Wang was reviewed and approved* by the following:

John M. Carroll  
Professor of Information Sciences and Technology  
Dissertation Advisor  
Chair of Committee

Andrea Tapia  
Associate Professor of Information Sciences and Technology

Rosalie Ocker  
Senior Lecturer of Information Sciences and Technology

Robert Macy  
Clinical Associate Professor of Management and Organization  
Director of Farrell Center for Corporate Innovation

Peter Forster  
Assistant Dean for Online Programs and Professional Education  
Graduate Programs Director  
College of Information Sciences and Technology

*Signatures are on file in the Graduate School
ABSTRACT

Open source software has revolutionized the way people develop software, organize collaborative work, and innovate. The numerous open source software systems that have been created and adopted over the past decade are influential and vital in all aspects of work and daily life. The understanding of open source software development can enhance its performance and product quality, and inform other types of software development collaboration and virtual organizations.

Linus’s law, “given enough eyeballs, all bugs are shallow”, underlines the power of the crowd in open source software peer review. Despite its importance, Linus’s law has surprisingly received very limited examination and elaboration. The peer review process, in which peers evaluate software products, finding, analyzing, and fixing defects and deficiencies, remain largely unclear.

The research this dissertation presents is to enhance the understanding of open source software peer review, contributing both a descriptive characterization and design recommendations of socio-technical support for the collaborative process. The empirical investigation involves comparative case studies on two established and well-recognized open source software communities, Mozilla and Python. Extensive analyses on bug reports archived in bug tracking systems, the central tools for open source software peer review, unfold the common process of open source peer review as well as the variations of collaboration between different types of projects. These findings indicate technology and organizational designs that may support those commonalities and differences. Situating the two cases in the evolution of open source software development, this research also articulates the differences in comparison to prior software review approaches as well as the implications for the emerged new ways of organizing open source software projects.
# TABLE OF CONTENTS

LIST OF FIGURES ........................................................................................................... vii
LIST OF TABLES ............................................................................................................... viii
ACKNOWLEDGEMENTS ................................................................................................. ix

Chapter 1  Introduction ................................................................................................. 1
  Research Statement and Scope ..................................................................................... 3
  Organization of the Dissertation ................................................................................... 5

Chapter 2  Overview of Open Source Software Development and Software Peer Review .... 7
  Open Source .................................................................................................................. 7
  Software Development Characteristics ....................................................................... 7
  Organizational Characteristics ..................................................................................... 14
  Software Peer Review ................................................................................................ 21
  Peer Review in Traditional Software Development .................................................... 22
  Peer Review in Open Source Software Development ................................................... 27

Chapter 3  Linus’s Law: a Preliminary Examination of OSS Peer Review Processes .... 32
  Software Peer Review ................................................................................................. 32
  Case Selection .............................................................................................................. 33
  Data Resources .......................................................................................................... 34
  Critical Activities of OSS Peer Review ...................................................................... 35
    Submission ................................................................................................................ 35
    Identification ............................................................................................................ 37
    Resolution ............................................................................................................... 40
    Evaluation ............................................................................................................... 41
  Discussion ................................................................................................................... 43

Chapter 4  Creative Collaboration in Open Source Software Peer Review ...................... 45
  Open Source Software Peer Review Practices ........................................................... 45
  Creative Collaboration ................................................................................................. 47
  Method: Comparative Case Studies ............................................................................ 47
  Data Collection and Analysis ....................................................................................... 49
  Results ......................................................................................................................... 50
    Subprocess 1: Problem Identification .................................................................... 51
    Subprocess 2: Preparation ...................................................................................... 55
    Subprocess 3: Solution Generation ......................................................................... 58
    Subprocess 4: Solution Evaluation ......................................................................... 61
LIST OF FIGURES

Figure 2-1: Peer Review Formality Spectrum (Wiegers, 2001) ........................................... 22

Figure A-1: An Example of Bug Reports in Mozilla................................................................. 151

Figure A-2: An Example of Bug Reports in Python................................................................. 152
LIST OF TABLES

Table 2-1: Comparison of traditional software peer review and OSS peer review. .................. 31
Table 3-1: Elements of OSS peer review critical activities. ..................................................... 42
Table 5-1: Distribution of the status of open bug reports. .................................................... 78
Table 5-2: Distribution of the resolutions of closed bug reports. .......................................... 78
Table 6-1: Overview of peer review processes in Mozilla and Python. ............................... 111
ACKNOWLEDGEMENTS

Choosing to a researcher’s life is choosing loneliness. But becoming a researcher is never a lonely process. I would like to thank all the people who guide me, help me, grow with me, and support me along my way to become a researcher. My parents gave me the simplest but best support, never interfering with my decisions but always being there whenever I needed them, while I could not be there for them as many of my peers did because of traveling all the way down here. I am so fortunate to have Jack Carroll as my advisor. There is an ancient Chinese saying, “once a teacher, forever a father”. Jack not only led me into the field of Human-Computer Interaction, but also influenced my views of science and engineering, career, and life. All my committee members have been so generous in supporting my dissertation research. They started their contributions early on during the planning of this dissertation research, and kept me motivated to carry out the tedious investigations. They, together with Angsana Techatassanasoontoon and Mary Beth Rosson, also greatly helped me with job search. Thanks to Patrick Shih and Yu Wu, who joined our research project at its later stage, offered a lot of insights and suggestions that shaped the development of this dissertation. I also would like to thank the colleagues graduated from the Computer-Supported Collaboration Lab, Umer Farooq, Paula Bach, Gregorio Convertino, Helena Mentis, Hao Jiang, Dejin Zhao, and Hansa Sinha, who shared their experiences and advice with me. The last but not least group of people I would like to thank are the lovely friends I met at Penn State. Yina Wei has been the best roommate one could ever get. Wen Yao, Lei Yao, Ruijuan Zhang, and Jingwen He have always been very supportive for me like my sisters. Zhe Liu and Chu Huang generously provided me a place to stay during my final semester at Penn State and considerately helped me get through this period.
Chapter 1

Introduction

“Given enough eyeballs, all bugs are shallow.”

- Linus’s Law (E. S. Raymond, 2001)

Linus’s law is a famous characterization of the widely believed cornerstone of Open Source Software (OSS) development, extensive peer review. It is a statement Eric S. Raymond made in his prominent book (E. S. Raymond, 2001) and named in honor of Linux’s founder, Linus Torvalds. Raymond coined it to emphasize the core difference between OSS development and traditional software development, for which Brooks’s law predicts that “adding manpower to a late software project makes it later” (Brooks, 1987). Despite anecdotal experiences shared by practitioners, Linus’s law has not been questioned or empirically examined. The mechanisms behind Linus’s law—whether detected bugs scale with number of peer reviewers and how the peer review process is organized in practice—remain unclear but critical to understand OSS development.

OSS emerged as a fundamentally distinctive but promising model of software development, distributed collaborative work, and innovation; the software applications and the socio-technical systems in which they are produced have also been evolving over the past decade. OSS development involves geographically dispersed experts collaborating over the Internet to create and maintain software that can be used by people all over the world for free (Kidane & Gloor, 2007). Beyond an ideology contradicting proprietary software, such as Microsoft, OSS development is no longer the synonym of the bazaar style as it was initially conceptualized. Some projects have undertaken a more institutional form with control and hierarchy, and some engage stakeholders of proprietary companies. More recently, the integration of social networking
infrastructures, such as GitHub\(^1\), has also created opportunities for OSS projects to leverage a large crowd that moves easily beyond project boundaries, transforming the way of self-presentation and trust building as well as the entire social structure.

Not only has OSS revolutionize the way people think of production, the OSS developed and adopted has also been influential and vital in all aspects of everyday life. It helps cut down the cost for numerous organizations (e.g., companies, schools, governments), support a variety of commercial software products (e.g., Apple’s OS X operating system, Google search engine), and enable research in various scientific disciplines (e.g., biological sciences, physics, mathematics). Therefore, enhancing the understanding of OSS development will contribute to its own improvement as well as inform other types of collaboration and innovation.

As the core practice of OSS development, peer review involves critical evaluation and improvement of software products (Moon & Sproull, 2000; Roberts, Hann, & Slaughter, 2006). The process often starts with someone reporting a bug or an issue. Others analyze the bug or issue and determine whether it should be fixed or addressed. Eventually, a solution (e.g., a patch) is reached and integrated into the current software product. Given the openness of OSS, peers in the review process can be anyone from the large community, which is beyond the group of software developers and encompass all the people contributing to the project. This nature of volunteer-based ad hoc collaboration mediated by Internet-based technologies and shaped by community culture, norms, and value renders literatures on other software development and collaboration models insufficient to explain the OSS peer review process. Despite the importance of peer review to OSS development, very few studies analyzed this practice in particular. They only reported some general statistics (Asundi & Jayant, 2007; Nurolahzade, Nasehi, Khandkar, & Rawal, 2009; P. Rigby, German, & Storey, 2008), described documented policies (J. P. Johnson, 2006; P. C. Rigby & German, 2006), or developed some extraction methods and quantitative

\(^1\) http://github.com
metrics to analyze online content of peer review (Barcellini, Détienne, Burkhardt, & Sack, 2005; Mockus, Fielding, & Herbsleb, 2000; P. Rigby, et al., 2008).

This dissertation is to enhance the understanding of OSS peer review processes. Its contributions are trifold. First, the investigation examines Linus’s law, articulating whether and how it is instantiated in OSS development. Second, it characterizes and codifies the work and social processes of OSS peer review. Third, the work provides guidance of designing socio-technical interventions to better support the OSS peer review process.

**Research Statement and Scope**

My objective is to understand and support OSS peer review processes. The numerous OSS projects emerged over the years and their evolving nature have formed an enormous research space. Moreover, these OSS projects differ in the products they are producing (e.g., intended user bases, complexity) and the socio-technical infrastructures they are co-developing with (e.g., social structure, collaboration technologies). Taking these characteristics into account, my empirical investigation seeks to answer the following questions:

*RQ1. How is OSS peer review commonly conducted?*

*RQ2. How may different OSS projects carry out peer review differently?*

*RQ3. How can socio-technical affordances be designed to better support OSS peer review processes, both the commonalities and differences across projects?*

The goal of this research favors relatively large and well-established OSS projects, in which peer review is likely to be carried out in a consistent and repeatable way rather than spontaneously, embody rich and complex phenomena instead of evident interactions (e.g., dyads reviewing each other’s code), and have substantial instances that include successfully completed processes. Such OSS projects often engage communities beyond a small group of software
developers. Although they cannot entirely represent the vast landscape of OSS projects, they constitute an important part of it. Furthermore, they are influential on shaping the open source paradigm, inspiring and guiding other OSS projects’ organization and operation.

I use comparative case studies to pursue the objective of this research work. It is not possible to investigate all type of OSS projects in depth for a dissertation, even if limiting the cases to large projects. Mozilla and Python, two established and well recognized OSS projects, are compared to both highlight the theoretical consistencies (Yin, 2008) and capture the contextual variations (K. Crowston, Wei, Howison, & Wiggins, 2012). The two cases were selected for several reasons. They are both supported by non-profit organizations, which to some extent exert control and impose hierarchies to their work processes. They are also both supported by relatively large communities, which consist of core developers who contribute significantly to the project and peripheral members who use the software but rarely participate in developing the software. Their similarities also lie in the computing infrastructures for their peer review processes, although the specific designs of these technologies vary. The task-oriented and communication technologies include version control systems (Mercurial for Mozilla and Subversion for Python), bug tracking systems (Bugzilla for Mozilla and Roundup for Python), IRC, mailing lists and wikis. For code review, Bugzilla implements a patch reviewer tracking code changes, while Python recommends an external code review tool Rietveld. In addition to the differences of supporting tools, Mozilla and Python differ in their primary software products in terms of the types—a web browser for end users (i.e., Mozilla Firefox) and a script language for software professionals (i.e., Python language)—as well as the scale and complexity—Mozilla Firefox has 9,326,438 lines of code, while Python implements 866,177 lines by the end of August, 2013 according to the analytics service provider for OSS code repositories, Ohloh².

---
² [http://www.ohloh.net/](http://www.ohloh.net/)
These distinctions between products may further result in different sizes and compositions of communities.

Studies in this dissertation are focused on the OSS peer review activities archived in bug tracking systems, which many large projects primarily rely on, rather than other leaner and more generic tools, such as mailing lists. Bug tracking systems play a central role in supporting collaboration between developers and users of a software project (Breu, Premraj, Sillito, & Zimmermann, 2010). They function as a database for tracking bugs, features, and other inquiries, and also serve as the focal point for communication and coordination for many stakeholders (e.g., customers, project managers, quality assurance personnel within and beyond software development team) (Bertram, Voida, Greenberg, & Walker, 2010).

Organization of the Dissertation

The organization of this dissertation follows the sequence of the research stages through which the studies were conducted. Beyond the methodology aspect, the structure also situates OSS peer review in a historical context—from characterizing how it connects and differentiates from the past (i.e., traditional software review methods) to how it informs the future (i.e., emerging projects supported by new socio-technical infrastructures, like GitHub). It begins with a software engineering perspective, sketching the process itself, to social and design perspectives, drilling down to the actors, their interactions, the community environment, and the technologies embedded.

Each research stage has a distinct chapter for the literature, research goals, methodology, findings, and discussion. Despite the inevitable overlaps between chapters, I feel that this division will not cause confusion considering each stage pursued distinct goals and conducted different data analyses. Specifically, the dissertation is organized as follows.
In chapter 2, based on OSS literatures I provide an overall characterization of OSS development from both a software engineering perspective and a social perspective. Narrowing down to the peer review process, I present the background of software peer review, enumerating its traditional approaches, and then the contemporary understanding of OSS peer review and the gap in current research. In chapter 3, I report the empirical examination of Linus’s law, which suggests its limitation in explaining the actual outcomes. I also present an initial codification of the common process of OSS peer review and compare it with the traditional software review methods. The study presented in chapter 4 takes a step further of characterizing the commonalities. It highlights the creative collaboration that emerged the OSS peer review processes. I also discuss how tools can be designed to support such collaborative creativity. After establishing an overall understanding of OSS peer review through two studies, I shifted my focus onto more nuanced analyses of individual cases, identifying their uniqueness and differences. Chapter 5 particularly shows the Mozilla case, the peer review of which was found to be more inconsistent with Linus’s law. I focus on the issue of member diversity, a key factor contributing to the inconsistency, to characterize the actors and their social environment in the peer review collaboration. Continuing the social lens, chapter 6 returns to a larger scope with a presentation of contrasts between Mozilla and Python. It discusses how the two cases can serve as different models for different growing projects that are approaching a large scale and are becoming mature thanks to the integration of new technical affordances into OSS development. I also raise issues related to socio-technical designs that may address the variations identified between projects. Chapter 7 concludes the dissertation with reflections on the major findings of the research, articulation of the contributions of this research, and plans for future research.
Chapter 2

Overview of Open Source Software Development and Software Peer Review

Open Source

Open source is a multi-facet concept. It can refer to a software development approach, or a type of virtual organizations, or a business model. In this section, I identify its software development characteristics from a software engineering perspective and its organizational characteristics from a social perspective drawing from the huge volume of OSS research.

Software Development Characteristics

Software, especially the one of a large scale, has a very high rate of failures. Its inherent properties, such as complexity, conformity, changeability and invisibility, are identified as impeding software success (Brooks, 1987). Traditional software engineering approaches, for example, the waterfall model and the spiral model, all suffer from these difficulties.

Distinct from its traditional counterpart, OSS development emerges as a successful model to organize software projects. It involves global volunteers motivated not by monetary return but by the enjoyment of work. Its practices are often described as counter intuitive, including implicit requirements and design, massive peer code review, and rapid releases of prototype code (Bollinger, 1999; Charles, 1998; J. Edwards, 1998; Fielding, 1999; Hecker, 1999; Lawrence, 1998; Madey, Freeh, & Tynan, 2002; O'Reilly, 1999; E. S. Raymond, 2001; Sanders, 1998; Torvalds, 1999).
Motivated Developers

One salient characteristic of OSS developers is their high non-monetary motivations. Rather than motivated by salary--like most of the software developers in proprietary companies, OSS developers work voluntarily. This feature is considered one of the fundamental success factors of open source software (Koch & Schneider, 2000).

Open source researchers put great efforts on discovering what motivate OSS developers and users participate in and contribute to the project. Ghosh summarizes the incentives as software functionality, enjoyment, and a desire to be “part of a team” (R. A. Ghosh, 1998). Raymond (E. S. Raymond, 2001) describes three basic motives. First, developers may use the software code developed by themselves. Thus they can benefit from their efforts directly. Second, they enjoy programming work, which is also known as “hacker culture” (Lakhani & Wolf, 2003). Third, they can enhance their reputation by contributing to the projects with massive peer evaluation (E. S. Raymond, 2001). Lakhani and Wolf (Lakhani & Wolf, 2002) and Hertel et al. (Hertel, Niedner, & Herrmann, 2003) articulate contributors’ motivations in a more quantitative way by conducting surveys. Their results show that the highest-ranking motivation is the contributors’ own need for the software developed. Improvement of programming skills and the enjoyment of programming also strongly stimulate their contribution. Reputation enhancement as an incentive is not well supported by their results, which is interpreted as the consequence of self-reporting bias. Furthermore, with an increasing commercial interest in OSS products, many firms are also involved in OSS development (e.g., IBM in Eclipse and Apache, Sun in Netbeans). Therefore, developers in these projects may also be motivated by monetary compensation.

Some researchers describe the motivations and drivers for OSS development in an even broader sense. Feller et al. (J. Feller & Fitzgerald, 2000) identify them from technological, economical and socio-political perspectives. While the socio-political drivers are similar as the
incentives discussed above, technological drivers include the need for robust code, faster development cycles, higher standards of quality, reliability and stability, and more open standards and platforms, and the economical one is the corporate need for shared cost and shared risk.

To understand the relationship among multiple motivations, Roberts et al. (Roberts, et al., 2006) categorize them into pure intrinsic ones (i.e. the joy and challenge of the work), internalized extrinsic ones (i.e. solve a problem of personal use benefit and reputation enhancement) and pure extrinsic ones (i.e. money) (Deci & Ryan, 2000). On the basis of their longitudinal study on Apache, they suggest that extrinsic motivation may not crowd out pure intrinsic motivations but internalized extrinsic motivations. They further indicate that status motivations as internalized extrinsic motivations enhance intrinsic ones.

*High Modularity*

As any software development approach, OSS development also has three fundamental components: people, process and product. Therefore, OSS development should also be characterized by its product—software. Most of OSS has high modularity and extensibility (K. Crowston & Scozzi, 2002; Jensen, 2003; Lerner & Tirole, 2002; O'Reilly, 1999), which is also regarded as the favorable characteristic for its success.

In software development, modularity refers to decomposition of the software system into multiple modules. It can minimize and standardize the interdependencies among components. Modularity is not a new term created for OSS development. Instead, it is a general property of complex systems, such as projects in industrial manufacturing and software engineering (Baldwin & Clark, 1997; Schilling, 2000).

Many successful OSS are highly modularized, especially operating systems originated from Unix. Torvalds, the inventor of Linux, attributes the success of Linux to its modularity
(Bonaccorsi & Rossi, 2003). He created the small and compact kernel of Linux that can perform basic tasks, and left functionalities to be extended by other voluntary developers. Its standardized interfaces and translation processes generate smooth module boundaries and facilitate rapid recombination (Tuomi, 2001). GNU operating system also has a massively modularized structure, which takes advantage of the architecture of Unix that enables easy decomposition into distinct development tasks (Lerner & Tirole, 2002). Another famous example of OSS, Apache, also leaves its features except the core functionality in modules. Developers can concurrently work on the particular feature they identified or they have skills and interest in. Various modules will be selectively compiled and configured (Mockus, et al., 2000).

Modularity is considered as the fundamental and indispensible feature of OSS because its developers are dispersedly located individuals or groups (Lerner & Tirole, 2002; Paulson, Succi, & Eberlein, 2004). By reducing the sequential or reciprocal dependencies, it minimizes the work coupling and enables self-governance structure (K. Crowston & Scozzi, 2002; Sharma, Sugumaran, & Rajagopalan, 2002).

**Informal Requirements and Design**

Traditional software development models mostly formalize their processes into several phases, including requirements, design, implementation, testing, deployment and maintenance. On the contrary, OSS development does not have specific and explicit requirements or design (E. S. Raymond, 2001; Vixie, Dibona, Ockman, & Stone, 1999). In OSS projects, requirements and design elicited, analyzed, specified, validated and managed through Web-based informal discourses.

Scacchi introduced the term “software informalism” for OSS requirements engineering from case studies on four OSS communities (i.e. networked computer games, the Internet/Web
infrastructure, X-ray astronomy and deep-space imaging and academic software design research) (Scacchi, 2002). By “informalism” he refers to the artifacts participants use to describe, proscribe, or prescribe what is happening in a project. These artifacts are implied efforts of requirements engineering that emerge as a by-product of community discourses about the system features and developers’ responsibilities. Furthermore, they serve as coordination mechanisms that help participants communicate, cooperate, document and make sense of the work processes (Jensen & Scacchi, 2005; K. Schmidt & Simonee, 1996; Simone, Mark, & Giubbilei, 1999). Jensen et al. (Jensen & Scacchi, 2005) further point out that the informalisms not only facilitate communication within a project, but “emerge as boundary objects (Star, 1989) through which intercommunity activities and relations are negotiated and revised”. Though no explicit objective or driving force leads to refining these informal requirements into formal ones (Cybulski & Reed, 1998; Jackson, 1995; Kotonya & Sommerville, 1998), the informalisms are still valuable because they capture the detailed rationale and debates for what changes were made in particular development activities, artifacts, or source code files (Scacchi, 2007).

Similarly, other researchers also emphasize the value of informal discussions in OSS projects. Yamauchi et al. (Yamauchi, Yokozawa, Shinohara, & Ishida, 2000) suggest that the informal discussions among OSS community members through e-mails can serve as organizational memory for learning. By examining the primary media (e.g., source code control systems, TODO lists, and mailing lists) of two OSS communities—FreeBSD Newconfig Project and GNU GCC project, they noticed that the messages of mailing lists are usually not abstract discussions or coordination but more action-based ones, such as discussions about specific software problems. Halloran et al. (Halloran & Scherlis, 2002) also consider these semi-structured texts in mailing lists as organizational memory. The texts record both issue resolutions and rationale for later use.
Warsta et al. also observed that no formal documentation is conducted in OSS projects (Warsta & Abrahamsson, 2003). They conceive the OSS development process as one without any predefined formal documentation norms or customs. They also claim that OSS development process lacks of system level designs, defined processes and many other traditional coordinating mechanisms like explicit plans and schedules. This observation is also consistent with the claims made by many other researchers that OSS projects are not dictated by lists of deliverables, nor formal plans, schedules (Mockus, et al., 2000; E. S. Raymond, 2001; D. C. Schmidt & Porter, 2001).

Nevertheless, some studies show conflicting conclusions about design formality in OSS projects. Reis et al. suggest that in Mozilla, most decisions on functionality are discussed by participants. Abstract high-level requirements are very few and they are set by Mozilla.org management. In contrast, user interface and the module APIs are specified and documented. (Reis & de Mattos Fortes, 2002). On the contrary, Twidale and Nichols observe the usability design is only informally discussed and documented (Nichols & Twidale, 2003; Twidale & Nichols, 2005).

Regardless of variation among different OSS projects including the cases discussed above and elsewhere (Henderson, 2000; McConnell, 1999), “inform” has two meanings here. First, ideas about requirements and design are not formally documented but informally discussed. Second, neither formal procedure nor explicit specifications for requirements and design are defined. In other words, decision for requirements and design is not made by any authority but may integrate perspectives of other participants.

*Extensive Peer Review*

Peer review is critical in OSS development. Given the high modularity of the software, OSS developers mostly create code independently. They will not be able to examine others’
individual work until it is implemented and submitted. Thus most projects require developers to test their own code before they submit it. Although the testing may ensure the code can function, the product still needs further review for meeting higher-level expectations. Eventually other developers will decide whether it is accepted by the project.

Extensive peer review is always regarded as the most appealing feature of OSS development. Raymond’s interpretation of Linus’s law, “Given enough eyeballs, all bugs are shallow” (E. Raymond, 1999), is quoted by many researchers to describe the advantage of OSS in quality (DiBona et al., 2000; O'Reilly, 1999; Vixie, et al., 1999). When the developers finish implementing the feature they chose to work on, they submit the code for peer review to avoid faulty code being integrated into the code repository. This review process varies across different projects. Some is via informal posting of the code on a mailing list, and adding it to repository if no one objects within a certain amount of time, while others require authorities to accept the code, as the cases in the Linux Project, Mozilla project and Apache project (Mockus, et al., 2000; Mockus, T Fielding, & D Herbsleb, 2002; O'Reilly, 1999; Reis & de Mattos Fortes, 2002). The review is mainly focused on code review, while design review is also claimed to be considered (Eich & Baker).

**Rapid Iteration and Release**

Another feature of OSS development processes is its iteration and rapid release (Jensen, 2003; Mockus, et al., 2000). Passing testing is not the final stable stage of complex software systems. Instead, OSS development is a continuous and cyclic process. This iterative process and fast release are beneficial for OSS development: it can adapt to contingent requirements and facilitate fast evolution of software systems.
Compared to other models of iterative development, OSS development is impressive at the speed and frequency of its iteration and release. Iterative development is not a new term in software engineering. As opposed to waterfall development model, iterative and incremental development was proposed in mid-1950s (Larman & Basili, 2003). In iterative development, developers revise their plans or requirement specifications on the basis of testing results and user feedback. As a result, they improve the product quality. Similarly in OSS development, the leader, the core team or even the whole community selectively combines committed modules as new features with the modified previous version and release it to the public. New implementations happen at a fast pace. Although this rapid release is usually not referring to the production but the development releases (Eide, 2007; Mockus, et al., 2002), designers and developers may still gain higher-level and diverse feedback on their committed product than at peer review state.

**Organizational Characteristics**

People engaged in OSS development constitute the OSS communities, which are virtual organizations in nature. Virtual organizations are the work organizations that rely fundamentally on the Internet (Mowshowitz, 1986). Their members have shared goals or interests, collaboratively doing their knowledge work via information and communication technologies (ICTs) (Ahuja & Carley, 1998). OSS communities can be characterized in the same way: (1) community members share a common interest in developing software, (2) most of the time, communities are geographically distributed and the activities of members are coordinated by Internet, and (3) members’ collaborative work is mainly mediated by source code control systems, mailing lists, and web portals (K. Crowston & Scozzi, 2002).
OSS communities differ from traditional proprietary organizations. I characterize the differences from organizational structure, composition and rewards system in the following sections.

**Hybrid Structure**

Structure is regarded as one of the key properties of organizations as well as one of the critical determinants of organizational creativity (Anderson, De Dreu, & Nijstad, 2004; Andriopoulos, 2001; Damanpour, 1991; Woodman, Sawyer, & Griffin, 1993). Organizational structure typically refers to “the formal allocation of work roles and the administrative mechanisms to control and integrate work activities” (Child, 1972).

Distinct from the hierarchical structure of traditional organizations, the structure of OSS communities is normally less hierarchical. However, their structures vary from community to community. Most of the successful OSS communities and the ones with commercial involvement are organized in a hybrid structure with leaders or core teams in the center and other members at the periphery.

The structure of OSS communities, or the roles of community members, is normally characterized by the levels of participation and contribution of members. Many researchers observed skewed distribution of participation and contribution. By analyzing 3149 distinct active open source projects Ghosh and Prakash found skewed distribution of contribution that 10% of the contributors accounted for 72% of the code contributed to the project (R. Ghosh & Prakash, 2001). Krishnamurthy examined 100 mature open source projects from Sourceforge database, and discovered that the vast majority of mature OSS projects are developed by a small number of individuals (Krishnamurthy, 2004). Although this analysis may not be representative for successful OSS communities because almost all of the sampled projects are small with rare
communication, but many other case studies indicate similar distributions. For example, in Apache project about 15 core developers contributed more than 80% of the code. A group larger than the core by about an order of magnitude contributed to defect fixing, and a group larger by an order of magnitude again contributed to problem report (Mockus, et al., 2000). In GNOME project, 52 programmers out of 301 wrote 80% of the code (Koch & Schneider, 2000). Analysis in other projects classified even more roles, such as maintainers in Linux (Lee & Cole, 2000), and module-level managers in Netbeans.org (Jensen & Scacchi, 2005). Some researchers even use a layer model to describe the organization (Maass, May, 2004; Moon & Sproull, 2000) which has been criticized by Ducheneaut (Ducheneaut, 2005) for its static view. Though researchers use different stratification strategies and different terms to describe the structural feature of open source community (Bezroukov, 1999; K. Crowston & Scozzi, 2002; Fielding, 1999; Gallivan, 2001; Maass, May, 2004; K. Nakakoji, Yamamoto, Nishinaka, Kishida, & Ye, 2002; E. S. Raymond, 2001), the basic components of OSS communities are core contributors (a single leader like Torvalds in Linux or a core group) and peripheral members. Another important feature of community members’ roles is that they are not enforced by external power, nor is the work assigned explicitly to developers and designers (Bonaccorsi & Rossi, 2003; Jensen & Scacchi, 2005). The core team may change dynamically according to their contribution to the community.

This hybrid structure of OSS communities at a large scale is a balance of organizational efficiency and creativity, especially favored by commercial interests in OSS (Sharma, et al., 2002). Jensen et al. (Jensen & Scacchi, 2005) explains that this lack of clear hierarchical structure is both a cause of freedom and chaos in OSS development.
Diverse and Dynamic Expertise

OSS communities can assemble complemented expertise without limited by corporate, geographical, national, and historical boundaries (Bement & Atkins, 2007). In a broad sense, skills and expertise can even serve as one of the community resources. They are self-allocated and dynamically flowing within and across communities.

Members of OSS communities possess more heterogeneous expertise than those of traditional organizations, although the diversity may not be greater than other virtual communities. They are diverse in terms of know-how, interests and user needs (Bonaccorsi & Rossi, 2003; Kogut & Metiu, 2001). The more significant characteristic of expertise in OSS communities is its dynamic configuration in communities. On the one hand, no central allocation of skills and expertise to a particular task is conducted in OSS communities. In presence of a pending job, more than one volunteer may contribute. Although contributors can be aware of what other members are doing via mailing lists, thereby the duplicated efforts, they probably will not be stimulated to shift to other tasks that are less appealing to or less suitable for them. Such strategy of allocating human resources is often considered by commercial companies to be inefficient. However, redundancy of efforts allows the generation of more resolutions, which is conducive to both quality of the code (Bonaccorsi & Rossi, 2003) and creativity of the community. On the other hand, rather than bound by explicit contract as in commercial corporations, members of OSS communities have the freedom to choose join, stay or leave the community. This dynamic and fluid membership is essential to community success in two aspects. First, it is difficult for OSS communities that fail to attract and retain new contributors to grow. Second, as a result of skewed distribution of contributions, the leave of core team members may cause loss of critical knowledge, which is detrimental to long-term development of the community (Ducheneaut, 2005).
By analyzing the socialization process of open source community newcomers, Sack et al. (Sack et al., 2006) suggest two obstacles for communities to attract new members to join and contribute. One obstacle is a significant delay from joining the community to feeling able to contribute. The other is that potential contributors lack the necessary political acumen to promote their ideas.

*Encouraging Reward Systems*

Reward is a common mechanism of organizations to motivate their employees. Although members in virtual organizations are often not motivated by extrinsic incentives, the reward systems are also critical. They are also challenging because intrinsic motivations are usually volatile. In the reward systems of OSS communities, members evaluate others’ work using certain criteria. Accepted contributions are rewarded by community’s recognition, reputation enhancement or even status promotion.

The criteria about what should be recognized as a contribution vary from community to community. One possible reason for this variation is that the way in which decision has been made largely depends on the structure of the community. In Apache project, even an identification of a problem without proposed a solution will be acknowledged as a contribution to the community. All contributions are posted on community’s website and most committed contributors are highlighted (Lerner & Tirole, 2002). Linux normally defines a patch as contribution (Hertel, et al., 2003). Edwards (K. Edwards, 2003) takes it even farther, saying when evaluating solutions, especially when alternatives also exist, performance and beauty may be two basic criteria. He describes a story of Linux that Linus Torvalds has rejected a patch that solved the problem, but the solution ‘looked’ wrong. “The right solution is simple and easy to
understand, it is logical and solves the problem with a minimum of workarounds”. Thus this beauty criterion is much more subjective than performance.

Moreover, unfairness is not always caused by centralized leadership, but also arises in decentralized organizations. Perhaps unfairness is not an exact description; it is the property of criteria that are not completely devoted to judging the technical expertise of contributors. Ducheneaut (Ducheneaut, 2005) noticed this phenomenon in Python, not all the participants of which with high skills become important developers. The ones who know the norms of status promotion are more likely to be raised among other community members by enrolling allies. Another possible reason for this unequal chance of promotion, which has not been discussed by Ducheneaut, is related to a common issue of virtual organizations. In virtual organizations, it is easy to become invisible and dismissed. Members and sites are visible in proportion to their contribution. People at peripheral sites must work harder to have the same visibility as people at larger and more central sites (Bos, Judith S. Olson, & Olson; Martins, Gilson, & Maynard, 2004).

Other than how a contribution is accepted and recognized, reward systems also determine what contributors will obtain from evaluation results. Both theoretical and empirical studies show that contributors will benefit from their accepted contributions in various ways, such as community identification, reputation enhancement, power, career development and money (R. A. Ghosh, Glott, Krieger, & Robles, 2002; Lakhani & Wolf, 2003; Lerner & Tirole, 2000; E. Raymond, 1999). Some researchers label this reward mechanism in OSS communities as meritocracy (Gacek & Arief, 2004; Roberts, et al., 2006), which means “the more work you have done, the more you are allowed to do” (Fielding, 1999). Gacek et al. (Gacek & Arief, 2004) also noticed that the transition of status may be delayed by the timing and other obstacles, and it also depends on the actual structure of the community.
Open Flow of Information Exchange

The public access to code and discussions enables the information in OSS communities flowing through less hierarchical but more networking structure. Communication and information sharing within virtual organizations, particularly OSS communities, is in electronic forms. Electronic communication is expected to support lateral communication and broad participation across social groups (DeSanctis & Monge, 1999).

The most significant feature of OSS communities, by its name, is that the major part of information that community members are sharing is their individual innovation—the code. Open code as a public information good has been discussed with great enthusiasm among economists, for example, the “private-collective” innovation model proposed by Hipper and Krogh (Hippel & Krogh, 2003). Issues around intellectual property also arise subsequently. These discussions are out of the scope of this paper. Giving public access to code is characterized by many researchers and practitioners as “gift-giving” culture of OSS communities (Bergquist & Ljungberg, 2001; E. Raymond, 1999). Stewart and Gosain (K. Stewart & Gosain, 2001) examined relationship between trust and the coherent ideology of gift giving norm and focus on reputation. They claim that this ideology enables the development of affective and cognitive trust in OSS communities, which leads to group efficacy and effectiveness.

Members of OSS communities also have personal communication and discussions as members in other virtual organizations. They interact with each other via web portals, mailing lists or even chatting tools (Halloran & Scherlis, 2002). This virtual form of communication allows higher anonymity than face-to-face communication.
Software Peer Review

Peer review is a process in which one’s performance or the quality of one’s work products is evaluated by other colleagues. It is used extensively in various fields, such as scientific research, software development, and healthcare. Its purpose is to detect flaws in one’s work and discover potential for further improvement.

Peer review is one type of software review, which is traditionally regarded as a quality control approach. Different from other types of review, such as management review and audit, the primary objective of peer review is to discover defects (or bugs) as early as possible during development processes, suggest improvements and even help developers create better products (Wiegers, 2001). Other than revealing that an execution outcome fails to meet expectation, peer review can also evaluate other dimensions of quality, such as understandability, maintainability and conciseness.

The benefits of software peer review not only include quality assessment, but also involve facilitating communication and knowledge sharing among project members. By spreading knowledge with respect to product, process, project, and technical issues, reviews enable other members to maintain a work product. At the same time, reviews help establish shared understanding of objectives as well as work products (Wiegers, 2002). Furthermore, these reviews may encourage creativity and innovation.

Code is not the only object that peer review can assess; review candidates can be any artifacts created along the course of development. According to the Institute of Electrical and Electronics Engineers “Standard for Software Reviews” (Std 1028-1997) (The Institute of Electrical and Electronics Engineers, 1999), software products that can be reviewed include requirements specifications, use cases, project plans, user interface design and prototypes, and
Regardless of the development model, peer review provides quality appraisal upon each major deliverable before subsequent processes that depend on the assessed deliverable (Wiegers, 2001).

Given the differences between OSS development and traditional software engineering models, their peer review practices also differentiate from each other. The following sections discuss the types, process, participants and roles, and tools of peer review in both traditional software development and OSS development. Table 2-1, at the end of this section, summarizes the comparison between the two types of reviews.

Peer Review in Traditional Software Development

Types

Peer review can take a variety of forms with different degrees of formality and flexibility (Iisakka & Tervonen, 1998). Wiegers (Wiegers, 2001) places some of the common review methods along a formality scale, as illustrated in Figure 2-1. The following discussion will focus on the most formal review—inspection, and one of the widely used informal reviews, walkthrough.
Both advantages and disadvantages of each type have been discussed extensively in terms of their cost and benefits (Freedman & Weinberg, 2000; Weinberg & Freedman, 1984). Different methods serve for different primary objectives; they can even be combined to complement each other, for example, inspections with informal reviews. Among all these types of reviews, inspections are the most systematic and rigorous one, which were introduced by Fagan’s salient work (Fagan, 1976). An inspection has a clearly specified process, although different methods for inspection have been proposed (Bisant & Lyle, 1989; Gilb, Graham, & Finzi, 1993; Knight & Myers, 1993; Parnas & Weiss, 1985; Schneider, Martin, & Tsai, 1992). However, it is also very labor-intensive (P. Johnson & Tjahjono, 1993; AA Porter, Siy, & Votta, 1998). Comparatively, studies found that walkthroughs cost less with their informality. In walkthroughs, “a designer or programmer leads members of the development team and other interested parties through a software product, and the participants ask questions and make comments about possible errors, violation of development standards, and other problems” (The Institute of Electrical and Electronics Engineers, 1999). They are usually used in evaluating usability, and are particularly helpful for novices to improve performance (Bias, 1991; Yourdon, 1979).

Even though reviews vary in terms of the activities involved, they pursue the same goal—to detect defects in software products at any phase of the development cycle. They often proceed through two stages: peers first independently inspect the software product; then they pool their efforts in a group meeting to reach consensus on all the defects. During the meeting, peers play different roles and finally generate a report with all their findings (Sauer, Jeffery, Land, & Yetton, 2000).
Criteria

A large number of quantitative metrics can be used to evaluate software quality. Boehm et al. (Boehm, Brown, & Lipow, 1976) defined a software quality characteristic tree to guide software quality assessment, including metrics such as device-independence, robustness, completeness, conciseness, and consistency.

Process

Activities in peer review consist of planning, preparation, examination (usually involves one or multiple meetings), rework/follow-up, and exit evaluation (The Institute of Electrical and Electronics Engineers, 1999; Wiegers, 2001). Inspections probably have the most strict procedure, commonly including planning, overview, preparation, meeting, rework, follow-up, and causal analysis. In contrast, walkthroughs do not have a defined process, nor do they specify exit criteria, or require management reporting (Fagan, 1986). Wiegers’ book provides a detailed comparison (Wiegers, 2001).

Participants and Roles

Multiple roles need to be established either formally or informally in peer review. In inspections, five types of roles are often engaged: the inspection leader/moderator, the recorder, readers, the author, and inspectors. The author can act as an inspector but not the other three roles. All participants are inspectors and they may play more than one role. Instead, walkthroughs only require no fewer than two participants. Roles include the leader, the recorder, the author, and team members. The author usually leads the examination, presents the products, and sometimes
even records the appraisal. In any type of peer review, most of the time managers are suggested not to participate (The Institute of Electrical and Electronics Engineers, 1999).

Previous research did not agree on the effective team size of peer review. Porter et al. (A Porter, Siy, Toman, & Votta, 1995) found that expert pairs work as well as 4-review teams. Madachy et al. (Madachy, Little, & Fan, 1993) suggest five reviewers, Grady (Grady, 1992) recommends four to five as the optimal size, while Gilb (Gilb, 1988) proposes three to five or even more.

Another controversy about peer review is the necessity of a meeting. Holding a meeting may enable collaboration on discovering new defects and synthesize their findings. However, some studies indicate that the effectiveness of meeting-based reviews is similar to that of asynchronous reviews without a meeting (P. Johnson & Tjahjono, 1997). In contrast, an inspection meeting may suffer from difficulty in coordinating participants’ time conflicts and relatively slow coverage of the material being inspected (Wiegers, 2001). Moreover, traditional face-to-face meeting can cause process losses such as attention blocking, air time fragmentation, and domination (Nunamakar, Dennis, Valacich, Vogel, & George, 1991).

Techniques

Techniques for detecting defects include checklists, scenarios, rule sets, and ad hoc reading (Ciolkowski, Laitenberger, & Biffl, 2003; Wiegers, 2001). Checklists are common in software inspections. They usually list the questions the inspectors need to ask when they examine the deliverables. Scenarios are an analysis technique that is often used to examine requirement specifications and usability (A Porter & Votta, 1998; Rosson & Carroll, 2002). Rule sets collect statements that direct authors to perform tasks or create product documents in a
particular way (Gilb, et al., 1993). Ad hoc reading does not have any defined standards; a reviewer just reads the document and notes any defects encountered (Ciolkowski, et al., 2003).

To reduce labor costs of software review and enhance its effectiveness, researchers developed several tools to support review collaboration. These tools are designed to handle documents, prepare individuals’ review, support meeting, and collect data. ICICLE (Intelligent Code Inspection in a C Language Environment) (Brothers, Sembugamoorthy, & Muller, 1990) is an automated intelligent inspection tool developed to support the face-to-face inspection of C code. It is unique in making use of knowledge to help find common defects. It is designed to support two phases of inspection: individual preparation and the inspection meeting itself.

Scrutiny (Gintell et al., 1993) is a general tool supporting inspection of software products. It supports four inspection stages: initiation, preparation, resolution, and completion. Collaborative Software Inspection (CSI) (Mashayekhi, Drake, Tsai, & Riedl, 1993) is designed to enable the use of either the Yourdon (Yourdon, 1979) or Humphrey model (Humphrey, 1990) of inspection.

InspeQ (Inspecting software in phases to ensure Quality) is a toolset developed by Knight and Myers (Knight & Myers, 1993) to support their phased inspection method. Collaborative Software Review System (CSRS) (P. Johnson & Tjahjono, 1993) is an environment to support the use of FTArm (Formal Technical Asynchronous review method) (P. Johnson, 1994), a development of Fagan inspection.

Despite great endeavors for improving software review processes, traditional software development projects are still reluctant to employ peer review in their work. Even the companies that adopt software review in their practices did not implement reviews systematically. The review participants fail to make the best use of review’s benefits (Ciolkowski, et al., 2003).
Peer Review in Open Source Software Development

**Types**

OSS development deviates from traditional software engineering models, with informality and no explicit stages. Peer review is also in an informal form in most of OSS projects, while commercial versions of some projects have relatively formal review practices, such as BSD, BIND, and Sendmail. Some other large and mature projects define their standards and manage processes in a more formal way than the small and developing projects (P. C. Rigby & German, 2006; Vixie, et al., 1999).

Although large scale of peer review is considered a salient advantage of OSS development, very little research has investigated review activities in depth. Most of that research focuses on code review, which is often triggered by a “check-in” action or a bug report (K Crowston & Scozzi, 2004; Nurolahzade, et al., 2009; P. Rigby, et al., 2008; Vixie, et al., 1999). Few discuss the reviews on design and usability issues (Twidale & Nichols, 2005). Many researchers emphasized the asynchronous and distributed nature of OSS peer review (Stark, 2002).

**Criteria**

Many OSS projects have a statement of their review criteria, although they vary with the projects’ philosophy. Other than traditional software quality metrics (Boehm, et al., 1976), evaluation may also be concerned with elegance, developers with a high level of reputation, distributions, and tools indicating the level of activity (de Joode, 2004).
Process

Peer review processes of OSS projects differ from each other. Rigby and German (P. C. Rigby & German, 2006) summarized the common elements of review, including a coding standard, documentation requirement, independent, complete and small code, and contributors’ responsibility of ensuring the occurrence of review.

Generally, OSS code review consists of two kinds of procedures (Jorgensen, 2005; Mockus, et al., 2000; Mockus, et al., 2002). One is Review-Then-Commit (RTC), which requires a submitted patch being reviewed by other developers before it is applied to the project. This procedure is considered not time-efficient. It is often used for significant change of the software system. The other is Commit-Then-Review (CTR), which highly relies on the developer’s work quality. It is usually applied to relatively trivial and simple modifications to the system.

Participants and Roles

Roles involved in OSS peer review mostly include a bug reporter, a contributor, a tester, a reviewer, and a committer (P. C. Rigby & German, 2006). Some projects have an explicit assignment of reviewers, usually including at least one core member (K Crowston & Scozzi, 2004; Eich & Baker). Some other projects only adopt an autonomous way to share the responsibility.

The size of OSS review team spans a wide range. It can be only two, including the author and a senior developer. It can also be the whole community, in which the peripheral members irregularly contribute (R. Ghosh & Prakash, 2001; Koch & Schneider, 2000; Krishnamurthy, 2004; Mockus, et al., 2000). Linus’s law best depicts the potential of such a large pool.
Szczepanska et al. (Szczepanska, Bergquist, & Ljungberg) interpret this core-peripheral structure as an illustration of meritocracy and power relationship within the community.

**Tools**

Since OSS development is highly geographically distributed, the majority of its activities are mediated and recorded by computer-supported tools (Yamauchi, et al., 2000). Tools related to its review activities include bug tracking systems/issue tracker, source configuration management systems, mailing lists, discussion forums, instant messaging, and so forth (Halloran & Scherlis, 2002).

Bug tracking systems or issue tracker systems provide environment for peer review by preserving all the registered bugs. Bugzilla, which itself is an open source software, is one of the most popular bug tracking systems in open source communities. It is used in Mozilla, GNOME, Netbeans, Jakarta Tomcat and so on (Halloran & Scherlis, 2002). It allows users to submit bugs with attachments which could be patches for bugs, test cases, screen captures and specifications. It also supports bug searching by its attributes. By integrating the email addresses, Bugzilla sends notifications to all users registered for a certain bug about requests on it and the bug’s progress (Reis & de Mattos Fortes, 2002).

Source configuration management systems (or referred as source code control, version control systems, etc.) are the most important and distinct technical support in OSS development (Yamauchi, et al., 2000). They coordinate the concurrent and parallel development of modules in open source communities. CVS (Berliner, 1990) is by far the most commonly used source configuration management system. Apache, Linux Kernel, Mozilla, Perl, and GNOME are all its adopters. It maintains a current central repository of source code along with multiple previous versions. It enables multiple developers working on the same code. Developers can take a local
copy of the current version of code by checking it out, make their own changes, and then commit it to the system by checking it back. Developers are also prompted to type a comment into the associated CVS log file about the changes they made when they are checking the code back (Fitzpatrick, Marshall, & Phillips, 2006). The role of CVS is expected to (1) provide a central consistent code base by a more formal coordination process than informal discussions, (2) allow users to see the status of a checked-out artifact and related code modules, and (3) highlight or resolve differences between versions (Froehlich & Dourish, 2004). There are also many other alternatives of CVS for version control in OSS development. For instance, BitKeeper (http://www.bitkeeper.com/) is one of the earliest alternatives to CVS. It is a commercial product that can be used to develop free software under a certain license. It does not include any group organization or communication tools. Another example is GNU Arch which is a completely non-centralized system. It lacks coordination facilities and communication tools. More examples and comparison of these systems can be found in Junqueira et al. (Junqueira, Sante, & Fortes, 2004).

Mailing lists are intensively used in open source communities as a complemented medium for coordination and communication. Neither high cost of learning to use it nor complicated technical infrastructure is required for implementing mailing lists in open source communities. Most of the informal discussions, for example, discussions about requirements and design, are supported and recorded by mailing lists. Normally there is no fixed rule for message construction. Thus the information is shared in various textual forms. Messages are often clustered based on topic or issue, forming semi-structured documentation upon problem solving process. All the members of open source communities have access to the community mailing list as long as they subscribe to it. They can broadcast any information to all the other participants, keeping them aware of what they themselves are doing (Yamauchi, et al., 2000).
<table>
<thead>
<tr>
<th></th>
<th>Traditional Software Peer Review</th>
<th>OSS Peer Review</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Types</strong></td>
<td>Inspections, walkthroughs, etc.</td>
<td>Unclear</td>
</tr>
<tr>
<td><strong>Criteria</strong></td>
<td>Software quality (Boehm, et al., 1976)</td>
<td>Project-dependent</td>
</tr>
<tr>
<td><strong>Process</strong></td>
<td>Planning, overview, preparation, meeting, rework, follow-up, and causal analysis</td>
<td>RTC, CTR</td>
</tr>
<tr>
<td><strong>Participants</strong></td>
<td>Inspection leader/moderator, Recorder, Readers, Author, Inspectors</td>
<td>Bug reporter, Contributor, Tester, Reviewer, Committer (P. C. Rigby &amp; German, 2006)</td>
</tr>
<tr>
<td><strong>Tools</strong></td>
<td>Checklists, Scenarios, Ad hoc reading, Computer-supported tools</td>
<td>Various computer-supported tools</td>
</tr>
</tbody>
</table>

Table 2-1: Comparison of traditional software peer review and OSS peer review.
Chapter 3

Linus’s Law: a Preliminary Examination of OSS Peer Review Processes

In this chapter, I present the preliminary results of my analysis on data from the bug tracking systems of Mozilla and Python. Adapting activity theory, I have identified the shared critical activities of their peer review practices and characterized each activity in terms of its three elements, participants, objects, and tools that mediate the activity (Nardi, 1996). The four common activities comprised of the OSS peer review process include submission, identification, resolution and evaluation. I discuss the distinctive features of open source software peer review compared to traditional review, as well as reconsiderations of Linus’s law.

Software Peer Review

Peer review is one type of software review for discovering defects (or bugs) as early as possible during development processes, suggesting improvements and even helping developers create better products (Wiegers, 2001). According to the “Standard for Software Reviews” (The Institute of Electrical and Electronics Engineers, 1999), software products that can be reviewed include requirements specifications, use cases, project plans, user interface design and prototypes, and program documentation.

Generally, peer review in traditional software development consists of planning, preparation (individual review), examination (review meetings), rework/follow-up, and exit evaluation (The Institute of Electrical and Electronics Engineers, 1999; Wiegers, 2001). In formal review, five types of roles are often engaged, the inspection leader/moderator, the recorder, readers, the author, and inspectors (Fagan, 1976). Despite disagreement on an exact number, studies mostly indicate a group of about 5 members is effective for software review.
One group of studies related to OSS peer review analyzes patch review (Asundi & Jayant, 2007; Nurolahzade, et al., 2009; P. Rigby, et al., 2008). We consider patch review as only part of OSS peer review activities because it only involves evaluation on a specific software product, patches. Therefore, I define OSS peer review as evaluation on any software products the project is producing, mainly the whole software application.

Another theme of OSS research investigates peer review processes, coordination mechanisms in particular. For example, Crowston et al. (K Crowston & Scozzi, 2004) applied coordination theory to four OSS projects, concluding the lack of task assignment in OSS development. They partly characterize peer review practices, the fixing activity. However, the evaluation related activities did not emerge in their study. More importantly, their perspective focused on personal interaction but overlooked the role of tools and their interaction with people.

The other body of OSS literature develops quantitative metrics to extract features of OSS projects, such as the work of Mockus et al. on Mozilla and Apache (Mockus, et al., 2002). However, this type of quantitative studies is constrained by their depth and descriptive ability.

**Case Selection**

We selected two established and well-recognized OSS projects for analysis, Mozilla and Python. Their success provides opportunities to access substantial peer review instances, particularly the ones illustrating effective practices. Multiple cases also enable contrasts that highlight consistencies.

Mozilla, operating as an OSS project since 1998, creates and maintains a variety of software applications, including the web browser Firefox, the bug tracking system Bugzilla, and the software detecting tool Tinderbox. The Mozilla community is organized and supported by the non-profit organization, Mozilla Foundation. Tools supporting peer review mainly consist of
Mercurial (version control system), Bugzilla (bug tracking system), Internet Relay Chat (IRC), and mailing lists (identical to discussion forums in Mozilla).

Python, initiated as an OSS project in 1991, develops the computer programming language Python. Similar to Mozilla, Python community is supported by a non-profit organization, Python Software Foundation (PSF). Tools currently employed for peer review practices include Subversion (version control system), Roundup issue tracker (bug tracking system), mailing lists, Python Enhancement Proposals (PEPs), Internet Relay Chat (IRC), and Rietveld (code review tool).

**Data Resources**

We retrieved bug reports of Firefox and Python language from the bug tracking systems of Mozilla and Python for our current analysis. Bug tracking systems are crucial data sources of peer review practices in large OSS projects. They keep track of reported defects or deficiencies of source code, design, and documents. Sampling from two core products, Firefox and Python language, helped focus our investigation. Furthermore, the difference of users’ technical expertise between two products may contrast peer review practices between projects: Firefox is an end-user application, while Python language is used by programmers.

We collected bug reports created between two recent stable releases of Firefox and Python for our quantitative examination. For Mozilla, reports were filed between the releases of Firefox 3.5 final on June 30, 2009 and 3.6 final on January 21, 2010. For Python, issues were generated between the releases of Python 3.0 final on December 3, 2008 and 3.1 final on June 27, 2009. This sampling strategy was intended to capture the possible behavioral changes near releases, as suggested by Francalanci et al. (Francalanci & Merlo, 2008).
Our qualitative analysis was carried out through two phases. We first explored our data set randomly to identify the critical activities and their elements of peer review practices. The findings also served as our first-level code. Then I selected reports with diverse traits of activity elements, such as different status, resolution and roles of participants, to refine our understanding of each activity.

Other data resources leveraging our study are online documents, such as projects’ review instruction and policy statement (MozillaFoundation; PythonSoftwareFoundation, 2010). We also identified core members of these two communities from their websites and wikis. For Mozilla, they include Firefox and Toolkit module owners and peers, super reviewers, security group members, and quality assurance (QA) members. For Python, committers, PSF officers, board of directors, nominated members, and emeritus members are counted.

**Critical Activities of OSS Peer Review**

Four critical activities of peer review practices emerge from our qualitative analysis of the two cases. A complete peer review may proceed along different sequence of these activities. Thus, the activity view offers more flexibility than a process view. Besides, I only discuss bug tracking systems as the tool of each activity for our current study. At the end of the section, Table 3-1 summarizes our findings.

**Submission**

Peer review process starts with a community member experiencing a failure, discovering a product defect, or desiring design enhancement. Any community members can submit a report for such issues they encountered. An issue can either describe what happened or what is expected.
The goal of submission is to evoke community’s awareness of an improvement request. To some extent, submission functions as individual review at the preparation phase of traditional software review.

Failures of using software are common incentives of participating in peer review.

“Cannot drag a page link to an empty bookmarks toolbar” (Bug 521429; Mozilla).

“[M]ap.move crashes by integer overflow” (Issue 5387; Python).

Content or design defects are also one type of submissions that are generated from members’ examination.

“[R]emove unneeded CSS and image data for toolbar buttons that are always enabled” (Bug 510442; Mozilla).

“[U]rlib: urlretrieve() does not close file objects on failure” (Issue 5536; Python).

Some submissions explicitly stated their requirement of new functionalities.

“Feature request: User controlled width of containers” (Bug 514538; Mozilla).

“Extend subprocess.kill to be able to kill process groups” (Issue 5115; Python).

Transforming an individual experience, discovery or demand into a submission for public attention requires a clear and informative description. Common elements of a bug report include a concise summary of the problem, the environment of the occurrence (e.g., operating systems and software components), steps to reproduce the problem, and the version of the reviewed product.

Large OSS projects often employ bug tracking systems to receive submissions. These systems present a semi-structured form for reporters to fill, with both required and optional fields. Bugzilla and Roundup also assist reporters with notes explaining the meaning of some fields.
Identification

Identification is an activity in which members detect the cause of product defects or determine the specifications of requested improvements. Compared to traditional software review, this activity is similar to review meetings but in an asynchronous manner. Agreement on whether the bug should be fixed is also expected when there is a discussion. Depending on the volume of submissions, an identification activity may involve more than one step: a large number of bug reports need screening before any further analysis.

Preprocessing

Preprocessing examines whether the submission discloses an actual defect of the product or a suitable request of new features. Some projects use the term *triage* in a general sense to refer to this stage. Several types of bug reports are filtered during this stage, including (1) failures that are not reproducible, (2) failures that are caused by third party applications or reporters’ mistake, (3) issues that are intended design, (4) issues that have already been reported, (5) issues that maintain the state of lacking indispensible information, and (6) requests contradicting to project objectives or raising risk of system security. Requests that seem plausible but depart from project’s short-term plan are suspended. Screening those bug reports should not be controversial but self-evident.

Active contributors usually mediate the screening process. They follow up reporters with questions that clarify the ambiguity in the bug description, solicit missing information, or guide the reporter to re-test the problem.

“Can you reproduce this problem after you change ...” (Bug 527853; Mozilla).

“Do you happen to have a ... environment variable? If yes, I suggest to remove it” (Issue 5528; Python).
Bug tracking systems enable qualified users to set flags indicating whether an issue needs further efforts. Both Bugzilla and Roundup use works for me, invalid, duplicate, and won’t fix to classify filtered issues. Members need to request privileges that grant modifying these flags. Although contribution is necessary for obtaining this editing privilege, neither Mozilla nor Python is conservative at offering permission.

To support identifying duplicate issues, bug tracking systems implement search features. Words in report text are the most common index for retrieval. Bugzilla and Roundup also allow search by bug ID, creation date, change date, status, resolution, creator, involved members, assignee, version, component, priority, and related bugs.

**Identification**

Bug reports not sorted out at preprocessing deserve further identification. When a defect is trivial and its cause is self-evident (e.g., a typo in a document), additional analysis is not necessary. Otherwise, further investigation may refine bugs’ classification. For example, during analysis members may realize the cause of failure is identical to an existing bug’s but just another symptom. Or they may find out the deficiency is beyond project goals. Identifying relationship between bugs, if any exists, is also expected. Awareness of bug connections helps coordinate resolving them.

Collaboration often takes place in identification activities, with both upsides and downsides. More “eyeballs” participating usually facilitate detecting problem causes. However, a larger group may face the challenge of reaching consensus, especially when the issue is a feature request. Under such circumstances, the group sometimes integrates both perspectives into a compromise. Regarding “[a]dding an option of non-zero exit status of setup.py when building of extensions has failed”, some members favored the suggestion, while others argued that users
might mistake the failure of compiling extension for the incorrect implementation of Python.

Finally, an agreement was reached.

"...add an option where you tell exactly which modules aren't expected to be compiled successfully...If any modules besides those specified are not built, then setup.py returns [failure]" (Issue 6731; Python).

Sometimes the group consults the core developers who have decision power on the specific issue (e.g., owners of the affected module and the benevolent dictator). One reporter criticized that “[f]avicon don't work unless using 'Remember' privacy setting” (Bug 502787; Mozilla). Two core members inclined to set the bug as Won’t fix. After the reporter elaborated the harm of this defect, they directed the issue to the module owner for decision, “I'll leave this decision up to the module owner then...”.

Some other times the group appeals to the rest of the community (e.g., posting the issue to relevant mailing lists). One member requested to “[a]dd a 'swap' method to list” (Issue 6326; Python). A core developer concerned that this feature might generate complicated effects. Thus he suggested that “[t]here are clearly enough subtleties to this proposal that it should be discussed on python-ideas first”.

The reporter, report reviewers, and module owners are three major roles in the discussion phase of an identification activity. The reporter should be available to provide additional information when queried by reporter reviewers. Report reviewers need to be knowledgeable of the affected components. They offer insights for assessing the defect; they may also mediate the discussion by involving the owners and relevant parties in decision-making.

Bug tracking systems provide each bug individual space for such discussions and display them on the same page with other reported information. They also save the history of all the actions taken on a bug report. This history records what decision was made by whom at what time. Additionally, the systems present links to related bugs.
Resolution

Similar to the rework step in traditional software review, resolution in OSS peer review addresses the identified defects or requests. Accurate and clear identification of defect causes or request specifications can facilitate the resolution process. OSS projects often implement the resolution as patches to their software product unless the repair is very trivial.

Patches sometimes are submitted by the bug reporter at the time of submission; otherwise, other members may take the responsibility of creating a patch voluntarily. Even though members can assign a specific developer to fix a bug, OSS communities encourage self-assignment: developers can choose what to work on and when to quit by their own interests. In one of Python’s issues, some core members conveyed this point explicitly:

“I'm a bit worried about committers giving other committers priorities on how they should work; as a principle, volunteers are free to work on whatever they please, in whatever order they please...So unassign [developer 1]” (Issue 4614; Python).

Similarly, a Firefox developer assigned another developer a bug without permission, which confused the assignee.

Developer 1: “[Developer 2] seems to know what the issue is (if there is one), so assigning to him”.

Developer 2: “As I say, I can't reproduce. There's no bug here AFAICT” (Bug 516828; Mozilla).

Parallel implementation of a patch rarely happened in Mozilla and Python. Members submitted a patch shortly after identification discussion and set themselves as assignees. If they needed a longer time to start implementation, they often explicitly claimed the implementation responsibility by changing the value of field “assignee” in bug reports, or stating it as a comment. For example, a developer decided to fix a bug after he confirmed its suitability. He indicated his decision by marking himself the assignee of this bug.
“Well, I think reformatting the about page could make some sense, so taking for now” (Bug 502228; Mozilla).

Contributing a patch usually requires sufficient expertise and knowledge of the affected software components. Consequently, core developers are intensively engaged in resolution activities, although they are not obliged to. In comparison, the rework duty completely falls onto the software author in traditional software review.

Both Bugzilla and Roundup support designating assignees and uploading patches as attachments to a bug report. Currently they both constrain the access to editing assignees within a group of authorized members. Discussion spaces reserve all the suggested approaches and specified requirements generated during discussion.

**Evaluation**

Evaluation is to ensure a resolution successfully repairs identified defects or satisfies specified requests, similar as follow-up to rework in traditional review. Patch review is the most common type of evaluation. If an implementation fails evaluation, resolution activities, and even identification activities will be reinitiated.

Evaluation is based on not only whether a resolution can perform as expected, but also maintainability, security, integration, usability, test, license and so forth. The following excerpt illustrates reviews on design conventions, behavior, styles, and testing.

“We've beginning to use gDelay=0 ... as code conventions. This'll matter a down the review...[t]his part is failing for me when running the testscript via command line (on the assertProperty function)...[o]therwise, this looks good. Short and sweet, the way I like it :)” (Bug 506325; Mozilla).

“[Y]ou need to indent the ‘else’ clause after you test for ignore_cookie...I'd like to see a test that shows that byte strings still have their cookies examined” (Issue 4626; Python).
Reviewers tend to let a resolution pass review if only minor changes are needed. This lenient convention encourages novice contributors, and expedites the cycle of product improvement.

“r=me with the last few changes talked about on irc” \(^3\) (Bug 534398; Mozilla).

“The patch looks about right...So please remove the note on releasing md_state.” (Issue 4614; Python).

Other than the resolution author, at least one core member has to participate in evaluation activities. Peripheral members can suggest any modification, but are not eligible to decide the evaluation result. When implementations may affect multiple modules, or the core developer is not confident to make a decision, more core reviewers will participate in the evaluation. For instance, Mozilla has the super review policy, which applies to significant architectural refactoring, any change to any API or pseudo-API, and all changes that affect how code modules interact. Python requires changes to a release candidate or a final release must be reviewed by two core developers unless the patch author is a core developer (PythonSoftwareFoundation, 2010). Both projects indicate that the resolution author should be responsible to keep pursuing a review when no timely response is received. Overall, Bugzilla and Roundup both support viewing patch content, notifying specific reviewers by adding email addresses, and displaying evaluation decisions.

Table 3-1: Elements of OSS peer review critical activities.

<table>
<thead>
<tr>
<th>Activity</th>
<th>Participant</th>
<th>Object</th>
<th>Tool</th>
<th>Traditional Review</th>
</tr>
</thead>
<tbody>
<tr>
<td>Submission</td>
<td>Any member</td>
<td>Experience, request</td>
<td>Semi-structured Form</td>
<td>Preparation</td>
</tr>
<tr>
<td>Identification</td>
<td>Moderator, any member with some knowledge</td>
<td>Bug reports, query responses</td>
<td>Search, discussion space, bug relation visualization</td>
<td>Examination</td>
</tr>
<tr>
<td>Resolution</td>
<td>Any member</td>
<td>Defect causes, request</td>
<td>Awareness of</td>
<td>Rework</td>
</tr>
</tbody>
</table>

\(^3\) Note that Mozilla uses “r=” to mark patches that pass review.
<table>
<thead>
<tr>
<th>Evaluation</th>
<th>Core developers, author, any other member</th>
<th>Implemented/revised resolution (e.g., patch)</th>
<th>Reviewer notification, patch viewer, discussion space</th>
<th>Follow-up/exit evaluation</th>
</tr>
</thead>
</table>

**Discussion**

The four common peer review activities identified from Mozilla and Python—*submission, identification, resolution* and *evaluation*—share similarities with four steps of traditional software review process, *individual reviews, review meetings, rework* and *follow-up*. However, they also show distinctive features of OSS peer review practices. (1) Products in review are constantly changing because open source code repository is updated on a daily basis. Compared to a static context faced by a traditional inspection team, OSS peer review participants need to maintain awareness of product status, eliminating submissions of bugs that no longer exist in current build. (2) Review is conducted by thousands of different groups, each focusing on a single issue. This scales down the product being reviewed, decreasing work complexity and chances of errors. (3) Individual reviews are initiated by utilitarian incentives. Inspectors in traditional review examine products by reading line by line, using techniques like checklist. By contrast, OSS reviewers are bug reporters, who perform reviews whenever experiencing unexpected behavior of software applications. This motivates wide participation, but may also increase other participants’ difficulty in sense making of defects. (4) Rework is shared responsibility. Traditional software review confines discussions upon defect resolutions during review meetings, leaving the rework responsibility to the product author. The affected OSS module owners, equivalent to product authors, are not obliged to implement resolutions. This may facilitate knowledge development within communities, but may also increase the risk of no
rework done. (5) Review consensus is based on meritocracy. Moderators of traditional reviews usually make decisions involved in review meetings and follow-up. OSS peer review values the opinions of product authors and the reviewers who can offer resolutions.

Activity characteristics urge reflection on Linus’s Law. Compared to traditional software development in which review teams often consist of fewer than ten developers, OSS development does benefit from hundreds or thousands of reporters. A large number of defects or improvement requests are reported. Each single defect is analyzed by different composition of participants, integrating diverse expertise and facilitating knowledge sharing. However, many “eyeballs” may also increase the cost of peer review, particularly when they are from inexperienced users. For example, Mozilla had to spend great efforts on screening. Even with thousands of “eyeballs”, a number of reported bugs still remained in identification activities with limited attention in both Mozilla and Python. This indicates eyeballs cannot be enough without awareness support for appropriate experts.
Chapter 4

Creative Collaboration in Open Source Software Peer Review

Extending the analysis reported in the chapter 3, this chapter focuses on creative collaboration that emerges from the OSS peer review process. Creativity can be evoked from peer review collaboration—collectively generating new and useful solutions to improve software quality. Creative collaborative peer review is especially important when bugs are difficult to fix. We show how peer review practices are unavoidably creative by studying Mozilla and Python. Taking a creative process view, I identify four common subprocesses of peer review process, problem identification, preparation, solution generation, and solution evaluation. For each subprocess, I characterize creative collaborative peer review interactions, and discuss the enablers and obstacles of the creative collaboration. We further recommend design implications to enhance creative collaborative peer review processes, including support for establishment of common ground, externalization of social networks, awareness of resolving progress, and articulation of design rationale. These implications may also apply to other social organizations and software development collaboration.

Open Source Software Peer Review Practices

In general, OSS peer review processes begin with one submitting a bug report to bug tracking systems—applications that help developers keep track of reported defects or deficiencies of source code, design, and documents. Others examine the defect cause or request specification, determines whether the bug should be fixed, and commits a change set (mostly a patch) to the current software product. Version control systems manage and synchronize such product changes,
while communication tools such as mailing lists and Internet Relay Chat (IRC) enable developers discussing bugs.

Crowston et al. (K. Crowston, 2008) characterized OSS peer review practices as three sequential steps, submit, fix and close, by analyzing four OSS projects from SourceForge. With the focus on coordination mechanism in peer review practices, they concluded that OSS development lacks traditional coordination mechanism such as task assignment.

Mockus et al. (Mockus, et al., 2002) generated hypotheses that a group larger than the core by about an order of magnitude contributed to defect fixing, and a group larger by an order of magnitude again contributed to problem report, based on their quantitative case studies in Apache and Mozilla. Their study provided insight of contribution distribution in OSS projects; however, the practices remain unexplained. Besides, their analysis on Mozilla was conducted before the first official release of Firefox.

Some recently studies concentrate on the participation and quality of bug reporting. Recent work by Ko et al. (Ko & Chilana, 2010) analyzed the reports of Mozilla contributors who reported problems but were never assigned problems to fix, indicating that the primary value of open bug reporting is in recruiting talented reporters. Other literatures suggested ways to improve bug reports (N Bettenburg et al., 2008; Breu, et al., 2010).

Other related work focuses on patch review, which is part of the peer review practices. They are merely statistical reports (Asundi & Jayant, 2007; P. Rigby, et al., 2008), or only described the policy statement (J. P. Johnson, 2006), or developed quantitative measurements (Barcellini, et al., 2005).
Creative Collaboration

Creativity theories suggest that creativity is not just about personal traits and products, but also about the problem solving process. Guilford (Guilford, 1950) analyzed creativity as arising from divergent thinking throughout the process of problem solving—preparation, incubation, illumination, and verification. Subsequent research on individual creativity proposed variations of this stage-based view, for which Lubart (Lubart, 2001) provided an excellent review. Common components of those various process models include problem finding and construction, divergent thinking and information synthesis, and idea evaluation.

Expanding from individuals to groups and organizations, creativity researchers identified several factors that affect creativity, such as intrinsic motivation (Amabile, 1996), diversity (Milliken, Bartel, Kurtzberg, Paulus, & Nijstad, 2003), task conflicts and minority dissent (D. Johnson, Johnson, & Tjosvold, 2000), shared objectives and reflexivity (West, 2002), open information sharing (Woodman, et al., 1993), and affect (Amabile, Barsade, Mueller, & Staw, 2005). However, much of the work has focused on business organizations but not volunteer/community organizations. Furthermore, their implications to computer-mediated collaboration still need articulation.

Method: Comparative Case Studies

We addressed our research questions by conducting case studies in Mozilla and Python, which are both established and well-recognized as creative. Other than commonalities, their differences enable contrasts that highlight consistencies (Yin, 2008).

Both Mozilla and Python communities are supported by non-profit organizations. Community members include both core developers who contribute significantly to the project and
peripheral members who use the software but rarely participate in developing the software. Both communities have Benevolent Dictator for Life (BDFL) to make final say over technical disputes.

Another important commonality between Mozilla and Python is their computing infrastructure that supports their peer review activities, including version control systems (Mercurial for Mozilla and Subversion for Python), bug tracking systems (Bugzilla for Mozilla and Roundup for Python), IRC, mailing lists and wikis. For code review, Bugzilla implements a patch reviewer tracking code changes, while Python recommends an external code review tool Rietveld. Additionally, Python creates Python Enhancement Proposals (PEPs) for initial design ideas.

Mozilla and Python differ at their product types, sizes, community scales and structure. Mozilla is a larger community creating various applications. One core application, Firefox, is an end-user application. In contrast, Python develops a computer programming language by a relatively small and homogeneous community. The roles of core members other than the BFDL are defined in slightly different ways between Mozilla and Python, which are particularly emphasized in patch review activities. Mozilla divides responsibilities of its core members into module owners, module peers and super reviewers. Each individual has specified areas to maintain and facilitate evolvement. For example, Firefox has a module Session Restore, which is supported by one module owner and three peers. Any changes toward this module have to be reviewed and approved by any of them. Peers not only help reduce the owner’s workload, but also improve decision quality, correcting the owner if necessary. Python, to our best knowledge, classifies its core developers in a relatively loose fashion. The core members share equal responsibilities and privileges over the software. They are elected annually on the basis of their contributions.
Data Collection and Analysis

Our current analysis draws on bug reports of Firefox and Python language from the bug tracking systems of Mozilla and Python. Bug tracking systems are one of the primary data sources of peer review practices in large OSS projects. Sampling from two core products, Firefox and Python language, helped focus our investigation.

We retrieved bug reports created between two recent stable releases of Firefox and Python for examination. This sampling strategy was intended to capture the possible behavioral changes near releases (Francalanci & Merlo, 2008). For Mozilla, 7322 reports were filed between the releases of Firefox 3.5 final and 3.6 final. For Python, 1850 issues were generated between the releases of Python 3.0 final and 3.1 final. Conjectured from organizational documents, we identified 211 core members of Mozilla and 218 of Python.

Our analyses iteratively explored the collected reports, with reflection on relevant literature. First, we randomly browsed the data set to understand the basic schema of bug reports in Bugzilla and Roundup. We identified several indicators to guide our following exploration, including bug status, resolution, number of participants and comments, whether attached a patch, reporter identity, and assignee identity (core or peripheral member). Then we performed statistical analysis over the entire data set to get an overview of the collaboration characteristics. Second, we employed a creative process perspective to investigate the peer review practices qualitatively, focusing on subprocesses instead of a sequential model. We stratified the reports using bug status and resolution, and then for each status and resolution we sampled reports with different values of all the other indicators identified at the first step. For example, among bugs that were resolved (status) and fixed (resolution), we examined reports submitted/assigned by core and peripheral members, involved very few and a large number of participants/comments, and implemented with or without a patch. Such an approach was intended to accommodate the
process variation across bugs. Other than considering indicator values, we did not stop including more reports for qualitative analysis until being able to answer our research questions (Glaser & Strauss, 1968), eventually ending with 68 Firefox bug reports and 44 Python issue reports. In addition, we triangulated our data with other sources, such as project policy statements, source code repository, mailing list threads, and PEPs.

**Results**

Collaboration is common among the processes of fixing every bug, even though some bugs may not involve any participant other than the reporter. On average, 2 Mozilla members participated in discussing one bug \(\text{median}=2; \text{skewness}=6.765\), contributing 3 comments in total \(\text{median}=3; \text{skewness}=14.305\); 3 Python members contributed to each issue discussion \(\text{median}=3; \text{skewness}=3.007\), generating 4 messages \(\text{median}=4; \text{skewness}=5.669\).

Our analysis also indicates that collaboration is crucial, to some extent, for a peer review process to accomplish its goal. By the time of our retrieval, 498 bugs of Mozilla Firefox were fixed, while 678 issues of Python were either fixed or accepted. Nonparametric tests (Mann-Whitney U Test) indicate that fixed bugs involved relatively more participants and comments. The difference is more significant in Mozilla \(p_{\text{participant}}<.001, p_{\text{comment}}<.001\) than in Python \(p_{\text{participant}}=.054, p_{\text{comment}}=.088\).

Adapting from previous literature on creativity, we characterize peer review practices into four subprocesses, problem identification, preparation, solution generation, and solution evaluation. Each bug can be fixed along different sequences or combinations of these subprocesses. Subprocesses are not perfectly clear-cut; some of them may be intertwined at times. We discuss each subprocess in response to our three research questions in next sections.
Subprocess 1: Problem Identification

As in general problem solving activities, the problem identification process in peer review practices consists of problem finding and construction. Moreover, it involves different stakeholders judging and negotiating whether the problem accords with community objectives and values. Specifically, a problem is found and described by an individual member experiencing a failure, discovering a product defect, or desiring a design enhancement. It is then elaborated, understood, verified and sometimes reframed through community effort.

RQ1: Characterizing Interactions in Creative Collaboration

One collaborative activity that often occurs in the problem identification process is community members establishing shared understanding of the problem and its causes. The problematic situation is hard to be completely presented in the shared space but only available to the bug reporter. Other members willing to solve the problem have to elicit information cues through interactions with the reporter, especially when the bug is difficult to fix. A typical strategy is to require reporters describing Steps To Reproduce (STR) when submitting bug reports, as Bugzilla does.

A bug regarding “clipboard data added in Private Browsing Mode causes Windows Explorer to crash when accessing menus which include Paste after closing firefox” specified the steps to reproduce the problem as “1. Close Firefox; 2. Right-click the desktop or Recycle Bin” (Bug 518412; Mozilla).

If information provided in a bug report is not sufficient for developers to approach the problem, they will query the reporter or ask the reporter to run some tests locally.

“Can you create steps to reproduce the issue in a new profile so that we can try to reproduce it from scratch?...” (Bug 536081; Mozilla).

“Do you happen to have a ... environment variable? If yes, I suggest to remove it” (Issue 5528; Python).
Collaboration provides opportunities to frame problems from multiple perspectives or derive more issues that may bring more software improvement. For example, some rejected bugs inspired other members to file new ones that reframe original problems into the ones suitable for a fix. A Firefox user criticized the behavior of “Not prompted for session restore after force-quitting due to lock-up” (Bug 501995; Mozilla). One core developer decided “Closing this bug as INVALID, as the described behavior is by design.” Another developer came up a scenario this design can be improved and filed a new bug, “the current settings and behavior is just fine for the case where there are few tabs or windows, but it is not as precise as it needs to be for the case where many tabs and windows are open at the time of last crash. Under those conditions start up after a crash or problem is quite jarring and the user is pretty much out of control.”

Another collaborative activity in identification process is to determine whether a bug should be fixed. It usually happens when the effects of changing a specific product have high uncertainty or will result in conflicting interests among different stakeholders. Decision-making rules vary across OSS communities. When consensus is difficult to achieve, the final decision is an individual choice by the owner in our cases, such as module owners in Mozilla and BDFL in Python. One reporter criticized that “[f]avicons don't work unless using 'Remember’ privacy setting” (Bug 502787; Mozilla). Two core members were inclined to set the bug as Won’t fix. The reporter argued the harm of this defect, “… under the logic that this is a violation of your privacy settings because you're telling the browser not to save your history, creating a bookmark at all is in violation of the privacy settings under the same rationale.” Then the core members directed the issue to the module owner for decision, “I'll leave this decision up to the module owner then…” and the module owner decided to accept the change request, “I agree with the reporter. If the user asks for a bookmark, we should store it's favicon, regardless of the history setting.”
RQ2: Enablers of Creative Collaboration

In contrast to ones implying an explicit solution like document typo, problems that are open to various perspectives and even incur risky changes can encourage experimentation and exploration of new opportunities. Some of the problems aim at improving current software performance, and some others are adding new features.

“Optimize bytecode for conditional branches” (Issue 4715; Python).

Another example from Mozilla is that one Firefox user suggested considering adding “undo” option for installation of personas. Core developers came up with three strategies to satisfy this request (Bug 518468; Mozilla).

When the involved people have no experience or limited knowledge of a bug, they reach out to engage more collaborators from the community for informed decisions, rather than committing to a conclusion early. For instance, a Python user raised question “Shou[ld] argparse be included in the Python Standard Library” and implied his support for adding this feature (Issue 6247; Python). The original code author supported the reporter, “… the optparse extension API is horribly designed, and … it’s nearly impossible to add any new features to optparse without breaking this”, but another core member was reluctant to accept the request, suggesting “It would be useful if several people could confirm that argparse is *not* horribly designed”. Several other members expressed favoring accepting the request from users’ perspective, while a developer concerned that “Do not add anything to the stdlib that is not needed to keep applications platform independent. argparse adds zero value to x-platform development.”
RQ3: Obstacles of Creative Collaboration

Failing to build common ground regarding a bug can inhibit its fixing processes. Participants in the identification subprocess hold heterogeneous knowledge of the problem—the reporter has the knowledge of the local environment, while other developers have technical expertise to discover the cause of software dysfunction. However, collaborative identification cannot progress until that unique information and knowledge is effectively shared, clarified and confirmed (Nickerson, 1999).

In our cases, sometimes collaboration broke down because of no response to developers’ queries from bug reporters. This is not unexpected: OSS community members, particularly peripheral ones, voluntarily participate without any obligation or commitment. For example, 489 Mozilla bugs in our data set (6.68%; n=7322) were suspended because of incomplete information. Both Bugzilla and Roundup specify items for submitting a report, which helps reduce such occurrences to some extent.

Disagreement on deciding whether a bug should be fixed sometimes results in conflicts, emotional but not informative. Insult arises from the fight; some members may even drop out of the community. For instance, end-users sometimes have unresolved conflicts with developers because they feel their interests were not appreciated. For example, Firefox developers decided to get rid of the “Properties” context menu item even though several users argued this item had been useful to them (Bug 513147; Mozilla). Tens of Firefox users complained. One said, “Sometimes a group with a very closed perspective hijack it. Congrats on making [Firefox] a laughing stock of a browser.” However, all the core members refused to reverse their decision. The users escalated the debate, continuing commenting at an external web site, suggesting “sounds like a good time to switch to [another browser].”
Subprocess 2: Preparation

Preparation process involves collecting and reactivating relevant information and resources for generating solutions. In OSS peer review practices, members actively search and gather information such as who has the best knowledge of defective source code, what are the current technical constraints, and what methods may apply.

RQ1: Characterizing Interactions in Creative Collaboration

Interactions often occur when members are not capable or available to fix a problem by themselves, appearing as instructional or help-giving. For instance, Firefox for different operating systems requires different domain expertise and knowledge, which is not widely accessible. Developers have to consult experts when they are not familiar with the system where the problem happened. In Mozilla, a developer requested “… [creating] an entry [in the Win7 new aero peek taskbar menu] for each of the currently open windows, and then also, if there's room remaining, entries for specific tabs within the windows” (Bug 522035; Mozilla). A participant explained the design of the aero peek taskbar menu, “There is no notion of recent windows. We have limited control over the ordering but the tabs will appear in the correct ordering (that is, left to right) and the actual positions are determined by the time that we made the preview.”

Interactions sometimes are not initiated by asking questions but voluntarily introducing external resources, such as providing pointers to information outside the community to facilitate analogical thinking.

“Prelude has had the same problem with signal 32: https://dev.prelude-ids.com/issues/show/133. According to their research, it is due to the linuxthreads implementation of the pthread API.” (Issue 4970; Python)
RQ2: Enablers of Creative Collaboration

Creative collaboration requires appropriate domain knowledge and expertise (Amabile, 1996). The most straightforward way to engage experts is probably to ask the original author of the misbehaved code for help, especially when solving it is beyond others’ capabilities. For example, for regression bugs (i.e., software stops functioning as intended after a change is integrated), members usually trace down the latest contributors of suspicious code changes and sent request to them.

“[M]ost likely regression from bug 244371. [Patch author of bug 244371] could you take a look at this?” (Bug 516088; Mozilla)

Similarly in Python, a member detected the failures in test_xmlrpc was due to a commit from developer 1 and forwarded the issue report to the developer. The developer explained the cause of the bug.

“Oh dear, this is because it skips those tests on windows.” (Issue 4939; Python)

The extent of information search also has impacts on creative performance of preparation process. A collective is more creative when they continue exploring other possibilities instead of settling for an existing satisfactory but suboptimal solution or method (Amabile, 1996; Simon, 1957). For Bug 506482 in Mozilla, after a long time of no progress, a peripheral member proposed “... that you add the option of not triggering the sessionstore just for "read-only" events (scrolling a page or selecting a tab).” to the bug “Constant rewriting of sessionstore.js creates excessive disk activity” The core member who was going to take the bug welcomed the contribution, scoped the problem down to “Don't write sessionstore.js to disk for "read only" events”, commenting that “[developer 1], [developer 2] & I talked about this and think this is a good bandaid, but we'll need to do something more ... for the long term.”

The following quote shows a similar situation that occurred in Python.
[Developer 1]: “Since linuxthreads is no longer maintained, I'm afraid we can't do much about this, except check for the threading library used and say "linuxthreads is obsolete and has known issues - please upgrade your system for reliable threading support".”

[Developer 2]: “But I think it's still worth trying to narrow down (and possibly work around) the cause of failure, just so that we can make this buildbot useful again on 3.x.” (Issue 4970; Python)

**RQ3: Obstacles of Creative Collaboration**

Directly asking the author of “culprit” only works when someone in the discussion knows who the desired expert is. However, peripheral members normally lack such knowledge. Core developers have such knowledge but they can hardly participate in every bug discussion because of heavy workload.

Relying on a single person (e.g., the original code author) can also be risky as a result of the dynamic coming and going membership of open source communities. The person may be busy or no longer active in the community.

“[Developer 1] fixed this the last two times it happened (bug 491530, bug 465910), so he might be interested in helping to fix it again.” “... [Developer 1]’s kind of out of pocket...” (Bug 539120; Mozilla)

Under the circumstances in which no one in solution preparation discussions can locate an appropriate expert or no single person has the perfect knowledge of a qualified approach, collaboration cannot continue without raising broad awareness in the community. Python Roundup issue tracker does not have specific features for this purpose, while Mozilla Bugzilla users sometimes highlight the request in its whiteboard field of report summary.
**Subprocess 3: Solution Generation**

During the solution generation process, candidate responses to a problem are produced. In our context, developers propose approaches and implement them mostly as patches to resolve the bug.

**RQ1: Characterizing Interactions in Creative Collaboration**

In Mozilla and Python, collaboration on generating bug solutions usually exhibits as contributing design ideas, or specifications for implementations, or sharing their implemented solutions. A group of developers working on different parts of the implementation, on the contrary, rarely happens. This reduces the interdependencies between members, conducive to effective coordination and collaboration. Divergent thinking and convergent thinking may both be involved before a solution is implemented as shown by the quotes below.

>[Developer 1]: “I don't think we want to move away from it being a localizable pref... What we could do for the moment is specify the pref in...”

>[Developer 2]: “Would it be simpler to move this file into a jar?” (Bug 524201; Mozilla)

A Python developer had created a patch but did not revise it as the reviewers suggested. After a long time of no response, another developer created a new patch with a different method.

>“Attached is a patch I wrote entirely independently of this issue ... I go further in terms of caching items than the original patch as I still got failures ...” (Issue 5099; Python).

Compared to the public discussion space offered by bug tracking systems, synchronous communication media, such as IRC or even face-to-face interaction, are preferable for developers to discuss technical details. Such personal spaces can enhance participants’ experiences and safety feeling, as well as allow responsive and efficient interactions with appropriate audience.
A touchstone characteristic of creative groups and organizations is that they produce diverse ideas (Nemeth, Nemeth-Brown, Paulus, & Nijstad, 2003). In Mozilla and Python, bugs may receive various suggestions of solutions if they can attract many participants, especially developers. After exploring possible solutions, how to converge on the final solution design and put it into action (i.e., implementation) determines whether the collaborative solution generation process can be accomplished. Synthesis and integration usually requires members augmenting the virtues and attenuating the costs of solution alternatives. This cannot be achieved unless tradeoffs of each alternative have been comprehensively speculated.

“Inspired by this latest mockup I did a PoC for something similar: ... This way, clicking on the thumbnail would just immediately switch it to the window, and clicking on the button shows all the tabs.” (Bug 522035; Mozilla)

“We can reach a middle ground by ... This will address most of the issues that I raised and ... is simple enough to write as long as you don't have to worry about sharing utc instance between modules.” (Issue 5094; Python)

Awareness of the progress of implementing solutions is critical to facilitate generation process. Community members do not have obligations to implement a patch; however, clarifying what should be anticipated can help other members plan their subsequent actions.

“This is a work in progress. I openly admit this isn't good code, but it's something of a start. ... I will come back to this patch sometime in the future, but if someone wants to take this over before I can, please do.” (Bug 515549; Mozilla)

“I did not finish the memoryview implementation, due to other demands on my time.” (Issue 4580; Python)

Awareness of the status of interdependent bugs is also crucial to solution generation. Despite efforts to achieve modularity in OSS, dependencies among changes of source code cannot be entirely eliminated. Bugzilla has both fields “Depends on” and “Blocks” to keep track
of related bugs. Similarly, Roundup uses “Dependencies” and “Superseder” to represent issue relationships. Participants in one bug sometimes also contribute to solving related bugs.

**RQ3: Obstacles of Creative Collaboration**

Not informing the developmental status, the solution generation process can hardly proceed and a bug will stay open for a long time. Once someone claims to take a bug, the community will have expectations from the individual and his/her accountability is established. One core developer of Python asked a patch author to change the patch, and explained the reason for the change in response to the author’s argument. “memcpy won’t work if the data type changes... When the datatype is the same, memcpy() should be used.” (Issue 5109; Python). For over a year, the patch author did not update the patch or post any comments.

Sometimes the informal conversations via synchronous communication channels may also cause loss of design knowledge since the community does not record the content of such interactions.

“[Developer 1] and I were talking on irc and thought that this would be simpler than using a pipe. I don't fully recall the reasoning (we looked at three or four options all together). For a more general case, a pipe might be better.” (Bug 522416; Mozilla)

“I had originally wanted to do something like that but [developer 2] at the time thought it was not the right approach. I don't recall the details. Maybe [developer 2] can recall. I think we had that conversation in person and it was a long time ago :(" (Issue 5094; Python).

Interestingly, some volunteers in the Mozilla community developed and are maintaining a quote database of Mozilla IRC conversations. This, to some extent, can alleviate the knowledge loss (e.g., design rationale).
**Subprocess 4: Solution Evaluation**

Solution evaluation process regards to the examination and validation of generated solutions, as well as the decision on integrating solution into current software products. In our context, we refer to the process when other community members critically review solution implementation to the defective software product rather than proposed ideas. Evaluation criteria include not only whether a resolution can perform as expected, but also other general principles in software engineering, such as maintainability, security, and usability, as well as special standards developed by the project core team, such as writing format.

**RQ1: Characterizing Interactions in Creative Collaboration**

Collaboration between solution authors and reviewers almost inevitably occurs during evaluation process, unless the authors do not remain engaged in the fixing process or the solution does not need any revision. In Mozilla and Python, most of the patches were reviewed: 4.8% Firefox bugs that were open and had patches had not received reviews from authorized reviewers; 3.8% Python issues that were open and had patches had not obtained any comment from the community.

At least one core developer is involved in the evaluation process. Collaboration mostly starts with the author briefly describing the design of the solution along with evaluation request. Peripheral members can suggest any modification, but are not authorized to determine evaluation results. When implementations will significantly change the software, or the core developer is not confident to make a decision, more core reviewers will participate in the evaluation. For instance, Mozilla has a super review policy, which applies to significant architectural refactoring, any change to any API or pseudo-API, and all changes that affect how code modules interact.
Moreover, if the solution may have impacts on User Interface (UI), localization or accessibility, correspondent experts will be requested for review. Python requires changes to a release candidate or a final release must be reviewed by two core developers unless the patch author is a core developer.

One common type of interactions during review is commenting through lines of code. No particular features of annotation are available in Bugzilla and Roundup. Reviewers usually quote a block of unsatisfactory code in their comment message and append their change suggestions. Authors do not always comply with reviews’ opinions; argumentation sometimes arises. A privileged Firefox reviewer doubted if a patch was correct, “if i read this correctly the behavior in the 2 cases is different ... Imo they should be consistent.” (Bug 501413; Mozilla). The patch author did not agree with the review, explaining that “No. ... Therefore, the code makes sure that the icon is clickable in both modes.”

Other than line-by-line reading through the code, reviewers who have no responsibility may just comment on what is interesting to them. This usually costs less time out of them. The following comment was created by an unassigned developer from Mozilla.

“I can’t help but question whether this really needs to load during startup at all. Seems like making any consumers of this code load the component would be a better direction since...” (Bug 506471; Mozilla).

Another type of interaction between reviewers and the author is asking-answering questions. Reviewers pick up the methods that do not make sense to them and solicit the explanation of the underlying rationale of the design. Mostly such questions convey disagreement with the methods used. Such interactions complement the knowledge that source code cannot fully convey. Not only is design knowledge shared between the author and reviewers, but also the author is motivated to reflect on his/her own design.

[Reviewer]: “Is there a real need to add a boolean argument to utcnow()? I think ... seems to be a more obvious way to produce TZ aware datetime.”
[Patch author]: “...using a boolean flag over an argument is much less error-prone for a developer from passing in the wrong timezone object...” (Issue 5094; Python).

**RQ2: Enablers of Creative Collaboration**

Participation of the appropriate reviews is a key enabler of creative collaborative solution evaluation. Both Mozilla and Python value patch contributions and put great effort to ensure the appropriate reviewers’ aware of review requests. Mozilla documents the owner and peers of each module, who are the only ones that can make decisions on patches. Bugzilla supports patch authors to set the flag as “review?” with a reviewer’s email address, which will alert the reviewer by email if his/her local notification setting is enabled. Even if patch authors appoint an incorrect reviewer, the appointed reviewer is often able to redirect the request to an appropriate one. Furthermore, the Bugzilla field whiteboard sometimes is used to highlight the demand of reviewers. Similarly, Python’s issue tracker provides a shortcut issues with patches to increase the visibility of issues that are pending for reviews. These issues are sorted out by keyword “needs review”, which can be selected by patch authors during patch submission.

Being encouraging and supportive when giving feedback can foster creativity (Amabile, 1996; Sternberg, 1999). In our cases, reviewers tend to let a solution pass review if only minor changes are needed. This lenient convention shortens the cycle of refining solutions, lowering the chances of bugs to be suspended.

“r=me with the last few changes talked about on irc” ⁴ (Bug 534398; Mozilla).

“The patch looks about right...So please remove the note on releasing md_state.” (Issue 4614; Python).

Informative critiques are generally considered conducive to creative performance (D. Johnson, et al., 2000; Tjosvold, 1985; West, 2002). In our data sets, we did not observe reviewers

⁴ Note that Mozilla uses “r=” to mark patches that pass review.
rejecting patches without giving any comments. They usually point out the defects of patches and explain the reasons.

Another factor that facilitates creative collaboration is high evaluation criteria. For instance, whether a patch can solve the problem presented in a bug is a lower level standard than robustness.

[Reviewer 1]: “[the patch] works just fine, no need for the scoping.”

[Reviewer 2]: “Why is this in the global scope rather than a DownloadMonitorPanel getter?”

[Reviewer 1]: “Good question! Let's do that instead.” (Bug 506099; Mozilla)

In another example in Python, a reviewer modified the way that a patch author used method _msize() in the patch although the patch could fix the issue. The patch author complimented, “Wow, clever use of _msize(), I hadn't thought of that. This is a much cleaner approach then the original and likely to be more robust” (Issue 5623; Python).

**RQ3: Obstacles of Creative Collaboration**

Making choices without fully considering all the available information will likely impede creative collaboration, although it did not frequently happen in our data set. For example, groupthink (Janis, 1982) and ignoring minority dissent may result in an ineffective fix that needs further repair. This might be avoided if the negative consequences of a solution can be well articulated and highlighted. For instance, other Python developers all agreed on a solution [developer 1] implemented. Although [developer 2] argued “...keep a reference to the required functions in the function arguments is ugly to me. The process should be destroyed before unloading the modules. … I think that [developer 1]’s patch is the wrong solution to the problem.” After the patch had been committed, a regression was detected (Issue 5099; Python).
Explicit review requests and specified responsibilities of core developers ensure that most of the solution implementations are assessed; however, such rules may depress the motivation of other members to participate in patch review. Fewer reviewers may decrease the probability of envisioning negative consequences of a solution. Inability to articulate negative consequences can incur the failure of a solution. In Mozilla, 11 bugs (0.15%) were reopened because last solution did not fix them. These did not include the bugs that had been reopened but were resolved by the time of our retrieval.

Additionally, the peer review processes that stopped in evaluation subprocess in our cases were mostly due to no new patches. In this scenario, existing patches failed to pass review and needed significant revision. Unless the bug was very critical, e.g., blocking product releases, or someone was interested, the fixing process could not move forward without the authors modifying their patches.

Discussion and Concluding Remarks

Peer review is sometimes regarded as "merely" routine maintenance or quality assurance. Our analysis shows how peer review in fact can be a creative collaborative work activity. Identifying and articulating a problem, recognizing an opportunity to elaborate or refine a design, improving an algorithm, creating a new feature or developing a new method are highly creative activities and accomplishments.

We describe four common components of peer review process that emerged from our analysis, problem identification, preparation, solution generation, and solution evaluation. They are essentially consistent with those creative process models in prior literature. For each component, we characterize its creative collaborative interactions, and codify the enablers and obstacles of the creative collaboration. The characterization and codification indicate
opportunities of supporting those interactions, enhancing the enablers and overcoming the obstacles.

Establishing Common Ground

A common construction of collective objectives, resources, and practices is indispensible for the creative collaboration in peer review processes to function, as indicated in our analysis results. Such common ground is especially important during the problem identification subprocess, when most of the bugs that fail to be fixed halt. Bug reporters have information of the specific problematic situation, which developers who are interested in fixing the bug do not. On the contrary, developers usually have the knowledge of what needs to be known to identify the cause of the problematic situation (not necessarily \textit{a priori}), which inexperienced bug reporter do not.

Simple and customized instruction of reporting bugs may facilitate establishing common ground of reported problems. Increasing the chance of reporters’ self-disclosing information can enhance the efficiency of information transfer (Convertino et al., 2008). This is particularly important in distributed collaboration, where people interacting with each other can withdraw at any time. Therefore, improving the quality of bug reports is critical. Suggesting specific steps of more tests for commonly reported problems may prevent reporters filing invalid bugs and reveal clues of problem causes. Bug tracking systems can also customize bug report formats for different types of bugs rather than a uniform format, or recommend similar bugs that are fixed to imply information in need.
Externalizing Social Networks

Assisting finding the appropriate experts via social networks may enhance the effectiveness of collaboration. One possible approach is to construct dynamic profiles that automatically reflect developers’ activities on software products as well as other people who have interacted with them over these products. An alternative could be visualizing the social networks of active community members based on their activities. Another solution is to nurture a larger group of active members than the core team can assist with connecting peripheral and core members. They do not have to acquire as much as the domain expertise of core developers.

Locating experts may be an efficient way to fix a bug, but may inhibit knowledge sharing and development within communities and impose too much workload on core members. Empowering active members with editing privileges other than core and loosely defining roles of core developers can encourage larger scale of participation, disseminating and reserving knowledge even if some experts leave the community.

Enhancing Awareness of Resolution Progress

The ability of reflecting on the shared goals and progress, planning accordingly and taking actions is crucial to collaborative creativity (West, 2002). During the solution generation subprocess in OSS peer review, keeping aware of resolution progress of a specific bug and all its interdependent bugs can facilitate discovering any demand of specific resources or help, adjusting developmental plans, and coordinating efforts across groups. Thus, such awareness largely determines the success of creative collaboration.

Visual aids and search facilities that highlight the resolution progress may enhance such awareness. Succinct identifiers of help/resource demands in bug summaries can probably attract
attention of volunteers, protecting them from being overwhelmed by hundreds of bug emails. Search for such demands can also be enabled by these identifiers, expediting filtering and prioritizing developers’ work. To facilitate fixing a group of interdependent bugs, tools may present timely updates of all related bugs for one bug report, such as notifications, and dependency visualizations.

**Articulating Design Rationale**

Design rationale stimulates software developers to reflect on their design by explaining and elaborating their ideas to other members, helps others understand and evaluate their design implementations, and may even facilitate developers explore or discover better design (J. Wang, Farooq, & Carroll, 2010). It is not explicitly documented in OSS development. However, collaboration in peer review processes creates opportunities to externalize such implicit knowledge. In our cases, it is articulated mainly in three ways, including self-introduction, responsive-explanation, and critiquing-comment.

A well-integrated code review tool into bug tracking systems may facilitate articulating design rationale. The code being discussed is often not at the level of an entire patch or a change set, but rather part of them, such as a method in a patch. Currently developers in our cases have to quote a block of code by copying it to the discussion space in bug tracking systems. This may fragment discussions, causing design rationale hard to be tracked especially when multiple participants interact. Providing a code-based discussion space for bug solution evaluation may facilitate the interactions between code authors and reviewers. Discussions upon a single method within the code can be clustered, which helps participants make sense of the conversations and aggregate design rationale.
Chapter 5

Diversity in Open Source Software Peer Review Processes: A Case Study on Mozilla

With the overall understanding established from the studies described in previous chapters, this chapter examines a single case in more depth. As Chapter 3 shows, Linus’s law appeared especially limited in Mozilla: the many “eyeballs” generated a large portion of bugs that had low quality. A likely factor resulting in this inconsistency between ideology and reality is member diversity. This chapter reports my efforts to understand the impacts of diversity on the open source software peer review process. The analyses show that diversity increases workload and communication, as well as frustration and conflicts. However, it facilitates problem characterization, design review, and boundary spanning. We discuss implications for work performance and community engagement, and suggest several ways to leverage diversity in the open source software peer review process.

Related Work

Open Source Software Peer Review

In general, the OSS peer review process begins with one submitting a bug report to the bug tracking system—an application that helps developers keep track of reported defects or deficiencies of source code, design, and documents. Others examine the defect causes and request additional information to determine whether the bug should be fixed. Once a solution is reached, they then commit a change set (mostly a patch) to the current software product. Version control
systems manage and synchronize such product changes, while communication tools such as mailing lists and Internet Relay Chat (IRC) enable developers to discuss bugs.

Most studies related to the OSS peer review process were conducted from the software engineering perspective, deliberately modeling the information needs in bug report quality (N Bettenburg, et al., 2008; Breu, et al., 2010), inaccurate bug assignment (Jeong, Kim, & Zimmermann, 2009), efficiency and effectiveness of patch review (P. Rigby, et al., 2008), and distribution of contributions in bug reporting and fixing (Mockus, et al., 2002). Rigby et al. characterized OSS review processes into review-then-commit (RTC) and commit-then-review (CTR) and described the OSS patch review in terms of its stakeholders’ behaviors (P. Rigby, et al., 2008; P.C. Rigby & Storey, 2011). They articulated stakeholders’ involvement and their approaches for managing broadcast-based patch review. They also found that stakeholders interacted differently when discussing technical issues and when discussing the project scope.

Much effort has been devoted to explaining the coordination mechanisms (K. Crowston, 2008; Sandusky & Gasser, 2005; Yamauchi, et al., 2000), negotiation (Sandusky & Gasser, 2005), leadership and governance (Fielding, 1999; Moon & Sproull, 2000), and the role of bug tracking systems (Bertram, et al., 2010). Recent work by Ko et al. (Ko & Chilana, 2010) analyzed the reports of Mozilla contributors who reported problems but were never assigned problems to fix, indicating that the diverse competences of crowd can lower the quality of the OSS bug reporting. Wang et al.’s analysis (J. Wang & Carroll, 2011) also showed large volume of bug reports failed to identify real bugs, increasing the cost of filtering them out. Despite these efforts, there is still limited understanding of how collaboration occurs in the OSS peer review practices, particularly the intertwined social processes (K. Crowston, et al., 2012).
Diversity in Collocated and Distributed Groups

Diversity is commonly defined as the differences of any attributes among individuals. As a complex construct, it has been studied in multiple disciplines, such as organizational behavior, sociology, and psychology. A complete review of this large body of literature is beyond the scope of this paper. However, regardless of its typology, diversity can be of the readily visible attributes (e.g., gender, ethnicity, and age), of informational attributes (e.g., education, work experience, and knowledge), and of attitudes and values (i.e., what members think the group's real task and goal should be) (Jehn, Northcraft, & Neale, 1999; Van Knippenberg & Schippers, 2007; Williams & O'Reilly, 1998).

Research on the effects of diversity has been inconclusive. Studies found diversity can affect work processes, performance, and member identification in both positive and negative ways, and effects of the same diversity dimension may vary greatly across contexts (Harrison & Klein, 2007; Joshi & Roh, 2009). For instance, diverse perspectives tend to benefit work performance in short-term tasks, but these positive effects become much less significant in longer-term teams, and conflicts start to arise (Joshi & Roh, 2009). In general, a broad range of expertise and knowledge can enhance problem solving, decision-making, and even creativity and innovation, while differences of perspectives and values can result in dysfunctional group processes, conflicts, and poor performance (Milliken, et al., 2003; Van Knippenberg & Schippers, 2007; Williams & O'Reilly, 1998). However, these studies were largely conducted in collocated groups in organizations or laboratory experiments. A few other researchers looked into diversity in virtual teams. Shachaf (Shachaf, 2008) explored the heterogeneity of national cultures of members from ad hoc global virtual teams at a corporation. The interviews showed such cultural diversity has positive influences on decision-making and negative influences on communication. Damian et al. (Damian & Zowghi, 2003) also focused on cultural diversity and found that it
increased team members’ difficulty in achieving common understanding of software requirements. Several chapters in (Hinds & Kiesler, 2002) discussed conflicts caused by differences of organizational cultures and informational diversity. However, such work still analyzed diversity in the settings in which group or organizational boundaries were clearly defined. There have been very few studies on the effects of diversity in volunteer-based large-scale online communities, such as OSS projects and Wikipedia, and therefore, they warrant additional examination.

Another theme of relevant research on virtual teams did not specifically analyze diversity but members’ differences that were caused by distance, such as differences of information about remote contexts and differences of time zones. These differences led to conflicts (Cramton, 2001), communication and coordination (Cataldo, Wagstrom, Herbsleb, & Carley, 2006; Herbsleb & Mockus, 2003; Olson & Olson, 2000). Our goal is to investigate diversity beyond these language and location differences, in order to understand how other aspects of diversity may influence collaboration.

**Diversity in Online Peer Production**

Little research on diversity has examined its effects on online peer production, such as OSS development and Wikipedia. To our knowledge, only one paper on OSS has directly addressed diversity (Daniel, Agarwal, & Stewart, 2013). Its authors quantitatively operationalized diversity with respect to roles, tenure, languages and activity levels to determine the relationship between diversity and user participation when analyzing 337 SourceForge OSS projects. They showed that project roles and participant tenure positively affect user participation, whereas variations in participant activity negatively influence user participation. They did not find any significant impact of language diversity on user participation. However, such a quantitative
approach is inadequate for characterizing diversity and articulating its consequences. For instance, tenure measured by the amount of time since one registered on SourceForge can hardly represent one’s experience with the platform, because experience clearly depends on the amount of interactions one has with the platform. Approaching the data qualitatively can help complement existing research on diversity and OSS.

Other work on OSS only characterized differences between members in OSS communities. Early work analyzed the motivations of participants in OSS development and identified incentives varied among individuals, including utilitarian reasons (own needs for software functions), joy and challenge, and reputation (J Feller, Fitzgerald, Hissam, & Lakhani, 2005; Roberts, et al., 2006). The inequity of contributions is commonly perceived in OSS development: a group larger than the core by about an order of magnitude contributed to defect fixing, and a group larger by another order of magnitude contributed to defect reporting (Mockus, et al., 2002). Diversity of expertise was also suggested by investigations on usability discussions in OSS communities (Bach, DeLine, & Carroll, 2009; Twidale & Nichols, 2005). They argued that end-users and usability experts could provide special expertise and knowledge the other developers do not have. Sandusky et al.’s work (Sandusky & Gasser, 2005) on negotiation activities in OSS bug finding and fixing processes uncovered the different values and perspectives of individuals involved. In addition, Ko et al. indicated the disparity of capabilities in OSS bug reporting (Ko & Chilana, 2010). However, their analysis only considered information created by bug reporters, dismissing other contributors who added other valuable information by commenting on the reported bugs.

Other research on Wikipedia used software visualizations to analyze patterns of how online volunteer contributors negotiate conflict and coordination (Kittur, Suh, Pendleton, & Chi, 2007; Viégas, Wattenberg, & Dave, 2004). They found that community surveillance is important for ameliorating hostile behaviors. Additionally, Chen et al. found that increased diversity in
experience with Wikipedia and interests in topics increases group productivity and decreases member withdrawal, but after a point increased diversity of experience will increase member withdrawal (Chen, Ren, & Riedl, 2010).

Methods

Case Selection and Description

To address the richness and complexity of the diversity of crowds in open source, we conducted a case study of Mozilla, a large and well-recognized OSS community. We focused our analysis on its core project, Firefox, expecting its end-user orientation would provide substantial instances of diversity, such as different expertise and experience.

The Mozilla community consists of both employees from Mozilla Foundation and Mozilla Corporation and volunteers. Core developers, mostly employees, are the ones who contribute significantly to the project. They are divided by a variety of different roles, including module owners, module peers, super reviewers, security group members, quality assurance members, and support team members. Each individual has to maintain and facilitate the evolvement of his or her specific areas. For example, Firefox had a sub-module Session Restore, which was supported by one sub-module owner and three peers. Some other developers also actively contribute to the project, but they are not affiliated to any of the organizations of Mozilla. Peripheral members rarely participate in developing the software. They are comprised of end-users, web developers who design extensions or other applications on top of Mozilla’s technology, network administrators, third-party developers, and developers from competitor products.
The peer review process in Mozilla follows a strict review-then-commit fashion. Any changes (e.g., patches) to a module or sub-module had to be reviewed by its owner or peers. Similar to other OSS projects, Mozilla’s peer review involves a wide range of work and communication tools, including version control systems (i.e., Mercurial), bug tracking systems (i.e., Bugzilla), IRC, mailing lists and wikis. Bug tracking systems are the primary space for issue reporting, analyzing, and fixing in Mozilla, unlike some other projects that rely primarily on mailing lists to do so.

Data Sampling and Analysis

We retrieved bug reports created between two stable releases of Firefox from Bugzilla. This sampling strategy was intended to capture the possible behavioral changes near releases (Francalanci & Merlo, 2008). The retrieval was performed on July 28, 2010. It includes 7,322 reports filed between the releases of Firefox 3.5 final (June 30, 2009) and 3.6 final (Jan 21, 2010). Additionally, we examined emails, blogs, wikis, documents and bug reports that were related to or mentioned in the bug reports we retrieved.

We used email addresses to identify contributors in Mozilla’s peer review process. Core developers were defined as the ones listed on Mozilla’s websites as module and sub-module owners and peers, Firefox development team members, super reviewers, security group members and members with additional security access, quality assurance team lead, and platform engineers during the time between Firefox 3.5 and Firefox 3.6 releases. 176 developers were identified in this category. Crowston and Scozzi suggested that it may be more useful to define core developers based on the amount of their actual contributions rather than their job titles (K. Crowston & Scozzi, 2002), but our approach may better differentiate the incentives and responsibilities of core developers from the normal volunteers.
Overall, we sampled 10% of the bug reports from our retrieved data set, which included 732 bug reports. To accommodate the possible variations of work processes and social interactions across bug reports, we performed stratified sampling with respect to bug status and resolutions. In Bugzilla, open bug reports have 4 types of status, *unconfirmed*, *new*, *assigned*, and *reopened*. Closed bug reports have 6 types of resolutions, *fixed*, *duplicate*, *invalid*, *incomplete*, *worksforme*, and *wontfix*. For each of the 10 types of bug status and resolutions, we sampled 10% of them: we first intentionally selected the bug reports with number of comments at the 98th percentile; then we randomly sampled the rest. Each bug report has its unique ID. In the discussion space of each bug report, the first comment is labeled “description”, which is generated by the bug reporter to describe the issue. All the other comments are indexed in order starting from 1. When quoting discourses in the next section, we use the format (bug ID; comment number; contributor type).

Our qualitative analysis was comprised of three phases. First, the first and the second authors randomly selected 50 bug reports and read them separately. They discussed their observations and established shared understanding of the peer review process in Mozilla. Then during the second phase, the first author coded the 732 bug reports, iteratively generated 628 codes occurring a total of 1623 times, and discussed and consolidated them with the other two authors during their weekly meetings. Finally, 12 groups of codes that occurred the most frequently and deemed the most relevant to diversity were selected. Overall, the 12 groups of codes occurred 419 times accounting for 25.8% of total occurrences. Then the authors integrated them into 5 themes, which are presented in the results section.

We identified the existence of diversity and its various dimensions through interpreting the discourses between participants during the peer review process. For example, commenters would be identified as inexperienced if they showed they did not know the commonly used terms, techniques, and procedures in Mozilla, such as creating a new user profile. Such qualitative
evidence of diversity may be less straightforward than the number of years worked in the industry (which was not available in our data set); however, it demonstrates participants’ experience with respect to their involvement with Mozilla’s peer review process rather than their programming abilities in general.

**Results**

Overall, the 7,322 bug reports in our data collection were created by 5,418 unique reporters. The number of bug reports created per reporter ranged from 1 to 82, with the median of 1 report per reporter. 1,286 additional contributors participated by commenting in the discussion space of the bug reports. The number of bug reports contributed per contributor varied between 1 and 990, with the median of 1 report per contributor. While the number of core developers was much smaller than the number of the bug reporters, the distribution of roles was relatively balanced within each bug report: 2 Mozilla members participated in discussing one bug \((\text{median}=2; \text{skewness}=6.765)\) and one of them was a core developer.

Core developers spent the majority of their efforts on patching bugs, whereas the other volunteers contributed more to bug reporting. The 176 core developers reported 832 bugs, which were 11.36% of all the bug reports. 579 bugs had specific assignees (mostly patch authors). 81.67% of these bugs \((n=474)\) were assigned to core developers.

From our qualitative analysis, we found that various types of issues were reported, including functional errors, user interface (UI) issues, performance issues, security and privacy concerns, and so forth. There were also several bugs created for administrative purposes by core developers, such as specifying third-party contact information.

We observed several dimensions of member diversity in the peer review process of Firefox, such as diversity in capabilities, experience, values, motivations, expertise, knowledge
and resources. Five consequences of diversity, both negative and positive, on work performance and community engagement emerged from our data, including increased workload and communication, frustration and conflicts, problem characterization, design review, and boundary spanning. We describe each consequence in the following sections.

**Increased Workload and Communication**

Much workload of the OSS peer review participants was spent on sifting through the “non-real” bugs (n=3418; 46.68%). These bugs constitute 87.28% of the closed bug reports, the rest of which are fixed bugs. They include the redundant reports (duplicate), issues caused by reasons other than Firefox (invalid), problems that cannot be reproduced by anyone other than the reporter or identified in the code (worksforme), vaguely described bugs (incomplete), and requests that will not be fixed (wontfix). Table 5-1 and 5-2 summarize the distribution of the status of open bug reports and the resolutions of closed bug reports respectively.

Table 5-1: Distribution of the status of open bug reports.

<table>
<thead>
<tr>
<th>Status</th>
<th>Number of Bugs</th>
<th>% of Open Bugs</th>
<th>% of All Bugs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unconfirmed</td>
<td>2680</td>
<td>78.68%</td>
<td>36.60%</td>
</tr>
<tr>
<td>New</td>
<td>676</td>
<td>19.85%</td>
<td>9.23%</td>
</tr>
<tr>
<td>Assigned</td>
<td>39</td>
<td>1.15%</td>
<td>0.53%</td>
</tr>
<tr>
<td>Reopened</td>
<td>11</td>
<td>0.32%</td>
<td>0.15%</td>
</tr>
<tr>
<td>Total</td>
<td>3406</td>
<td>100%</td>
<td>46.51%</td>
</tr>
</tbody>
</table>

Table 5-2: Distribution of the resolutions of closed bug reports.

<table>
<thead>
<tr>
<th>Resolution</th>
<th>Number of Bugs</th>
<th>% of Closed Bugs</th>
<th>% of All Bugs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed</td>
<td>498</td>
<td>12.72%</td>
<td>6.80%</td>
</tr>
<tr>
<td>Duplicate</td>
<td>1346</td>
<td>34.37%</td>
<td>18.38%</td>
</tr>
<tr>
<td>Invalid</td>
<td>937</td>
<td>23.93%</td>
<td>12.80%</td>
</tr>
<tr>
<td>Worksforme</td>
<td>514</td>
<td>13.13%</td>
<td>7.02%</td>
</tr>
<tr>
<td>Incomplete</td>
<td>489</td>
<td>12.49%</td>
<td>6.68%</td>
</tr>
<tr>
<td>Wontfix</td>
<td>132</td>
<td>3.37%</td>
<td>1.80%</td>
</tr>
<tr>
<td>Total</td>
<td>3916</td>
<td>100%</td>
<td>53.49%</td>
</tr>
</tbody>
</table>
Diversity of participants’ capabilities and preferences of characterizing a bug increases the chances of submitting duplicated bug reports. Sometimes the same problem can cause multiple symptoms, as Bettenburg et al. also suggested in their analysis (N. Bettenburg, Premraj, Zimmermann, & Kim, 2008). Participants differ at their capabilities of identifying the cause and drawing connections between their own problems and existing bug reports. For example, a reporter could not figure out that his browser freeze was due to large numbers of checkboxes on the web pages. In contrast, another commenter pointed that out and was able to identify it as a duplicate of the bug “Session restore hangs/not responding with high CPU on large form with many checkboxes” (Bug 477564).

“Firefox 3.5 is locking on MusicBrainz artist pages that have a lot of releases. ... The lock-ups also occur when using the parameter "safe-mode". (Bug 506553; description; reporter/end-user)

“There is a login at the page and that might change the page itself. Does this page contain many checkboxes if you login?” (Bug 506553; comment 1; commenter/active volunteer)

When reporting bugs, core developers prefer to use very technical languages to describe bugs at the code level, in order to ease the communications with other developers. In contrast, other less technical end users tend to draft bug reports based on the observed symptoms. Such differences create barriers for non-technical people to find out whether their issues have already been reported and acted upon. For instance, the master report of a bug about displaying file icons was summarized as “nsIconChannel::GetHIconFromFile does not support Unicode filenames on Windows” (Bug 415761). In contrast, one of its duplicates described the issue in a language that is more familiar to the end-users.

“I've just noticed that in the Downloads menu (which shows when you download something) all icons are [shown] as folders. In previous version of Firefox if you download something in the Downloads menu it was shown (icons)- I mean Pictures had Picture icon, Files had File icon and so on. But now all items (pictures, files...) have the same icons - they are shown as folders” (Bug 504350; description; reporter/end-user).
Diversity of experience is another factor that the creation of the many “non-real” bugs can be attributed to. The experience of participants varies at reporting bugs, debugging, and understanding organizational norms and practices. To proceed with the peer review process, experienced participants have to mentor or explain to the inexperienced collaborators through online discourses. One recurring communication pattern in invalid bug reports demonstrate such disparity—experienced contributors teach novice reporters to run test cases in the safe mode and under a new browser profile, attempting to isolate problems caused by extensions or other third-party applications.


“[U]sing those modes I was able to determine it was the sk[yp]e plugin causing the issue (which was installed a few days ago)” (Bug 507708; comment 2; reporter/end-user).

Another resource that experienced participants frequently suggest to inexperienced participants was how to collect additional information using a stacktrace memory dump, which is critical for crash analysis.

“[C]an you please be more specific about this CRASH ID (STACKTRACE) for a start.” (Bug 509950; comment 10; reporter/end-user)

“We need a crash ID (if the crash reporter comes up) or a stack trace (if the crash reporter doesn’t start). ... The link contains step by step instructions to either get a crash ID or a stacktrace. Start here https://developer.mozilla.org/en/How_to_get_a_stacktrace_for_a_bug_report#How_to_get_a_crash_ID_with_the_Mozilla_Crash_Report”. (Bug 509950; comment 11; commenter/end-user)

Diversity of experience also entails the time to correct faulty operations and to explain organizational norms and practices, clarifying any misunderstanding or confusion. People who are not regularly involved in Mozilla’s peer review process probably know very little about, for example, what the meta-data of a bug report represents, but are only capable of reading with their own interpretations. In the following case, a reporter who had only commented on 1 bug in our
data set believed the resolution of his bug should be flagged *fixed* because Firefox behaved normally after he upgraded to version 3.5.2. However, an active volunteer, who had contributed to discussions of 990 bugs in our data collection, modified this flag to *worksforme*, indicating the reason that FIXED was not an appropriate resolution flag.

“Resolution FIXED is reserved for bugs that had code applied to resolve the problem. So at least a bug number is needed”. (Bug 502328; comment 19; commenter/active volunteer)

The increased communication arises not only from sharing norms, but also from redirecting requests to appropriate experts. The awareness of who is qualified or available to address the request can only be gained by extensive and long-term involvement in the project. Without such awareness, participants usually specify developers that are not suitable for the bugs involved, thus creating a long forward chain before the bugs land in the right developers’ hands.

“I can’t review code in js src. I’m forwarding to [developer 1], feel free to forward again in case I did a bad choice”. (Bug 526422; comment 2; commenter/core developer)

In summary, diversity of technical capabilities, terminology preferences, experience with the OSS peer review process, and familiarity of developer expertise increases the work of screening useless information from numerous bug reports as well as the communication of repeating how to improve bug reporting to novice contributors. The increased communication also helps orienting the newcomers, reinforcing shared norms, and connecting resources.

**Frustration and Conflicts**

The “many eyeballs” do not accomplish work magically. They must go through extensive negotiations, and convergence is not always easy. Consequently, frustration and conflicts emerge from the interactions of diverse values, perspectives, and motivations. Firefox as an end-user
oriented application may amplify such effects even more so than other developer-oriented applications (e.g., Apache).

A common controversy is what actually constitutes a bug. The diversity of values and perspectives becomes salient in such discussions: the possible consequences of not fixing a bug per request are not simply software dysfunctions; rather, they are manifestations of personal assessments and judgments. Developers tend to consider factors such as how many users a change could affect and how much the cost would be for code maintenance. In contrast, end-users prioritize the usefulness of a feature to the individual user’s task at hand and the consistency of interaction paradigm and interface layout. The debate on whether to remove “Properties” context menu illustrates such diversity.

“When you have 50 images in the page, or even much less, you have *no* idea which URL points to the image you want. Till now, we had an easy way to get meta-data about images and links, and you remove this feature because you want to save 48 Kb of code”. (Bug 515368; comment 9; commenter/end-user)

“If you'll permit me to restate your point, I think you're saying that you estimate the maintenance cost of 1300 lines of unowned code and the UI cost of an extra context menu entry as being collectively lower than the value of providing more direct built-in access to image metadata on pages with many images than page info provides. Is that fair? ...whether we think the feature serves a large portion of our users, what the cost is of keeping it, and what downstream impacts removal might have”. (Bug 515368; comment 12; commenter/core developer)

Diversity of views on a bug’s significance also results in unresolved conflicts. Unsubstantiated arguments cannot convince the opposing parties. Ultimately, the core developers have the final say in the decision-making process and the end-users are frustrated or irritated.

“When, let's not make "there are X many comments" or "X many votes" extrapolations. It's not good science, and that besides, 61/300,000,000 = not much of a convincing argument, even if you apply the standard "one comment represents a thousand users" math. ... You need to understand that most users do not find this disturbing, and indeed don't really run into the problem. ... I need to build a browser that's best for most people, not specifically the 61 people who commented on this bug”. (Bug 456405; comment 71; commenter/core developer)

“People who comment on bugzilla are the users who made Firefox popular, you should value them and their opinion very much. ... How do you know what is
good for the nameless mass of people ..., they don't even know that bugzilla exists. Did you conduct a survey or asked the users some other way?" (Bug 456405; comment 73; commenter/end-user)

Different beliefs of community values and interpretations of community cultures can also stimulate complaints and criticism on work processes, such as how decisions are made. Open source, originally emerged as an opponent of proprietary software, provided users an expectation of openness and an heightened ability to participate and contribute to the development process. Therefore, users tend to conceive of OSS as democracy, whereas the ones who have been involved in OSS development for a fairly long time, such as core developers, consider it meritocracy. Accordingly, users can easily feel unsatisfied when their requests are not fulfilled.

“Sometimes the democratic system works. Sometimes a group with a very closed perspective hijack it. Congrats on making FF a laughing stock of a browser”. (Bug 513147; comment 70; commenter/end-user)

“Mozilla is not a democracy. Open source does not mean "the developers must do my bidding". And being able to comment in Bugzilla is a privilege, not a right”. (Bug 513147; comment 71; commenter/core developer)

“Mozilla is a meritocracy, and as such, the module owner/peers have final say over decisions concerning the codebase”. (Bug 513147; comment 79; commenter/core developer)

“It's these types of options which makes Firefox better than IE/MSFT. In IE, you don't get a choice, and you will never have one. [Firefox] is about choice and user control (on many levels), and what's happened on this issue has gone against those principles”. (Bug 508474; comment 8; commenter/end-user)

Aside from the variance of their philosophies, participants also conceive differently the role of the bug tracking system. Lack of shared understanding may cause unproductive collaboration and discourage further involvement.

“... I never got any help howto really diagnose the problem, so I simply don't care anymore. The data is gone, as is the problem (hopefully)”. (Bug 515538; comment 10; reporter/end-user)

“Notice this is bugzilla, not supportzilla, to get support help you should go to support.mozilla.org. Here we only try to debug and fix issues but clearly we needs steps or files to reproduce the issue. And if we ask for informations you
should not consider them irrelevant, we need them to try reproduce the issue”.
(Bug 515538; comment 11; commenter/core developer)

Additionally, perceived differences of motivations to participate sometimes can impede collaboration. For example, some reporters contribute because they want Mozilla developers to alter a Firefox module to fit their product websites or third-party browser extensions. Developers would decline such requests but their responses are typically not well received.

“I'm not sure why you're coming here asking for help ... without being willing to accept our analysis”. (Bug 528153; comment 13; commenter/core developer)

Yet, developers can sometimes have different agendas and interests, such as minimizing workload. Such discrepancy of motivations and intents creates tension within the community. Accordingly, these reporters find it difficult to trust the developers and appreciate their effort.

“I am also fully aware that in situations where people are doing technical support for a lot of people, their first objective is to keep their inbox clear by finding someone else to blame”. (Bug 528153; comment 14; reporter/end-user)

In short, diversity of individual values, perspectives, interests, expectations, and motivations entails negotiation of bug legitimacy, priority, community values, and norms. Failures to establish shared understanding and mutual appreciation lower the community morale, generate emotional conflicts, and even cause member withdrawal.

**Problem Characterization**

One of the advantages of OSS peer review lies in the diverse resources of testing environment. Individual access to operating systems, software versions and other settings is usually limited compared to what is accessible to the crowd. Moreover, users vary in the ways they use the software, habitually or accidentally, encountering a problem through different paths. Such diversity enables a bug report to be collectively constructed; it helps developers approach the underlying causes by enhancing the accuracy of problem description. The role of bug
reporters extends beyond the ones submitting reports to bug tracking systems, encompassing whoever complements the information about a problem or an issue.

“Firefox 3.5 does not ask for save all tabs even when configured to do so and even when there is only one Firefox window opened, ... My Firefox is 3.5 Brazilian Portuguese version. My Windows is Vista 64 Ultimate”. (Bug 502532; description; reporter/end-user)

“I have a similar problem. Mine does ask, and I tell it to save my tabs. I open it up the next time, and I get my home page, not my saved tabs. I tell Firefox to clear the following data when I close it: Browsing History, Download History, ... I'm running Firefox 3.5 English and Windows Vista x64 Ultimate”. (Bug 502532; comment 1; commenter/end-user)

“This issue also occurs for me on Mac OS 10.5. I tried to uninstall and re-install after wiping the profile folder but that doesn't help”. (Bug 502532; comment 2; commenter/end-user)

Other than describing personal experiences of the problematic situation, participants with different levels of analytical skills (e.g., debugging) discover relevant information by performing additional investigation. This often refines the problem characterization, allowing someone else with better expertise to eventually identify the cause of the defect.

“[I] face the same issue. ... i suspected it's a network connection, but other tabs would be loading without issue. chrome doesn't face the issue. i don't face this on ubuntu. i started facing this only in 3.5. it's most readily seen while stumbling, and it's not a traffic issue (i rarely have more than 3 or 4 tabs open)”. (Bug 502963; comment 2; commenter/end-user)

Another type of diversity—commitment and time resources—also contributes to problem characterization. Participants in OSS projects are willing or able to devote their time and effort to the peer review process at different degrees. When a contributor is not available to address certain requests or inquiries, other contributors will be able to do so in his/her place. In the episode below, a reporter did not provide a test case in his post more than one month ago. An active contributor had asked for a test case to analyze the problem but the reporter never responded. After a few weeks, another participant created and uploaded a workable test case.
2009-07-18 “A test case is far more useful, then a crash report ... Can you attach input to this bug, that crashes FF? Of course, a full test case would be better ...” (Bug 506844; comment 2; commenter/active volunteer)

2009-08-28 “Since https://bugzilla.mozilla.org has banned our country IPs I could not check my reported bug regularly”. (Bug 506844; comment 12; reporter/end-user)

2009-09-16 “Created attachment 401081

This demonstrates the slowdown on emptying a div with a bunch of contents via innerHTML ... Using the testcase: 1) fill the div with either 250, 500 or 1000 lines. 2) Empty it with "Empty (innerHTML)" or by filling it with any number of lines.” (Bug 506844; comment 14; commenter/end-user)

Overall, despite the noise that “many eyeballs” introduce to the bug finding process, OSS peer review still benefits from the diversity of the crowd for effective characterizations of software defects and deficiencies. The OSS peer review process allows individuals to complement the bug reports with their personal experiences and findings, facilitating the process of problem characterization.

Design Review

The large number of end-users in the Mozilla community not only generates the disparity of capabilities and knowledge with respect to software development work, but also increases the variety of expertise and perspectives. The rest of the community also varies in terms of the areas they are specialized or issues they know well. Such diversity allows early design ideas and actual design implementations of bug solutions to be reviewed by different experts, creating opportunities of design improvement.

Users are the best for providing first-hand knowledge of how the software will be used. Even though they may not always be able to reflect or explain their experiences or behaviors, they can still articulate the use scenarios that are overlooked or misconceived by the developers.
In the bug report that proposed to “improve Mac install experience: detect that we're running from disk image or Download folder”, a participant reminded the designers a special use case of the Firefox installer.

“Another thing to consider, if only for being aware of it, is that users with encrypted home folders (FileVault) will be copying and not moving Firefox when they drag it to the Applications folder (since the home folder is an encrypted volume different from where the Apps folder reside, and drag&dropping to another volume defaults to copying instead of moving).” (Bug 516362; comment 12; commenter/end-user)

Given the complexity of OSS like Firefox, the implications of a design implementation could be very complicated and require knowledge and expertise from different fields, which is beyond an individual’s ability to comprehend. Elaboration and dissent among parties with diverse perspectives help articulate rationale of software design, improving its quality. Such disagreements, or even critiques, are often conveyed in a concrete and constructive way. For instance, Firefox has teams specialized in user experience, platform, security, localization (L10n), and so forth. Members from different teams tend to evaluate design from their own lenses. The following comment on a patch of moving browserconfig.properties to a jar file was posted by a member of the localization team, who works on “translating an application's menus, dialogue boxes, labels and other settings (font, character encodings, dictionaries, default search engine etc) for a particular locale.”

“browserconfig.properties contains locale dependent information, and should not be in browser.jar. ... We're also contemplating changing the url [of browserconfig.properties] completely for some of our smaller locale, in particular for Kurdish google doesn't seem to be a good choice as it prompts users of our latin-based kurdish with arabic script kurdish”. (Bug 524201; comment 13&15; commenter/core developer)

“Oh, right. Just means we need to package it in the locale jar, then”. (Bug 524201; comment 16; commenter/core developer)

Patch review policies in Mozilla only authorize module/sub-module authors to evaluate changes to their own code in order to ensure the proper expertise of the reviewers. However, this
also means that these designated patch reviews typically only involve 1 developer per patch, which is sometimes not enough to capture all the possible defects. It is not uncommon for contributors outside of the designated review groups to capture faulty designs. The episode below shows that the requested reviewer missed one defect in a patch, which was caught by an outside developer.

“The patch is also wrong because bookmarksBarContent may be absent from the document”. (Bug 504858; comment 30; commenter/core developer)

“Indeed! I should have caught that…” (Bug 504858; comment 31; commenter/core developer)

To sum up, diversity of expertise, knowledge, and perspectives, both in general and specific to a bug, enriches the variety of use scenarios being articulated at the early stage of design. Furthermore, it facilitates the articulation of design rationale through dissent and critique, assuring the quality of implementations.

**Boundary Spanning**

Participants in OSS communities often differ at their access to various information and resources without constraints of group and organizational boundaries. Similar to boundary objects (Star, 1989), participants share or leverage knowledge and resources from other work groups, sub-communities, and organizations that would not otherwise be accessible or obtainable by the developers involved in resolving a bug.

Contributors often cite reports and discussions outside bug tracking systems, such as user support forums and informal community websites, to confirm a bug’s validity. Although this is often employed as a strategy to attract developers’ attention and entice them to act on the problem, it sometimes provides supplemental information that facilitates problem analysis.
“I’m gonna confirm this then because I seen a report on mozillazine with a trunk build that this happens.” (Bug 501904; comment 4; commenter/active volunteer)

“... adding additional versions to the bug summary based on the interesting-modules-with-versions report in http://people.mozilla.com/crash_analysis/ ; I'd note that the versions reports show that the older versions seem to be ok, but the newer versions seem to be causing crashes.” (Bug 527540; comment 10; commenter/core developer)

Knowledge regarding third-party applications is also critical to bug analysis because of the high interdependencies among software applications. For instance, Firefox is dependent on a variety of existing third-party software infrastructure, including database, operating systems, and multimedia platforms. At the same time, its design also affects extensions and websites. Developers of these third-party applications are actively involved in Firefox’s peer review process, communicating and collaborating with other participants on diagnosing and resolving issues affecting both sides. The excerpt below illustrates a situation in which the reporter encountered the bug, “Browser segfaults attempting to delete a database entry (in do_lookup_x () from /lib/ld-linux.so.2) on Twitter and others.” A developer from Redhat, which was the relevant third-party in this problem, shared the link to a similar report at Redhat that offered his analysis and possible solutions.

“We received a similar crash report at https://bugzilla.redhat.com/show_bug.cgi?id=560662” (Bug 501992; comment 4; commenter/third-party contact)

[Reporter]: “[Y]our problem is caused by canberra which is presumably delivered by your vendor (fedora), please use [comment 4]’s bug link to deal with this problem.” (Bug 527029; comment 6; commenter/core developer)

Other than cooperating with third-party contacts, participants in Firefox peer review collaboration also observe, compare, and learn from other browsers, such as Microsoft Internet Explorer, Google Chrome, and Opera. Participants with experiences using multiple browsers often report their personal experiences with other browsers back to the peer review community.

“I believe that the "Paste and Go" option in both Opera and Chrome is something that would greatly benefit Firefox. For those that are not familiar
with what I am talking about, on both of the aforementioned browsers, if you right click on the address bar one of the options that appears in the menu alongside "Cut", "Copy", and "Paste" is the "Paste and Go" option. This is a small time saver that belongs in Firefox”. (Bug 501558; description; reporter/end-user)

More interestingly, developers of competitor browsers occasionally join the conversation and share their design knowledge. The following episode describes a case in which a Chrome developer shared the implementation details of the Chrome browser when he observed that Firefox developers were trying to match Chrome’s perceived scrolling speed. This also indicates that developers of major browsers are monitoring features of competing products closely, and sometimes they even collaborate to make their products better.

“Just so you guys know, if the goal was "match Chrome behavior", doubling the system scroll lines value isn't Chrome's behavior at all. (I wrote Chrome's behavior.) ... In short, we use 100 px per "three lines", and the system ticks-lines mapping (which defaults to three lines per tick, so by default we do 100 px per tick)”. (Bug 513817; comment 89; commenter/competitor developer)

In short, the fluidity of OSS organizational boundaries and the flexibility of OSS community structure allow information flow and resource configuration spanning across groups and organizations. Participants with multiple identities contribute their knowledge and information that would otherwise be inaccessible to the peer review process. Such diversity benefits problem diagnosis and design idea generation.

Discussion

Our investigation suggests that diversity of experience, capabilities, experience, values, motivations, expertise, knowledge and resources generates both benefits and challenges, affecting both work processes and social processes involved in the OSS peer review. For work process activities such as bug submission, problem identification, solution generation, and solution evaluation that are common to the OSS peer review process (J. Wang & Carroll, 2011), different
expertise, knowledge and resources enable the crowd to complement bug reports with their own experiences and analysis. The crowd enriches the articulation of use scenarios and design rationale through critique and dissent, and exchange information with external groups and organizations. Other than enhancing problem characterization, design review, and boundary spanning, diversity also creates challenges to the work process of OSS peer review. Specifically, the disparity of capabilities and experience leads to a large number of redundant, invalid and ambiguous bug reports submitted, increasing the workload of screening bug reports and the need of repeatedly communicating to novices about how to report bugs. For the social processes in the peer review, the communication of organizational practices incurred by diversity of experience also provides the opportunity for cultivating novices, reinforcing shared norms, and building social connections. However, the diverse values and motivations stimulate different opinions and negotiation about bug legitimacy, priority, community cultures, and norms. Such diversity challenges the mutual trust and appreciation among community members. Consequently, frustration and emotional conflicts arise, which may discourage community engagement or even result in member withdrawal.

Our findings both confirm and complement the current understanding of diversity and its effects. The benefits of diversity of expertise, knowledge and resources we identified are in accordance with previous literature on diversity—informational diversity can serve as a resource to enhance work performance (Harrison & Klein, 2007; Van Knippenberg & Schippers, 2007). But these benefits manifest themselves in the OSS community differently from workgroups with clearly defined group or organizational boundaries. Deviating from prior research, configuration of diverse expertise, knowledge and resources is not pre-selected and participants are not obliged to remain committed in the OSS context. These results also differ from existing research on OSS bug reporting—Ko et al. (Ko & Chilana, 2010) emphasized the cost caused by large numbers of inexperienced reporters and advocated for recruiting more talented volunteers. However, our
study shows that OSS participants differ not only in their technical expertise, but also their perspectives and resources that they add to the OSS community.

Diversity of experience in our study is relevant to tenure diversity discussed in other studies (Chen, et al., 2010; Daniel, et al., 2013; Williams & O'Reilly, 1998). Both Chen et al. (Chen, et al., 2010) and Daniel et al. (Daniel, et al., 2013) operationalized tenure in online volunteer groups by calculating the length of time a participant first registered for an account on the respective platform. Chen et al. found that increased tenure diversity in Wikipedia increases group productivity and decreases member withdrawal, and Daniel et al. reported tenure diversity positively influences user participation. Researchers studying diversity in traditional organizations generally considered that tenure diversity is associated with less effective groups process with respect to integration, communication, and conflict. In contrast to all these studies, we observed more complicated and nuanced effects. Diversity of experience in involvement with the community can increase the workload of filtering bug reports and result in additional communication required for teaching novices how to report bugs and debug. Experienced community members have to explain organizational practices and norms.

Additionally, consistent with Jehn et al.’s (Jehn, et al., 1999) work, we also found diversity of values and motivations causes conflicts and frustrations. These effects were discussed as results of different organizational cultures in other studies such as (Hinds & Kiesler, 2002; Shachaf, 2008). However, in our context, such differences are not divided by organizational boundaries but largely associated with participants’ roles and expertise. We also want to emphasize that these types of diversity are not always distinct in practice. For example, individuals with different experiences may consequently espouse different values.

The existence of both benefits and challenges indicates opportunities to support the OSS peer review process by mitigating the downsides and augmenting upsides of diversity. We suggest four design implications to approach this goal.
Enabling Alternative Representations of Bug Reports

The challenge of increased workload and communication and the benefit of problem characterization entail special support for peripheral members, who do not contribute regularly but constitute the majority of the community. Moreover, people with different perspectives could become a resource for knowledge development in the communities. Current bug tracking systems are largely designed to support core developers’ work. Providing alternative representations of reported issues can increase the visibility of parallel ontologies, which may mitigate diversity.

One approach to reducing submissions of duplicate reports is to translate archived reports and ongoing resolving progress in an end-user-accessible way. For example, for reports or code changes regarding user interface, bug tracking systems can represent them in an interactive visualization that groups and displays them with each visible component of the browser interface. Whenever someone wants to create a report, s/he can hover over the browser component s/he has problems with to obtain a list of references to the issues filed under this component. This will help narrow down the duplicate candidates, though this alternative representation of bugs may not apply to issues that are embedded in components that are invisible on the browser interface.

Another approach to improve identification of duplicate reports before submission is to enable flexible creation of meta-data that bridge the gap of contributors’ varying use of terminologies to characterize a problem. Other than providing system-defined classifiers to maintain consistent structure within the code repository, allowing user-generated vocabularies can accommodate a greater variety of mental models of the software application. Tags may be a good design option for this purpose. Although they are usually personal and not always informative, tags can still serve as an extra index for searching in the system. Moreover, participants who were to submit a bug report but found out the issue had already been reported may add their interpretations of the problem as tags. Such accumulation can further increase the
likelihood of identifying duplicates through bug search. Additional analysis and visualization of the tags, such as aggregation and highlights, may also inform the core developers of how end-users perceive problems and issues that they deem to be more important.

In addition to preventing from filing duplicate bug reports, the ability to consolidate information from different reporters is equally important. As our results show, different reporters do not always report identical information on the same issues. Offering cross-references to duplicate reports is a simple way for aggregation, but lacks some integration. Tags may be an alternative: they are easier to be aggregated than bug reports; they would be useful if each of them is informative and distinctive.

Reducing the creation of invalid bugs can also address the increased workload of filtering reports. Users usually do not read through documentations that provide extensive instructions about how to participate in OSS peer review process, either because they are not willing to or because they are not aware of these documents. That is probably why more experienced contributors must repeatedly ask the novice contributors to repeat test cases in the safe mode and under a new browser profile. Minimizing the instruction to guide reporters to differentiate user support issues from real bugs before they submit a report may be an effective design. For instance, rather than archiving instructions in other community sites, a page before the report submission form can be designed in the bug tracking system, asking the reporter to try to reproduce the problem in safe mode or with a new profile.

**Enhancing Awareness and Mediation for Expert Matching**

Increased communication is partly due to failure of engaging the right experts in the first place, which might delay the progress of bug analysis and fixing. Being able to discern the expertise and availability of developers requires long-term involvement or at very least—long-
term observation—in the peer review process. However, this requirement contradicts the motivation of many contributors who just want to “scratch their itch”. Therefore, externalizing the knowledge of developers’ specialties and status, which is currently implicit and only accessible to co-developers in the community, can enhance the awareness of the general public and expedite the review process.

Assisting with finding the appropriate experts via social networks may enhance the effectiveness of collaboration. One possible approach is to construct dynamic profiles that automatically reflect developers’ activities on software products as well as other people who have interacted with them over these products. Another option could be visualizing the social networks of active community members based on their activities. Adding a social feature that shows the volume and status of developers’ work in the community can also help other participants decide whom they should ask. Developers of Mozilla appropriated their user names in Bugzilla to indicate the time period they will be unavailable, such as being away on vacation. This confirms the need of such a feature.

Besides relying on the crowd to locate experts or automatic recommendations from the system, another solution is to nurture a larger group of active members who are not core Mozilla developers to mediate expert matching. These contributors are motivated to maintain active and long-term involvement in the entire OSS peer review process. They do not have to acquire as much domain expertise as the core developers, but they have more advanced knowledge than the mass for refining bug classification. Such triage can increase the visibility of bugs relevant to individual developers, assisting with connecting peripheral contributors and core members. To sustain this group of contributors, the community needs to show appreciation and acknowledge the accomplishments of these triagers who fill the essential roles of bridging end-users and core developers, which has not been the case based on our observations. One possible improvement is to make the responsibilities and the role of these triagers explicit, such as organizing a formal
team to work specifically on bug prioritization, turning the invisible work to visible (Suchman, 1995). It may also be beneficial to expand the types of work this group of people do to distinguish them from the rest of the community. For instance, other than linking relevant bug reports to core developers, they can also help build social connections between end-users and core developers.

Locating experts may be an efficient way to fix a bug, but it may inhibit knowledge sharing and development within communities and impose too much workload on core members. Cultivating a learning community of active members who can analyze problems and review patches can free up the time of the core developers while also disseminating and reserving knowledge even if an expert left the community. For instance, promoting a mentoring program can help newcomers identify their interests and match them with experienced developers who are able to provide them with specific advice. These personal interactions may enhance novices’ perception of engagement with the community, and thus retain long-term membership and sustain long-term contribution.

**Establishing Creditability, Trust, and Incentives**

The challenge of frustration and conflicts implies that participants do not trust the legitimacy of each other’s expertise. Bach et al. reported similar findings in their analysis on expertise in usability (Bach, et al., 2009). The trust is also critical to effectively integrating critiques raised in design review. Mozilla acknowledges volunteers who have made significant contributions by announcing their names in its weekly project meetings and core developers’ blogs. It also credits the ones who have made enough quality contributions with privilege of editing bug reports in Bugzilla. However, this coarse characterization of contributions is weak and insufficient.
Enhancing the visibility of contributions may facilitate trust building. One design solution is to create profiles for individuals that record past contributions and displaying them when the participants interact with others. Tracking the accumulation of experience and quality work may also enhance participants’ self-efficacy in producing valuable contributions to the community, motivating them to continue participating.

Additionally, a sophisticated evaluation system can address the variety of contribution types and motivations of participation. Currently, the community only maintains a web page that lists contributors who are identified by whether they wrote code or documentation or participated in testing tasks for Mozilla. A finer characterization of contributions would be more beneficial for rewarding participants. For example, a bulletin board that specifies different types of contributions and updates representative work for each type might be a design option to present quality of contributions rather than quantities. Furthermore, the bug tracking system may also allow contributors to rate their own contributions. A contributor can rate a comment highly if s/he finds the information useful for bug analysis. The contributors who have provided many highly rated contributions could receive special recognition in the community as a reward.

Creating Channels for Information Integration

Third-party contacts that bridge different work groups and connect OSS communities with other organizations do not participate regularly in OSS communities. Probing them only when issues arise is not always an effective practice. In contrast, keeping regular contact with them may augment the benefit of boundary spanning, forming early design ideas and foreseeing potential problems.

Leveraging new media, or even creating a new communication channel for debriefing the most important news and updates of other groups and organizations may facilitate information
exchange. Such information are currently scattered in various places and information sources, and most members do not have a clue of how to gather that. The integration of information sources provides the basis for content synthesis. Furthermore, updates like this do not have to be frequent or overly detailed. Have integrated information sources could also reduce redundant dyadic interactions. Mozilla currently hosts weekly project meetings, in which the user support team raises popular concerns or complaints in the support forum to the development team. It also implemented a blog feed that incorporates relevant blogs within the Mozilla community where members share their work progress and reflections in their blogs.

Limitations and Future Work

Our current analysis is constrained by its case selection and data sources. We chose Mozilla because of its member diversity, complexity and recognition; however, some of its characteristics differ from other OSS projects, which could be affected differently by the diversity dimensions described in this paper. Unlike purely volunteer-based communities, Mozilla is comprised of both paid members and volunteers. Moreover, Mozilla’s OSS practices may differ from those of smaller and decentralized OSS projects; it follows strict and formal procedures of peer review and clearly defines the roles and responsibilities of its core members. We focused our investigation on bug reports archived by bug tracking systems, even though the interactions in the peer review process may involve the usage of other computer-mediated communication tools. Other OSS projects that heavily rely on mailing lists to perform peer review may exhibit different effects of diversity. We plan to compare our findings with other OSS communities in future work.
Conclusion

Diversity is important to effective collaboration and innovation. Large-scale volunteer-based distributed online collaboration requires understanding of diversity and its effects. Our investigation on the peer review processes in an OSS community, Mozilla, shows that diversity brings both challenges and benefits to the online collaboration, increasing workload of screening bug reports and communication of work techniques and organizational practices, causing frustration and conflicts when negotiating bug legitimacy, priority, community norms, and cultures, enhancing problem characterization with complementary information from the crowd, facilitating design review through critiques and dissent, and supporting boundary spanning across workgroups, organizations, and even competitors. These results extend our understanding of diversity at a much finer level than prior work. They also complement the studies that were conducted on ad hoc teams in organizational and laboratory settings. We recommend some approaches to augment the benefits as well as overcome the challenges—enabling alternative representations of bug reports, enhancing awareness and mediation for expert finding, establishing creditability, trust and incentives, and creating channels for information integration. Our study also suggests that leveraging crowds to collaborative knowledge work not only requires mutual respect and appreciation, but also entails synthesis and integration.
Chapter 6

Lessons from the Classics: Differences between Mozilla and Python

Recent development of social media enables open source software projects to leverage a larger crowd than they ever accessed, fostering many new projects to emerge and grow. However, as they evolve, projects will face new challenges of engaging contributions and coordinating collaboration, and different projects need to design different socio-technical support to address these challenges. The study presented in this chapter characterizes the differences in collaboration between relatively established open source communities to provide design guidance for emerging open source software projects. I focus our investigation on the core work practices—peer review—of open source software development. By contrasting these processes in Mozilla and Python, I identify their differences of contribution styles, reporting motivations, decision-making processes, division of labor, and uses of tools. I further articulate design implications for OSS projects that are advancing towards their maturity.

As a pervasive and influential computing infrastructure that often relies on the involvement of virtual communities, OSS has been an appealing topic for Human-Computer Interaction researchers (Bach, et al., 2009; Ko & Chilana, 2010; Terry, Kay, & Lafreniere, 2010; Twidale & Nichols, 2005). Numerous OSS products have been produced over the past decade, targeting various types of users ranging from tech-savvy software developers (e.g., programming languages), professionals with some knowledge of programming but focusing on domains outside software development (e.g., bioinformatics software), to end-users (e.g., web browsers).

More recently, the advances of social media have instilled new excitement to OSS development, demonstrated by Github surpassing other open source forges by 2011 (Finley, 2011). This creates opportunities for OSS projects at the starting phase to leverage a large crowd
to grow (Dabbish, Stuart, Tsay, & Herbsleb, 2012). They will encounter new challenges of engaging contributions, coordinating work, and ensuring software quality as they grow to a certain extent (Lattemann & Stieglitz, 2005; Scacchi, 2006). Therefore, reflecting on the practices of well-established large OSS communities will inform emerging OSS projects of how to resolving such challenges.

To provide contextualized guidance for growing OSS projects, we contrast two large OSS communities, Mozilla and Python, which are both established and well-recognized but serve different types of users—software developers and end-users. Although prior research has compared OSS projects at different stages of evolution (Lattemann & Stieglitz, 2005; Scacchi, 2006), very little work has been done to contrast OSS communities at similar stages, which are likely to differ in the ways community members collaborate when the communities are comprised of different types of users. Unlike typical comparative case studies designed to identify commonalities across cases, our investigation was intended to highlight the variances between the two communities.

We focus on one of the core collaborative work practices in OSS communities—peer review. Raymond coined “Linus’s law” (“given enough eyeballs, all bugs are shallow”) to emphasize the power of extensive peer review in OSS development (E. S. Raymond, 2001). During peer review, community members evaluate and test software products, identify and analyze defects or deficiencies, and contribute and verify solutions (e.g., patches) that repair or improve them. By examining interactions recorded in bug reports archived in their bug tracking systems, we identify differences of peer review processes in Mozilla and Python in their contribution styles, reporting motivations, decision-making processes, division of labor, and uses of tools. We further articulate the design implications that emerged from these different characteristics for a growing OSS project to effectively keep engaging contributions, coordinate work, and ensure software quality. Our reflections on the classic models of OSS communities are
expected to shed light on designing and managing newly emerged OSS projects as well as designing tools supporting large-scale software peer review.

Related Work

Overview of Open Source Software Development

Despite tremendous research efforts on OSS development from various fields, very little work was conducted to understand the differences between OSS communities. Researchers have largely been focused on contrasting OSS development with traditional models of collaborative software development. They attributed the success of this unconventional approach to intrinsic motivations (Roberts, et al., 2006), high modularity of software (Bonaccorsi & Rossi, 2003), informal and flexible development processes (Scacchi, 2002), extensive peer review (E. S. Raymond, 2001), and rapid releases (J Feller, et al., 2005).

OSS studies that analyzed multiple projects were primarily focused on their commonalities for generalizability. The well-cited work from Mockus et al. (Mockus, et al., 2002) examined early stages of Apache and Mozilla. They generated hypotheses by quantifying aspects of developer participation, core team size, code ownership, productivity, defect density, and problem resolution intervals of Apache, and then used Mozilla to confirm and refine these hypotheses. Yamauchi et al. (Yamauchi, et al., 2000) contributed early understanding of collaboration in OSS projects by analyzing two cases—FreeBSD Newconfig Project and GNU GCC Project. They found that in both projects, “spontaneous work coordinated afterward is effective and rational culture helps achieve agreement among members” and “communication media (CVS, TODO lists and mailing lists) are used in a good balance between centralization and spontaneity.” More recently, Rigby et al. (P.C. Rigby & Storey, 2011) sampled five OSS
projects—Apache, Subversion, FreeBSD, Linux, and KDE—to achieve a general understanding of the broadcast-based patch review processes in OSS development.

A few studies on the evolution of OSS and its communities examined differences between projects that are at different stages—introduction, growth, maturity, and decline or revival (e.g., (Lattemann & Stieglitz, 2005; Scacchi, 2006)). They were focused on the changes of software products themselves (e.g., (Fernandez-Ramil, Lozano, Wermelinger, & Capiluppi, 2008)) or the evolution of individuals’ roles in the community (e.g., (Kumiyo Nakakoji & Yamamoto, 2005)). Additionally, Lattemann et al. (Lattemann & Stieglitz, 2005) claimed that the motivations and objectives of members are also different at different life cycle stages.

In contrast to the existing multi-case studies, our work attempts to address the lack of characterization and articulation about differences between OSS communities. In particular, our study is aimed at investigating different OSS communities at similar stages, unfolding the differences of how their members collaborate with each other. Our comparison is not meant to judge which community is better; instead, our goal is to understand how we can better support different OSS communities through such a comparative analysis.

**Open Source Software Peer Review**

We consider OSS peer review evaluation of the entire software application, which is closer to the original meaning of “extensive peer review” as highlighted in OSS literature (E. S. Raymond, 2001). The review process often begins with one submitting a bug report. Others diagnose the defect causes and request additional information to determine whether the bug should be fixed. Once a solution is reached, they then commit a change set (mostly a patch) to the current software product.
OSS peer review has been a topic of interest for HCI researchers because its unique power lies in its openness to users, which is always the focus of the HCI community. Recent work by Ko et al. (Ko & Chilana, 2010) analyzed the contributions of power users, who reported problems but were never assigned problems to fix, to bug reporting in Mozilla. Breu et al.’s (Breu, et al., 2010) examination of bug reports from Mozilla and Eclipse showed that users are not just bug reporters: their active and ongoing participation is crucial for making progress of fixing the bugs they reported. They emphasized the central role of bug tracking systems in supporting collaboration between developers and users of a software project. Based on their analysis, Breu et al. suggested four new ways to improve bug tracking system: (1) evolving information needs, (2) tool support for frequent questions, (3) explicit handling and tracking of questions, and (4) community-based bug tracking. Despite those few studies, the majority of studies on OSS peer review were conducted from the software engineering perspective, deliberately modeling the information needs in bug report quality (Breu, et al., 2010), inaccurate bug assignment (Jeong, et al., 2009), efficiency and effectiveness of patch review (P. Rigby, et al., 2008), and distribution of contributions in bug reporting and fixing (Mockus, et al., 2002).

Research has also identified ways other than traditional OSS peer review processes to enhance the code review process. For example, open source code repository platforms such as GitHub (Dabbish, et al., 2012) use contributors’ reputation on past work and social profile as a preliminary way to infer code quality. However, these projects tend to be much smaller; they do not require an extensive peer review process, as the project owner is typically the only person responsible for approving changes to the code module. Thus, we focus our study on OSS communities that involve large-scale collaboration and have established peer review processes.
Technology Support for Open Source Software Peer Review

OSS peer review is supported by a variety of work-related tools and communication technologies. Bug tracking systems serve as the central place of collaboration for most large OSS projects. They help developers keep track of reported defects or deficiencies of source code, design, and documents. Version control systems manage and synchronize the occurred code changes in the OSS peer review process, while communication tools such as mailing lists and Internet Relay Chat (IRC) enable developers to have additional discussions on bugs.

Bug tracking systems have been widely adopted in large-scale OSS peer review practices, because scaling becomes increasingly difficult for mailing lists despite developers’ high familiarity with emails. Besides functioning as a database for tracking bugs, features, and other inquiries, bug tracking system also serves as focal point for communication and coordination for many stakeholders (e.g., customers, project managers, quality assurance personnel within and beyond software development team) (Bertram, et al., 2010). Bugzilla is employed by many large OSS communities, which include Mozilla, GNOME, Netbeans, and Apache (Halloran & Scherlis, 2002). It is relatively sophisticated, sharing the common features with most other bug tracking systems. These features consist of customizable fields, email notifications, report exportation, discussion spaces for each bug report, and simple charts visualizing the overall characteristics of activities in the entire system. Although the design of bug tracking systems has attained some incremental improvements, many of them remain merely better interfaces to a database that stores all reported bugs (Just, Premraj, & Zimmermann, 2008). Their designs have largely overlooked the different types of users and distinct organizational contexts.

Recent development of social media brings new options for software developers to coordinate with peers, to communicate with and learn from users, to become aware of technology development, and to create informal documentation (Storey, Treude, van Deursen, & Cheng,
Despite the great promise, existing practices and systems only have some initial experimentation with social media features; it is still unclear how to incorporate social media to benefit the work and social processes of OSS development (Begel, DeLine, & Zimmermann, 2010; Storey, et al., 2010). Github and Ohloh.net provide social networking functionality at their integrated platforms for OSS developers to follow other people’s work activities and develop social connections. More general social media tools, such as Twitter and Yammer, were used to make announcement, mobilize and support end-users, and keep co-developers aware of one’s work status (Begel, et al., 2010; Jing Wang & Carroll, 2013). Nevertheless, obtaining information from these general media sources remains ad hoc, affording deliberate designs to incorporate it.

To enhance the understanding of how technologies can better support OSS peer review processes, our study extends the prior research with a focus on different types of users and community contexts, elaborating contrasts in contribution styles, reporting motivations, decision-making processes, division of labor, and tool uses between two distinct OSS communities. The findings of this study could generate additional insights that could ease coordination, communication, and collaboration in younger OSS communities.

Methods

Case Selection and Description

Case studies have been widely used as a way to generate useful insights from successful exemplars in a variety of domains, including open source software development (Breu, et al., 2010; Yamauchi, et al., 2000; Yin, 2008). We selected Mozilla and Python for comparison. They both have relatively established and stable peer review practices, which enabled us to observe substantial instances of collaboration in their peer review processes. In addition, they have been
less intensively studied than Linux and Apache, especially Python, as indicated by Crowston et al.’s review (K. Crowston, et al., 2012).

An important reason we chose Mozilla and Python is their different user bases, which also led to different characteristics of community composition. Mozilla is mainly focused on producing end-user applications, such as Firefox, engaging a highly heterogeneous community. In contrast, Python develops a computer programming language by a relatively more homogeneous community. Mozilla divided its core members into module owners, module peers, super reviewers, security group members, and quality assurance members. Each individual had specified areas to maintain and facilitate evolvement. For example, Firefox had a module Session Restore, which was supported by one module owner and three peers. Any changes toward this module had to be reviewed and approved by one of them. Python classified its core developers in a relatively loose fashion. The core members share the same privileges over the software code repository.

Aside from the differences, Mozilla and Python share general characteristics of community structure and supporting technologies with other established OSS communities. Community members include both core developers who contribute significantly to the project and peripheral members who use the software but rarely participate in developing the software. Both communities have a Benevolent Dictator for Life (BDFL) who makes the final say over technical disputes. Mozilla and Python also have similar computing infrastructures that support their bug fixing activities, including version control systems (Mercurial for Mozilla and Subversion for Python), bug tracking systems (Bugzilla for Mozilla and Roundup for Python), IRC, mailing lists and wikis. For code review, Bugzilla implements a patch reviewer tracking code changes, while Python recommends an external code review tool Rietveld. Additionally, Python creates Python Enhancement Proposals (PEPs) for initial design ideas.
Data Collection and Analysis

Our current analysis draws on bug reports of Firefox and Python language from the bug tracking systems of Mozilla and Python. Bug tracking systems are one of the primary data sources of bug fixing practices in large OSS projects. Sampling from two core products, Firefox and Python language, helped focus our investigation. Although the data analysis focused on bug reports archived in the bug tracking systems, our interpretation was also informed by a variety of other relevant data sources. For Mozilla, we examined 41 weekly meeting notes, 22 web documents describing work procedures and member roles, and 6 blog posts from individual members discussing Mozilla’s peer review processes. Python has slightly different peer review practices: mailing lists were also sometimes in the developer community to discuss controversial issues; although Mozilla also maintains a super-review mailing list for patches that need super reviewers’ assessment, the conversations at this list were duplicate of discussions archived in bug reports. Thus, we searched for the ID of bug reports we sampled through the archive of the Python developer mailing list, which is the list that allows bug discussions. This returned us 15 email threads. Another difference is that Python did not hold regular meetings to manage its software development. The similar data sources we used from Python’s public web resources consisted of 2 blogs explicitly discussing the peer review process from the active contributors of Python and 17 web documents describing work procedures and member roles. Throughout the duration of data collection and analysis, I also actively participated in developer and community meetings and IRC discussions as a participant in these communities. The participation helped clarify and validate concepts whenever issues arose during the data analysis.

We retrieved bug reports created between two recent stable releases of Firefox and Python for analysis. The retrieval was performed on July 28, 2010. This sampling strategy was intended to capture the possible behavioral changes near releases (Francalanci & Merlo, 2008).
For Mozilla, 7322 reports were filed between the releases of Firefox 3.5 final and 3.6 final. For Python, 1850 issues were generated between the releases of Python 3.0 final and 3.1 final.

We used email addresses to identify Mozilla’s contributors and Roundup user names to identify Python’s contributors. Core developers of Mozilla Firefox were defined as the ones listed on Mozilla’s websites as module and sub-module owners and peers, Firefox development team members, super reviewers, security group members and members with additional security access, quality assurance team lead, and platform engineers during the time between Firefox 3.5 and 3.6 releases. 176 developers were identified in this category. For Python, core developers were defined as committers listed on the community website between Python 3.0 and 3.1 releases. Crowston and Scozzi suggested that it may be more useful to define core developers based on the amount of their actual contributions rather than their job titles (K. Crowston & Scozzi, 2002), but our approach may better differentiate the incentives and responsibilities of core developers from normal volunteers.

Overall, we sampled 10% of the bug reports of each project from our retrieved data set, which included 732 Firefox bug reports and 185 Python issue reports. To accommodate the possible variations of work processes and social interactions across bug reports, we performed stratified sampling with respect to bug statuses and resolutions. In Bugzilla, open bug reports have 4 types of statuses—unconfirmed, new, assigned, and reopened. Closed bug reports have 6 types of resolutions—fixed, duplicate, invalid, incomplete, worksforme, and wontfix. In Python, closed issue reports have 10 types of resolutions—fixed, duplicate, invalid, worksforme, wontfix, accepted, later, out of date, postponed, and rejected. The rest of the issue reports have 3 types of statuses: open, pending, and languishing. For each of the 10 types of bug statuses and resolutions in Mozilla and 13 types of bug statuses and resolutions in Python, we sampled 10% of them: we first intentionally selected the most highly discussed bug reports with a number of comments at the 98th percentile; then we randomly sampled the rest. Each bug report has its unique ID in both
Bugzilla and Roundup. We ordered the comments for each bug report based on their posted time, numbering from 0 for the oldest comment. When quoting discourses in the next section, we use the following format—(bug ID; comment number; roles in the bug report; roles in the community; case name).

Our qualitative analysis adopted an open coding approach to discover patterns and recurring themes. The iteratively refinement of themes was comprised of four phases. First, the first and the second authors randomly selected 50 bug reports from both cases and read them separately. They discussed their observations and established a shared understanding of the peer review process in Mozilla and Python. Then during the second phase, the first author coded the 732 Firefox bug reports, iteratively generated 628 codes by discussing with the other authors during their weekly meetings. The first author continued to code the 185 Python issue reports mainly for identifying the differences from our observation of Mozilla. Additional 174 codes were generated during the third phase. The first author discussed the new codes as well as observations other than the discourses archived in bug reports with the other two authors to further codify the differences between cases. Finally, 16 groups of codes that occurred the most frequently and were deemed the most relevant to differences between the two cases were selected. Overall, the 16 groups of codes occurred 203 times, accounting for 10.87% of total occurrences. This ratio does not seem to be very high because we did not code every difference, such as some characteristics that occurred too frequently. For instance, Mozilla’s patch reviewers usually did not modify patches by themselves, but this characteristic happened in almost every bug report that had patches (n=494). Then the authors integrated the codes with observations other than discourses into 5 themes, which are presented in the results section.
Results

A quantitative overview of the activities involved in the peer review process captured by our data sets shows remarkable differences between Mozilla and Python, in terms of activity intensity and participation scale (see Table 6-1). Within similar lengths of time—approximately half a year between two official releases of products—Mozilla reported a much larger number of bugs (n=7322) from a larger group of people (n=5419) than Python did. The contributors involved in the Mozilla peer review process through both reporting and commenting on bug reports (n=6704) were also of much larger scale than those in Python (n=1188).

Table 6-1: Overview of peer review processes in Mozilla and Python.

<table>
<thead>
<tr>
<th></th>
<th>Mozilla</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bugs reported</td>
<td>7322</td>
<td>1850</td>
</tr>
<tr>
<td>Distinct bug reporters</td>
<td>5419</td>
<td>829</td>
</tr>
<tr>
<td>Distinct bug contributors (reporters and commenters)</td>
<td>6704</td>
<td>1188</td>
</tr>
<tr>
<td>Core developers</td>
<td>176</td>
<td>127</td>
</tr>
<tr>
<td>False positives (non-real bugs)</td>
<td>3418 (46.68%)</td>
<td>469 (25.35%)</td>
</tr>
<tr>
<td>(Ratio among bugs reported)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fixed bugs</td>
<td>498</td>
<td>832</td>
</tr>
</tbody>
</table>

Bugs filtered out, namely false positives, in Mozilla (n=3418; 46.68%) were significantly more than in Python (n=469; 25.35%). These included bugs closed as duplicate, invalid, incomplete, wontfix, and workforme in Mozilla, and issues closed as duplicate, invalid, rejected, wontfix, worksforme, out of date, later, and postponed in Python. Additionally, bug reports that had not been confirmed as real bugs constituted a significant portion of the Firefox open bugs (n=2680).

Our subsequent qualitative analysis identified more differences of collaboration during peer review processes between Mozilla and Python. The results show that the two communities differ with respect to contribution styles, reporting motivations, decision-making processes, division of labor, and use of tools.
Contributing by Talking vs. Acting

Although communicating opinions and generating code are common peer review activities in both Mozilla and Python, participants in Mozilla seemingly tend to contribute by talking (i.e., stating ideas), whereas those in Python collaborate often through acting (i.e., implementing solutions). In Mozilla, 1170 patches were created by 111 authors for 494 bug reports. In contrast, 1694 patches were uploaded by 295 authors for 562 bug reports in Python. In other words, 1.66% of contributors in Mozilla (n=6703) contributed patches, and the ratio was as high as 24.83% in Python (n=1188). Moreover, 6.75% Firefox bug reports had patches (n=7322), whereas the portion for Python was 30.38% (n=1850).

The differences of contribution styles emerged first from the ways of reporting bugs. Compared to Mozilla, reporters in Python were more likely to submit bug reports with patches. Regardless of bug legitimacy, reporters were also the patch authors in 276 Firefox bug reports (3.77%; n=7322). In contrast, reporters of 260 Python issues created patches for their own reports (14.05%; n=1850). The following episodes illustrate the different reporting styles in Mozilla and Python.

“The left-most tab title display conflicts with the close button. It seems to require installing a person[a] first - the default with no personas works. If the title is long and disappears off the tab." (Bug 530108; comment 0; reporter; peripheral participant; Mozilla)

“This patch takes the existing "simplegeneric" decorator, ... and exposes it as a feature of the functools module. Documentation and tests have been added, and the pkgutil code has been updated to use the functools implementation.” (Issue 5135; message 0; reporter; peripheral participant; Python)

The preference of acting over talking in Python also appeared when developers were reviewing patches. In Mozilla, patch reviewers usually only gave comments, either questioning the overall design of patches or suggesting specific code changes line by line and letting the original patch author modify the patches. In the following example, the reviewer pointed out a
flaw of the patch design and suggested the direction of modification. The patch author then indicated his lack of knowledge to achieve this in the way the reviewer wanted and asked for further help from the reviewer. Eventually the author did not update his patch, nor did the reviewer.

“I disagree on the l10n part here, at least for the source of it. The plugin should read a plain text file from a known good location, and that text file needs to be create by the repackaging process out of one of our known mozilla l10n file formats.” (Bug 524338; comment 8; patch reviewer; core developer; Mozilla)

“Can you point me at an example of such a file (or instructions on how to integrate such a file into the l10n build system) so I can switch the code to using it? I'm not familiar at all with the Mozilla l10n repackaging process.” (Bug 524338; comment 9; patch author; peripheral participant; Mozilla)

In contrast, when reviewing patches in Python, developers often modified the patches by themselves and left summaries as comments in the bug reports if the changes were minor. If they did not agree with the design, they created a new patch to implement their own approach. The illustration below shows that a patch reviewer rewrote the patch and elaborated why his approach was better after the original patch author misunderstood his recommendation.

“...I don't think that's relevant to the point I was attempting to make ... I'm working on refactoring PyLong_AsScaledDouble and PyLong_AsDouble to have them both use the same core code. This would make PyLong_AsScaledDouble correctly rounded...” (Issue 5576; comment 3; patch reviewer; core developer; Python)

The different preferences between talking and acting were demonstrated not only by how the contributors themselves behaved, but also by how other participants interacted with them. In both Mozilla and Python, experienced contributors would explain the organizational practices to the relatively inexperienced ones when their contributions showed poor quality. However, Mozilla contributors often taught how to report bugs (i.e., talking), such as what type of information the reporter should provide for the bug. For instance, a Firefox end-user submitted a bug report titled “Doesn't preview in a browser when choose Firefox 3.5 from Dreamweaver CS4
and CS3” with a brief description of his experience. Another participant who was an experienced contributor in the community was confused, asking for clarification.

“Could you please read [a link] and comment with what exactly the problem is, exact steps to reproduce the issue, screenshots/screencast if needed, also if there are any errors in the error console.” (Bug 502022; comment 1; commenter; active contributor; Mozilla)

In contrast, advice or instructions provided by Python participants involved how to submit patches (i.e., acting), such as adding documentation and unit tests to patches.

“I have created a small patch, that adds method that formats using a dict. It's the first time I have written anything in python implementation, so I would very appreciate any advice.” (Issue 6081; comment 2; patch author; peripheral participant; Python)

“Thanks for the patch. It would be nice if you could include unit tests too.” (Issue 6081; comment 3; commenter; core developer; Python)

**Reporters Pulling vs. Pushing Help**

Relevant to contribution styles, the motivations of bug reporters in Mozilla and Python were also distributed differently. Contributors might have been motivated by various incentives as identified in OSS literature (Roberts, et al., 2006), including utilitarian reasons (“scratch own itch”), joy and challenge, reputation, and even monetary rewards; however, in our cases, reporters in Mozilla more often reported bugs to pull help from others, whereas those in Python submitted bug reports for pushing others help. This difference was partly suggested by the much larger percentage of reporters who provided patches for the bugs they reported in Python (14.05%) than in Mozilla (3.77%) as mentioned earlier. The difference in motivations was also inferred from reporters’ articulation when they described bugs at the beginning of their reports, although we did not directly interview them.
Specifically, many reporters in Mozilla reported bugs because they experienced software dysfunction but were not capable of either diagnosing the problem or fixing it. They contributed in order to seek support from developers.

“[I] cant install pics for uploading to another site (snapfish) for printing..it doesn't do anything on any website when i try to upload pics (facebook)..is there something i need to download in order for mozilla to open these pics files?? ...”
(Bug 505703; comment 0; reporter; peripheral participant; Mozilla)

Comparatively, Python reporters already found the solutions to the problems they had encountered before submitting their bug reports. The reason that they took the time to describe the problem and upload fixes was to benefit other users, improve the software, or at least share their findings. The example below shows that a reporter suffered from the deficiencies of Python’s design and spent a lot of effort developing a solution. He contributed back to the Python community for others who might have similar experiences.

“When I first came across decorators (about a year ago?), they confused the hell out of me, and the syntax is completely google-proof, so I ended up having to ask on irc. One of the things that I tried was help('@') but that didn't work either. This patch is to make that work, and also a lot of other google-proof syntax constructs. ... I hereby donate my code to whoever wants to commit it. Do what you want with it, and clean it up however you want.” (Issue 4739; comment 0; reporter; peripheral participant; Python)

**Individual vs. Collective Decision-making**

One remarkable difference between Mozilla and Python lies in how they make decisions. Things that need to be decided mostly consist of bug legitimacy and patch design. Although both Mozilla and Python have BDFLs who give a final say over technical disputes, both BDFLs only participate in a small number of discussions: Mozilla’s BDFL commented on 2 bug reports in our data set (0.03%; n=7322) and did not report any bugs; Python’s BDFL was relatively more active, contributing to 27 bug discussions (1.46%; n=1850) and creating 3 bug reports (0.16%; n=1850). Consequently, decisions regarding the majority of bugs rely on other core developers’ judgment.
Participants have to make significant and quality contributions to the community in order to become candidates of core developers, which is the common rule (i.e., meritocracy) among OSS projects.

For a specific bug, Mozilla authorizes its affected module’s owner to decide whether a change is appropriate for the module, especially when only one module will be affected. For instance, a Firefox reporter criticized that “[f]avicons don't work unless using ‘Remember’ privacy setting”. Two core members were inclined to set the bug as wont fix, but the code owner decided to accept the request.

“… under the logic that this is a violation of your privacy settings because you're telling the browser not to save your history, creating a bookmark at all is in violation of the privacy settings under the same rationale.” (Bug 502787; comment 5; reporter; peripheral participant; Mozilla)

“I'll leave this decision up to the module owner then...” (Bug 502787; comment 6; commenter; core developer; Mozilla)

“I agree with the reporter. If the user asks for a bookmark, we should store it's favicon, regardless of the history setting.” (Bug 502787; comment 7; commenter; module owner; Mozilla)

Sometimes other Mozilla contributors (e.g., end-users) criticize this way of making decisions, and even emotional conflicts arise. These contributors feel that they have a stake in the software design, but they are excluded from the decision-making process and their opinions are unappreciated. For instance, a Firefox core developer decided to remove the “Properties” context menu item though several users argued that this item had been useful to them. Many Firefox users complained, but all the core developers participating in the discussion refused to reverse the decision.

“Sometimes the democratic system works. Sometimes a group with a very closed perspective hijack it. Congrats on making [Firefox] a laughing stock of a browser.” (Bug 513147; comment 70; commenter; peripheral participant; Mozilla)
“Mozilla is a meritocracy, and as such, the module owner/peers have final say over decisions concerning the codebase.” (Bug 513147; comment 79; commenter; core developer; Mozilla)

Different from individual decision-making, Python mostly approaches decisions collectively. It gathers feedback and judgment from core developers who are interested in the issue through informal voting. The voting emphasizes the value of providing rationale together with voting numbers. It does not have “a binding force” (Warsaw, 2007), but agreement often emerges from argumentation—either some parties are persuaded by the rest, or all the parties reach a middle ground. Otherwise, the majority wins. Such negotiations often occur outside the bug tracking system, such as mailing lists. For example, two Python core developers were rewriting “the IO stack in C” to enhance the performance of Python 3.0 until they had to make a choice of whether to maintain both the original Python implementation of the IO file and the new C version. One of the two developers suggested only keeping the C implementation, but another developer who joined the conversation later disagreed. They directed the issue onto the mailing list for core developers: 7 developers argued for maintaining both versions, while 3 felt neutral and 2 supported to drop the old Python version. Ultimately, they followed the majority’s opinion.

“I think we should just drop the Python implementations. There's no point in trying to keep two implementations around.” (Issue 4565; comment 11; patch author; core developer; Python)

“I disagree. I've found great value in keeping a pure python version around for things I've converted to C. The former serves as documentation, as a tool for other implementations, ... and as a precise spec. The latter case is especially valuable (otherwise, the spec becomes whatever CPython happens to do).” (Issue 4565; comment 16; commenter; core developer; Python)

“It seems the decision of Python-dev is to keep both implementations. We'll stuff the python one in _pyio and rewrite the tests to test both.” (Issue 4565; comment 21; patch author; core developer; Python)

Voting does not happen every time disagreement arises. Instead, if the developers currently involved in a bug discussion at the bug tracking system do not consider the issue critical enough to invite broader participation from the community, they are prone to reaching a middle
ground or even compromising. In the following excerpt, two core developers argued that a singleton was not a good data structure to implement the patch. Another core developer disagreed and proposed an alternative solution that both sides could accept.

“I also agree with [core developer 1] that a singleton looks rather unnecessary...” (Issue 5094; comment 18; commenter; core developer; Python)

“I still don't understand your aversion to singletons and you did not address any of the advantages that I listed in my previous comment. I don't think singletons are foreign to Python: after all we write None rather than NoneType(). We can reach a middle ground by ... This will address most of the issues that I raised and utc = datetime.UTC() is simple enough to write as long as you don't have to worry about sharing utc instance between modules.” (Issue 5094; comment 19; commenter; core developer; Python)

**Designated vs. Voluntary Responsibilities**

Related to the different decision-making rules, the division of labor also varies between Mozilla and Python. Given the increase of the complexity and the variety of work activities in large-scale OSS communities, the responsibilities are defined in an increasingly articulated way. For instance, both Mozilla and Python have non-profit organizations, in which their members taking various positions supporting the product development and community evolvement.

An important work responsibility for the OSS peer review process is evaluating patches, which is a required duty for relevant module owners or peers in Mozilla but not to any specific core developer in Python. Specifically, patch authors have to explicitly request the designated reviewers (usually 2 or 3 for one module) for assessment (i.e., the affected module’s owner or peers) in order to integrate their patches into the code repository of Mozilla. Any other participants can provide a review but cannot substitute the owner or peers to approve any changes. Some bugs require more than one patch reviewer when the solutions involve special expertise, such as UI design and localization. These experts are also comprised of a very small number of core developers (usually only 1). Upon submitting a patch, its author needs to set the
review flag in Bugzilla as “review?” with a reviewer’s email address, which will alert the reviewer by email if his/her local notification setting is enabled. The patch review process cannot proceed without the designated reviewers’ participation.

In contrast to Mozilla’s strictly defined patch review responsibilities, Python does not specify owners for each module and allows any core developer to evaluate patches voluntarily and accept code changes. We did not observe any patch author in Python explicitly asked for a certain reviewer but only leveraged the bug tracking system or mailing lists to highlight the need of review. Such vaguely divided responsibilities do not ensure someone is obliged to review patches. Therefore, sometimes it may inhibit the progress of the peer review process. The episode below illustrates a patch author complaining about the lack of response towards his patch and attributed it to the absence of a module owner.

“I don’t understand why this is so difficult to review...” (Issue 5949; comment 10; patch author; peripheral participant; Python)

“Rest assured it has little to do with the difficulty of reviewing it. Rather we are all volunteers.” (Issue 5949; comment 11; commenter; core developer; Python)

“... I understand that we are all volunteers here. My frustration in the lack of a de facto owner of the imaplib module and not with you personally or any other committer for that matter.” (Issue 5949; comment 12; patch author; peripheral participant; Python)

Other than reviewing patches, the responsibility of triaging—categorizing and prioritizing bug reports—is also fulfilled differently. Neither Mozilla nor Python defined triagers as a formal role in the communities. The task of triaging was open to public participation, but it was performed repeatedly by a relatively stable and dedicated group of active contributors in Mozilla. To reward their contributions, Mozilla named these people Friends of the Tree, of whom we identified 12 were involved in the bug discussions we retrieved. In contrast, the triaging role in Python was even more informal and was played by participants who mostly made ephemeral contributions.
Articulation vs. Appropriation of Tool Use

Aside from the differences of collaboration during the peer review processes, Mozilla and Python are also distinct from each other in terms of their bug tracking systems and the ways their contributors interact with these tools. Both projects primarily depend on bug tracking systems to conduct their peer review, unlike many other small OSS projects that use emails to achieve this purpose.

Bugzilla for Mozilla and Roundup for Python offer similar basic support, including a summary of the key information about bugs in a structured fashion (e.g., title, bug status, resolution, priority, keywords, affected component, affected version, dependencies, report creator and created time, last modification performer and time, assignee, a list of people who have subscribed to the bug report), a space for uploading patches, a space for discussion, and a log of report change history.

Despite the similarities, the designs of Bugzilla and Roundup are different in several aspects. First, the flags for a field in the summary section are defined by different lists of values. For instance, bug statuses in Bugzilla can be chosen from 6 types, unconfirmed, new, assigned, reopened, resolved, and verified. These are simplified into 2 options in Roundup, namely open and closed. Roundup also has two additional types of bug statuses, pending and languishing. Moreover, Python provides an extra field, stage, to indicate the progress of peer review. The flags for this field consist of unit test needed, needs patch, patch review, commit review, and committed/rejected. Other than the variations of summary indices, the patch viewers in both systems are also designed differently. Bugzilla has a relatively sophisticated patch viewer, in which developers can directly review patches by contrasting differences between versions and editing comments on code. Roundup does not support patch review and only displays patches in plain text.
Although both Bugzilla and Roundup enable the flagging of bug statuses and resolutions, Mozilla contributors follow the predefined values and mapping in a very articulate way, whereas Python participants tend to label them more loosely. Compared to the consistent mapping between status and resolution in the bug reports of Mozilla, the status of 16 bug reports in our Python data set were flagged as open but had a resolution. Furthermore, the stage field was not always used in Python: the stage was not indicated in 1387 reports (74.97%; n=1850).

The informal use of bug tracking systems in Python may sometimes create ambiguities or difficulties in managing information. For instance, the component field in Roundup provides a list of values that combine both platforms (e.g., Windows, Macintosh) and modules (e.g., installation, library, tests), which are defined separately in Bugzilla. Although this field allows multiple sections, participants may only assign a single value. In the following episode, a core developer of Python who was an expert of Macintosh found himself almost having missed a relevant bug.

“I have two procedural questions: 1) Who should I contact to get e-mail for all bugs that get tagged with the Macintosh component? 2) Please tag all mac-related bugs with the 'Macintosh' component, that's the one I most often check for new issues.” (Issue 6154; comment 21; patch reviewer; core developer; Python)

Another difference of how contributors interact with bug tracking systems emerged from the use of the assignee field. Neither Bugzilla nor Roundup allows multiple people to be set as an assignee at the same time. In Mozilla, the value of assignee is used consistently to indicate the patch author of a bug. In contrast, Python contributors appropriate this field to attract attention from a specific developer; the value of assignee could mean any type of role played in resolving a bug. For example, a developer assigned a bug report to the core developer who was managing the upcoming release of Python to check if the patch could be integrated into the next release.

“Probably too late for 2.6.3 - assigning to [release manager] to check anyway.” (Issue 5949; comment 5; commenter; core developer; Python)

“Unassigned from [release manager] since this isn't an [release management] review issue anymore.” (Issue 5949; comment 9; commenter; core developer; Python)
Understanding Variations of Open source software Development Models

After reviewing publications on OSS from multiple fields in the past ten years, Crowston et al. suggested “[f]uture research needs to compare projects … of varying types in order to advance our understanding of FLOSS development” (K. Crowston, et al., 2012). Our study is a step towards this goal. We are particularly focused on the projects that have established work practices, are well recognized, and keep evolving. Reflecting on the differences of contribution styles, reporting motivations, decision-making processes, division of labor, and tool uses, we articulate the underlying factors that may contribute to these variations. This can enhance our understanding of OSS development and the communities supporting the process.

The different types of products Mozilla and Python are developing have led to the differences in community composition and heterogeneity of expertise, which may further result in the different contribution styles and reporting motivations that we observed. Firefox, as an everyday web application, is likely to recruit a broader user base than Python, which is designed for programming: the majority of Mozilla community members are end-users, while the Python community is comprised of software developers. This determines members’ expertise as more diverse in Mozilla than in Python, which can bring both benefits and challenges to the peer review processes of the two projects. For Mozilla, the mass of end-users lacking technical expertise generated voluminous information of very limited value, increasing the overhead of processing it. Ko et al. reported similar findings that Mozilla bug reporters created many redundant and useless reports (Ko & Chilana, 2010). However, such diversity among contributors may also be resources that facilitate different perspectives, such as usability, to be considered and integrated into the work process. In contrast, becoming a user of Python already requires technical expertise. Therefore, even the peripheral members in Python are competent to write code. They can play a variety of roles in a bug discussion with quality contributions, from
reporter, analyst, to patch author and patch reviewer. However, the preference of code-speaking over word-speaking might discourage the articulation of design knowledge and rationale, which is harmful to the community’s long-term development.

The different types of products may also result in different cultures in Mozilla and Python. It is commonly believed by both scholars and practitioners that a distinctive feature of OSS development is its “hacker culture”, which contributes significantly to its success (E. S. Raymond, 2001; K. J. Stewart & Gosain, 2006; Von Hippel & Von Krogh, 2003; Weber, 2004). From our observation, this culture is widely appreciated within the entire community in Python: both peripheral and core members value code contributions and mutual help. However, such cohesion may not be easy to achieve in communities that produce software for end-users, who value usability most, like in Mozilla. Multiple subcultures exist in different groups of stakeholders in Mozilla. Consequently, negotiation is likely to go beyond technical virtues and fail when controversies arise, resulting in frustration and conflicts.

The different community sizes and participation scale of Mozilla and Python have also resulted in the differences with respect to power and roles, which are demonstrated by the different decision-making processes and ways of dividing responsibilities. Consistent with prior OSS literature (e.g., (Kogut & Metiu, 2001; Mockus, et al., 2002)), meritocracy explains how members gain power and resources in Mozilla and Python—through significant quality contributions. However, other than the BDFLs, the power is distributed more equally among core members in Python than in Mozilla. The decision-making process of Python is closer to the totally decentralized process in Apache, in which members adopt a voting mechanism to reach a consensus, and there is no single leader like a BDFL but a shared leadership (Fielding, 1999). In contrast, decisions in Mozilla are made in a more centralized fashion as those in Linux (Moon & Sproull, 2000). Besides the BDFL, no other core developers can veto the decision of the module owner or peers on issues specific to their module. In addition to power hierarchy, roles are also
defined differently in Mozilla and Python. Core developers and peripheral participants are the common roles in OSS communities as many other types of online communities. However, Mozilla further articulates the roles of its core developers and designates them specific responsibilities, such as module owners and peers maintaining their modules, super reviewers overseeing the changes affecting multiple modules, UI developers evaluating usability design, a localization team coordinating international versions of the software, and quality assurance people triaging and verifying bug reports. In comparison, the roles of Python core developers are not divided at this level.

The differences of tool uses and designs also reflect and reinforce the different cultures and practices of Mozilla and Python. Designing the structure of bug reports in a rigid and articulate way, as Mozilla does, enhances the ease of search and the quality of documentation. This is particularly beneficial for a community largely comprised of inexperienced participants, facilitating identification of duplicate bug reports and elicitation of critical information about the reported problem. However, such articulation may also increase the required effort to contribute, suppressing participants’ motivation. In contrast, Python allows improvisation of using its bug tracking system, which may respect the volunteers’ interests—coding rather than documenting, but such flexibility can decrease the accuracy of indices labeling bug reports, increasing the difficulty in becoming aware of bugs that call for attention.

Implications for Organizational and Technology Design

Our comparison of the peer review processes between Mozilla and Python shows that even having approached similar stages of their life cycles after overcoming the difficulty in achieving critical mass (Lattemann & Stieglitz, 2005), OSS projects still vary in the ways that members collaborate with each other and interact with the supporting tools, entailing different
strategies of designing socio-technical support to address the variations. Our analysis characterizes when, how, and why these variations of peer review practices emerge in different OSS communities: as a community dedicated to developing end-user oriented software applications, Mozilla has engaged a large number of contributions in the form of talking or expressing ideas. Moreover, its members report bugs for pulling help from the community, reach decisions through individual judgment, divide responsibilities to designated members, and use the bug tracking system in an articulate way. In contrast, Python is targeted at providing a programming language for software developers. Its contributors often make their contributions by acting on code patches, reporting bugs to help others in the community, reaching decisions collectively through voting or consensus building, leaving responsibilities to voluntary choices, and appropriating their bug tracking systems in various ways.

Our findings characterize the differences of the four common activities, which comprise the OSS peer review process as our previous work has identified, including submission, identification, resolution, and evaluation (J. Wang & Carroll, 2011). These contrasts afford specific and contextualized ways to support project and community sustainability. The key obstacles OSS projects face when they have grown to a relatively large scale lie in engaging contributions and coordinating work, which have been emphasized by literatures from both organizational science and software engineering perspectives (Lattemann & Stieglitz, 2005; Storey, et al., 2010).

To address the differences that emerged from our analysis, we suggest the design implications for OSS projects that are advancing towards their maturity to make their design decision by comparing their contexts with Mozilla and Python, in order to engage contributions as well as coordinate their efforts. Open source forges, such as Github and Sourceforge, may also provide configurable designs accommodating these variations to support different OSS projects.
Both Mozilla and Python are recognized as successful OSS projects of two very different categories. Their differences present two alternative solutions for designing OSS peer review processes. The two communities can learn from each other, while other younger OSS communities (e.g., GitHub) can also adapt these practices to their own contexts. For example, “bootstrap” (https://github.com/twitter/bootstrap), a front-end framework for web development, mainly attracted software developers to participate in bug reporting and fixing, which is alike Python community. In the contrary, TextMate 2 (https://github.com/textmate/textmate), an end-user application, draws many users to report usability issues and bugs, which is much more similar to Mozilla Firefox community. As these projects evolve and grow, it is likely that they will encounter issues found in successful communities of similar nature, and they could adopt the lessons learned in the case studies of successful open source projects in order to scale. Our observation indicates the following opportunities to improve the design of current tools that support the peer review process.

**Providing Alternative Views of Contributions**

The different contribution styles in Mozilla and Python, especially reporting styles, represent two types of knowledge resources—requests and patches. They constitute an issue-based space (i.e., talking) and a change-based space (i.e., acting) respectively, similar to “issues” and “pull requests” at Github, even though Bugzilla and Roundup do not differentiate them.

To effectively leverage those knowledge resources as they increase to a large scale, alternative ways of organizing the knowledge for prioritization, evaluation, and integration become important. Given the enormous amount of bug reports submitted, it is more critical for Mozilla to keep developers aware of the ones with patches attached, assigning high priority to them. In contrast, this may not be sufficient for Python; other indicators regarding the importance
of the patches, such as how critical they are to upcoming release, will help further sort
contributions and focus efforts of integrating them. Other than technological support, triaging
may also entail different sets of experience and skills to evaluate the legitimacy of reported
bugs—the ability to link symptoms (i.e., talking) to code base for Mozilla and familiarity with
organizational agenda to assess issues beyond its technical quality for Python.

Contributions of patch reviews and instructions also constitute sources of knowledge,
creating opportunities for articulating design rationale and externalizing implicit norms. It is
challenging for Mozilla to make patch reviews traceable and easy to associate with specific code
changes, whereas for Python the more critical issue is how to facilitate the underlying reasons of
patch revisions to be articulated. Instructions regarding how to create a better bug report in
Mozilla and how to develop a better patch in Python enhance the visibility of community norms.
As common strategies to externalize and share norms, exemplars of practices and mentorship
programs need to tailor their content to focus on the different emphases of instructions.

Adding Acknowledgment for Bug Reporting

The contrastive motivations of reporting bugs—acquiring user support versus providing
help with peers—indicate that tailoring acknowledgment to address what reporters care most may
reinforce their motivations, encouraging their future contributions. Mozilla reporters are largely
focused on solving their own problems of software usage. Thus, highlighting the fact that
reporters’ software applications have returned to normal will enhance their experiences of
reporting an issue. In contrast, Python reporters attempt to benefit other members with their
findings and workarounds. Acknowledging such generosity by introducing mechanisms for others’
to provide positive feedback and make the impacts visible, for example, allowing “like” on a code
change and displaying the number of “like” with the change-maker’s profile, can enhance the feeling of social influences and sense of community.

**Balancing Review Coordination Efforts**

A critical choice to make when organizing OSS peer review is how to define responsibilities of core developers. For instance, with respect to patch review, how many reviewers should be authorized to approve patches? Mozilla and Python provide two extreme but opposing options. One is dividing the review responsibilities at very fine level, such as modules and sub-modules, and only allowing a very small group (e.g., 1 to 4) of developers to permit patches for a specific module as Mozilla did. Such division reduces the need of coordination and may increase the chance of receiving responses to a patch. However, too much reliance on a small group may also increase the risk of no progress when none from the group is available. Python represents the other option—everyone shares the responsibility, which increases the bandwidth of reviewer resources but may also cause redundancy or fertility of efforts. Depending on the volume of patches and availability of core developers, OSS communities can employ strategies that balance between these two options.

**Supporting Different Mental Models in Tool Use**

The different characteristics of tool uses in Mozilla and Python, articulation and appropriation respectively, reflect participants’ different mental models of evaluating software products. The very articulated design of Bugzilla can be partly attributed to the gap between end-users and developers in regard to how they perceive the composition of software applications. Automated approaches to capturing the context in which software failed to function and
generating bug reports accordingly may help bridge the gap. More importantly, designing languages that end-users can easily apply to reconstruct their difficult situations, such as an additional layer mocking the software interface for users to point at, may further engage their uses of reporting tools.

Aside from supporting end-users’ mental models, Python’s improvised ways of using its bug tracking systems suggest developers’ capacities and inclination to creating their own ontologies to maintain their awareness and coordinate workflow. Therefore, enabling flexibility for configuring bug reports, such as selectively importing their work items in their own workspaces, may mitigate developers’ resistance against documentation but still obtain some structure of bug reports. In addition, allowing self-generation of indices, like tags, for bug reports may also make the system more compatible with developers’ mental models, although tags have their own weaknesses.

Limitations and Future Work

Our current analysis is constrained by its case selection and data sources. We chose Mozilla and Python because they both have established peer review practices and differ at product types and patch review policies; however, more differences may be identified when comparing with more OSS projects. Our investigation was focused on bug reports archived by bug tracking systems, although the collaboration in the peer review process may involve the usage of other computer-mediated communication tools and the archived information may be incomplete (Aranda & Venolia, 2009). The first author’s role as a participant-observer in these two communities throughout data collection and analysis helped mitigate this gap so that we were able to clarify and validate our findings. Other OSS projects that heavily rely on mailing lists to perform peer review may suggest different characteristics of interactions. We are planning to
analyze more data sources to refine and verify our current findings, for instance, mailing lists, commit logs, design documents, and interviews with active members.
Chapter 7

Conclusion

Summary of the Research

This dissertation work investigated OSS peer review, the importance and distinctness of which Linus’s law was coined to emphasize. It seeks to understand how a large crowd collaborates on improving software products. Comparative case studies of two established, well-recognized but contrastive OSS communities, Mozilla and Python, unfold the characteristics of OSS peer review processes and indicate the opportunities of supporting these processes with socio-technical designs. OSS peer review processes are commonly constituted of four work activities, submission, identification, resolution, and evaluation, which share similarities with the steps of individual reviews, review meetings, rework and follow-up in traditional software review but differ in the ways each activity is organized. An initial quantification of bug report archival shows “many eyeballs” contributed to the volume of reporting, which partly confirms Linus’s law. However, the effectiveness of finding real defects/deficiencies and fixing them is contingent on the community: compared to Python, Mozilla appeared to face greater challenges that a remarkable amount of reports were ignored, ineligible for a fix, or unsolved. Focused examination of the Mozilla case suggests that participation of the large crowd in peer review created challenges of increased workload and communication as well as frustration and conflicts, but meanwhile it enhances problem characterization, design review, and boundary spanning in the review process. The different review performance of Mozilla and Python also indicates the need of understanding the variations of OSS peer review processes so as to provide context-appropriate design recommendations. Subsequent comparative analyses on the Mozilla and
Python’s peer review processes identified their differences with respect to *contribution styles, reporting motivations, decision-making processes, division of labor,* and *uses of tools.* These contrasts may lie in the distinct product types, community scale, member diversity, community culture, and power and roles.

Drawing on the empirical findings from two successful but different OSS communities, a group of design strategies are outlined to address the opportunities and challenges. Support for establishing common ground, externalizing social networks, enhancing awareness of resolution progress, and articulating design rationale may facilitate the common activities of OSS peer review processes, and even foster creative collaboration among community members. In the meanwhile, the differences between Mozilla and Python present two alternative solutions for designing OSS peer review processes. They can inform each other of their practices and designs, while other emerging OSS communities (e.g., GitHub) can also tailor these practices and designs to their own contexts. The adaption needs to consider providing alternative views of contributions, adding acknowledgment for bug reporting, balancing review coordination efforts, and supporting different mental models in tool use.

**Contributions of the Research**

Although this research is primarily focused on one of the various work practices OSS communities perform (i.e., peer review) and its findings derived from a relatively small sample of cases (i.e., Mozilla and Python), it tackles the larger scope of community dynamics than the production activities themselves and places the two projects in the background of OSS evolution. Beyond codification of the collaborative peer review processes in Mozilla and Python and design guidelines for supporting the effective operation of such processes, this work contributes to the
OSS research community, the software engineering research field, and the research investment on virtual organizations in the following ways.

This research examined the under investigated phenomenon, Linus’s law, to enhance the understanding of OSS development. Besides loose coupling and diverse perspectives, as Raymond explained his proposition with self-experiences (E. S. Raymond, 2001), studies presented in this dissertation painted a more complicated and complete picture. Although finding problems is relatively independent among community members, validating the reported problems and fixing the valid ones rely on visibility of the reports, commitment and competence of reporters, coordination among developers, and even value negotiation among different stakeholders. Nuances of these issues can threat the validity of Linus’s law, which was demonstrated by Mozilla’s practices to some extent. Such deviation from the original ideology of OSS reflects the evolution of the paradigm of OSS development: the dual-identity of community members—users themselves are also developers (von Hippel, 2001)—breaks up when the software is created for users who do not share the same skill set with the developers, such as end-user applications; the hacker culture started to incorporate other cultures, such as the ones emphasizing user experience. Despite the awareness of the evolution of the social systems, the technological infrastructures have not reflected these changes, especially lacking support for user participation.

The codification of OSS peer review shows its distinctive characteristics compared to other software review methods, but design recommendations articulated based on this codification are not limited to supporting OSS peer review. One of the main differences between OSS peer review and traditional software review lies in the roles involved and their associated power. Reporters in OSS peer review perform the role of reviewers in traditional software review; however, they are not empowered in any decision-making as traditional software review does. Their merit should not be defined in terms of technological proficiency in the domain most
developers work on, but rather the type of contributions they attempt to make. In this context, it can be gauged and externalized as the competence to find real bugs, submit quality reports, and even as specific as experience with similar technical environment (e.g., operating systems, devices) or familiarity with similar designs (e.g., interaction patterns). Current software review practices increasingly engage experts from various domains, such as user experience designers and marketing analysts, in reviewing software products. The design proposals articulated above may facilitate their communication with developers in the same team as well as motivate their further contributions.

In addition to enhance the understanding of OSS development and inform other software review approaches, the peer review practices of Mozilla and Python also present two design patterns for designing effective virtual organizations. As we outlined in (Carroll & Wang, 2011), problem solving is a touchstone aspect of the effectiveness of virtual organizations. Peer review undoubtedly falls into this category. Mozilla and Python, well-recognized examples of virtual organizations, illustrate how to manage work quality and performance in organizations where members collaborate on production in contrast to organizations where members primarily pursue their personal development and collaborate on knowledge sharing (e.g., virtual communities of educators, online health communities). These patterns of practices also indicate opportunities for other organizational practices to connect with. For instance, we found users also reported bugs, requested features, and complained about software performance by interacting with organizational microblogs (Jing Wang & Carroll, 2013). Interfacing with these social media may enhance the problem solving performance and community building.
Future Research

Aside from addressing the methodological limitations as discussed in previous chapters, future research will investigate the evolved models of OSS development that are supported by the relatively new computing infrastructure (e.g., GitHub). The social networking infrastructure enables developers to maintain the visibility of their merit in their social profiles when crossing project boundaries. Despite the evident technological changes, how they may have affected peer review practices needs additional investigation.

Beyond focusing on OSS communities, I will continue to explore the larger space of online communities/virtual organizations. OSS communities present a good example of peers collectively producing an artifact, while many other online communities do not have such a collective goal that affords focused contributions, member cohesion, and sharing individual work. Therefore, those communities are likely to face different challenges. Understanding collaboration and social interactions in those communities will complement the current investigation on OSS communities. Part of the ongoing efforts are dedicated to studying how online communities supports everyday health issues with advanced persuasive technologies (J Wang & Carroll, in review).
References


Bos, N., Judith S. Olson, & Olson, G. Science on the net.


Ghosh, R., & Prakash, V. V. (2001). The Orbiten Free Software Survey


---

**References**


---

**Note:** The references have been formatted according to a standard citation style, which is not specified in the image. The text is intended to be a natural reading of the document content, without hallucinations or extraneous information.


Raymond, E. S. (2001). *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary:* O'Reilly.


Szczepanska, A., Bergquist, M., & Ljungberg, J. High Noon at OS Corral.


Appendix A

Interfaces of Bug Tracking Systems

Figure A-1: An Example of Bug Reports in Mozilla.
Figure A-2: An Example of Bug Reports in Python.
Appendix B

Codes from the Analysis on Mozilla

Theme 1: Increased Workload and Communication (*frequency*=155)
- Code 1: a large number of duplicate and invalid bug reports
- Code 2: teaching how to report bugs and debug
- Code 3: explaining organizational practices and norms
- Code 4: redirecting patch review requests
- Code 5: consulting specific developers

Theme 2: Frustration and Conflicts (*frequency*=97)
- Code 1: frustrated users’ complaints
- Code 2: value statements from developers and users

Theme 3: Problem Characterization (*frequency*=75)
- Code 1: complementing information in bug reports

Theme 4: Design Review (*frequency*=27)
- Code 1: articulating use scenarios
- Code 2: disagreeing with design

Theme 5: Boundary Spanning (*frequency*=56)
- Code 1: citing other sources to prove bug importance
- Code 2: third-party participation
- Code 3: comparing with competitors’ design
Appendix C

Codes from Comparative Analyses between Mozilla and Python

Theme 1: Contributing by Talking vs. Acting (*frequency*=91)
- Code 1: reporters reporting with patches
- Code 2: reviewers modifying patches
- Code 3: teaching how to report bugs vs. how to submit patches
- Code 4: suggesting submitting patches

Theme 2: Reporters Pulling vs. Pushing Help (*frequency*=97)
- Code 1: bugs for user support issues
- Code 2: value statements from developers and users

Theme 3: Individual vs. Collective Decision-making (*frequency*=109)
- Code 1: controversial issues decided by the community
- Code 2: core developers compromising
- Code 3: explaining how to propose and determine significant changes

Theme 4: Designated vs. Voluntary Responsibilities (*frequency*=3)
- Code 1: criticizing the undefined responsibility in Python
- Code 2: developers feeling responsible to repair the bugs they generated
- Code 3: developers explaining the volunteer nature of their work

Theme 5: Articulation vs. Appropriation of Tool Use (*frequency*=9)
- Code 1: assignee indicating patch author vs. versatile roles
- Code 2: tools supporting awareness
VITA

Jing Wang

Education

Ph.D. Candidate of Human-Computer Interaction, The Pennsylvania State University
2007-2013, University Park, PA

Bachelor of Management Information Systems, Renmin University of China
2002-2006, Beijing, China

Work Experiences

User Experience Professional (Intern), IBM Corporation
Jun 2013-Aug 2013, IBM Digital Design Lab, New York City, NY
Project 1: PatternLab and Design Innovation
Project 2: IBM Watson for Healthcare, a Visual and Interactive Diagnostic and Learning Tool for Healthcare

Research Assistant, The Pennsylvania State University
Jan 2008-Jun 2013, Center for Human-Computer Interaction, University Park, PA
Project 1: Designing for Everyday Weight Management Support
Project 2: Designing Effectiveness in Virtual Organizations
Project 3: Wireless State College, a mobile application that supports local community civic activities
Project 4: Investigating Requirements for Supporting Scientific Creativity in Collaboratories

Teaching Assistant (Lab Instructor), The Pennsylvania State University
Fall 2007 & Fall 2009, College of Information Sciences and Technology, University Park, PA
Course: IST220 Networking & Telecommunications