Information and Software Technology 67 (2015) 1–12

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Comparative case studies of open source software peer review practices

Jing Wang [a,*], Patrick C. Shih [b], Yu Wu [a], John M. Carroll [a]

[a] College of Information Sciences and Technology, The Pennsylvania State University, University Park, PA 16802, USA
[b] Department of Information and Library Science, Indiana University, Bloomington, IN, USA

ARTICLE INFO

Article history:
Received 24 November 2014
Received in revised form 5 May 2015
Accepted 10 June 2015
Available online 17 June 2015

Keywords:
Open source software
Virtual community
Software peer review
Design

ABSTRACT

Context: The power of open source software peer review lies in the involvement of virtual communities, especially users who typically do not have a formal role in the development process. As communities grow to a certain extent, how to organize and support the peer review process becomes increasingly challenging. A universal solution is likely to fail for communities with varying characteristics.
Objective: This paper investigates differences of peer review practices across different open source software communities, especially the ones engage distinct types of users, in order to offer contextualized guidance for developing open source software projects.
Method: Comparative case studies were conducted in two well-established large open source communities, Mozilla and Python, which engage extremely different types of users. Bug reports from their bug tracking systems were examined primarily, complemented by secondary sources such as meeting notes, blog posts, messages from mailing lists, and online documentations.
Results: The two communities differ in the key activities of peer review processes, including different characteristics with respect to bug reporting, design decision making, to patch development and review. Their variances also involve the designs of supporting technology. The results highlight the emerging role of triagers, who bridge the core and peripheral contributors and facilitate the peer review process. The two communities demonstrate alternative designs of open source software peer review and their trade-offs were discussed.
Conclusion: It is concluded that contextualized designs of social and technological solutions to open source software peer review practices are important. The two cases can serve as learning resources for open source software projects, or other types of large software projects in general, to cope with challenges of leveraging enormous contributions and coordinating core developers. It is also important to improve support for triagers, who have not received much research effort yet.

1. Introduction

A distinct and powerful characteristic of open source software (OSS) development is the involvement of communities that engage general users who do not belong to typical software development roles. This has made OSS development an appealing research area [2,24,42,43]. Recent advances of social computing infrastructure create opportunities for OSS projects to leverage an even larger crowd [12], as is demonstrated by GitHub's surpassing other open source forges in total number of commits in 2011 [15].

Among various forms of participation in a community, peer review is widely regarded as the one that significantly benefits from the community involvement. Raymond coined "Linus's law" ("given enough eyeballs, all bugs are shallow") to emphasize the advantage of extensive peer review in OSS development [31]. It is the practice in which 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.

This community-based practice raises the question of how communities organize their peer review process, especially when they have reached a large scale. As OSS communities grow and mature, they encounter new challenges of engaging contributions, coordinating work, and ensuring software quality [27,38]. Without explicit coordination mechanisms and governance, their sustainability

* Corresponding author at: 316C IST Building, University Park, PA 16802 USA. Tel.: +1 814 863 8856.
E-mail addresses: jzw143@ist.psu.edu (J. Wang), patshih@indiana.edu (P.C. Shih), yuw132@ist.psu.edu (Y. Wu), jcarroll@ist.psu.edu (J.M. Carroll).

http://dx.doi.org/10.1016/j.infsof.2015.06.002
0950-5849/

and evolution will be challenged [17,26]. Therefore, reflecting on the practices of well-established large OSS communities could prepare other growing OSS projects of how to overcome such challenges.

Each project has its own uniqueness and no one-size-fits-all model can ensure success [4]. Studies on activities related to OSS peer review (e.g., [21,28,34]) have already showed initial evidence that the peer review practice is likely to vary across different OSS communities. However, those analyses did not spend much effort articulating the variances. They were largely focused on extracting commonalities among OSS projects or characterizing a single project, reporting the common constituting activities of OSS peer review including submission/bug reporting, analysis/design discussion, fix/patch development, test/patch review [10,45].

Our investigation is aimed at extending prior research with a dedicated codification of different OSS peer review practices, which can provide contextualized implications for other developing OSS projects. We use the term "peer review" in a broad sense to include all types of efforts in detecting software defects or deficiencies rather than confine it to code reading. As a distinctive and pivotal characteristic of OSS peer review, user involvement undoubtedly contributes to the differences among the practices. Users are also a substantial participating group of OSS communities, from which peer review practices cannot be detached. Considering the unique quality and the context complexity, we contrast OSS peer review practices through case studies on two well-established large communities that produce software targeting remarkably different types of users—*software developers* and *end-users*. Two communities can by no means cover all the variations of OSS peer review practices, but can still provide important insights [4]. Moreover, comparing the two ends of a spectrum of user technical expertise highlights the differences of significance as well as enables an appropriate and flexible combination in-between.

Our study contributes to alternative designs of social mechanism and technology for OSS peer review. We identify differences in *bug reporting*, *design decision*, and *patch development and review*, the key activities constituting peer review practices, as well as in the tool affordance and use between two well-established large OSS communities. Our findings highlight the importance of *triagers*, an emerging group of contributors who mediate between the core and periphery and facilitate the peer review process. We also characterize how core developers collaborate differently in response to the different types and sizes of peripheral participants. This extends prior research that primarily focused on comparing between core and periphery.

2. Related work

2.1. Community-based open source software development

The openness of OSS to users and the engagement of virtual communities have been attracting considerable research efforts. Findings and discussions have centered on the different contributions from core and peripheral members. Quantitative examinations repeatedly detected skewed contribution distribution: a small group of developers in the core contributed majority of the code, while the rest in the community mainly made occasional contributions by reporting bugs [18,25,28]. Subsequent work elaborated the roles of core and peripheral members in OSS development. Dahlander and Frederiksen [13] studied how peripheral participants affect OSS innovation. They found that one's position in the core/periphery structure is more consequential for innovating than his/her expertise. Rulliani and Haefliger [36] described the role of core developers and those in the peripheries, and how the propagation of such standards is communicated through non-material artifacts such as code and virtual discussions as a social practice. Another related theme focused on how peripheral participants advanced to core developers through either legitimate peripheral participation [29] or socialization [14,44].

Some OSS studies described the roles in OSS development at a finer level than the dichotomy, but most of them still classified through the lens of users versus developers. The best-known examples from this group of work probably include the "onion model" [8] and the layered community structure [52]. They share similar ideas: developers other than the core contribute their patches or review or revise others' code, either regularly (i.e., active developer/co-developer) or sporadically (i.e., peripheral developer); users consist of active ones who report bugs and passive ones who only use the software. Ko and Chilana [24] also identified a group of users, *power users*, who submit quality bug reports. A very small portion of research discussed the roles between users and developers. Barcellini et al. [3] characterized the emerging roles in the Python community. They studied 2 mailing lists (i.e., one user-oriented and one developer-oriented) and found that several key participants act as boundary spanners across the communities for driving feature sets. They proposed a "role emerging design" to support the coordination process in OSS.

Our study extends the understanding of roles in OSS communities through identifying and elaborating the emerging role of bug triagers. While the OSS peer review process is not the same as feature discussions reported in the aforementioned OSS literature, we found that bug triagers serve a similar boundary-spanning role for resolving issues surrounding bugs. This role entails different types of tasks depending on the characteristics of users and developers in the community.

2.2. Software peer review in open source

Peer review is the evaluation of the quality of one's work products by others. In software development, 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 [48]. Code is not the only object that peer review assesses but any artifacts created during the software development process, such as requirements specifications, use cases, project plans, user interface design and prototypes, and documentation [22].

To this end, we use the term OSS peer review inclusively to refer to the evaluation of the entire software application instead of confining it to reading code or reviewing patches (i.e., change sets to software products). We consider that such inclusiveness best suits the original description of "extensive peer review" in OSS literature [31]. Previous studies have identified common activities in the peer review process [10,45]. It often begins with one submitting a bug report (i.e., submission/bug reporting). Others diagnose the defect causes and request additional information to determine whether the bug should be fixed (i.e., analysis/problem identification/design decision). Once a solution is generated (i.e., fix/solution generation/patch development), they evaluate then commit the solution to the current software product (i.e., test/patch review and commit).

Previous OSS studies have touched upon each individual activity involved in the OSS peer review process, but were largely focused on the common characteristics across projects. With respect to bug reporting, researchers found a mismatch between the information users reported and the information developers needed [6,45]. Ko and Chilana [24] examined the massive bug reports in Mozilla, suggesting that reports that led to changes were reported by a comparably small group of experienced frequent reporters, who were the only valuable users to recruit in open

bug reporting. Regarding design decision-making, scholars observed lack of usability discussions. They argued that end-users and usability experts could provide special expertise and knowledge the other developers do not have [2,42,43]. Greater participation in decision-making was also found to be associated with more effective projects [21]. Sandusky et al.'s analysis of Mozilla's bug reports also uncovered the different values and perspectives of individuals involved in determining bug legitimacy and best design option [37]. Work on patch development or bug fixing was mainly focused on the coordination mechanisms, which appeared to be biased toward action [50] and lacked formal assignment [10].

Patch review, another critical activity in OSS peer review, received relatively less research effort. Rigby et al. characterized OSS patch review processes as two types, review-then-commit (RTC) and commit-then-review (CTR) [33]. They studied several OSS projects that all relied on mailing lists for patch review. They found that roles were not assigned during the review process. Broadcast-based review was carried out effectively when mailing lists had separate topics or review requests specified reviewers as Linux did [34]. More recently, Yang et al. [51] and Hamasaki et al. [20] investigated the importance of OSS patch review contributor roles and their review activities from the Gerrit Code Review System (Gerrit) used in the Android Open Source Project (AOSP) by using social network analysis. Their analysis identified roles across three phases of the code review process: (1) pre-review: author and committer; (2) review: code reviewer, verifier, and approver; (3) post-review: submitter. Mukadam et al. [30] also analyzed the code review data from Gerrit used in AOSP and found similar division of labor exists. However, these recent studies predominantly analyzed quantitative data provided in the code repository (e.g., identifying activity trends based on timestamp, roles) without correlating their findings using qualitative content analysis. Thus, it is unclear how those roles were carried out and interacted with each other.

A limited number of studies on peer review related activities examined another interesting issue, the role of triagers. Xie et al. [49] quantified the types of contributions triagers made to Mozilla and Gnome, including filtering issues, filling missing information, and classifying issues into their affecting products. Among these contributions, non-developer triagers performed relatively poor in accurately determining relevant product. They also suggested that the issue attributes triagers modified differed between Mozilla and Gnome, which they attributed to the differences of user bases and community policies. Specifically, they found that triagers modify Product, OS, Version and Severity in a larger fraction of triaged issues for Mozilla and Priority in a larger fraction of triaged issues for Gnome.

In line with Xie et al.'s [49] motivation, we study very different OSS communities, seeking to characterize how each activity of the peer review process was carried out differently. Our in-depth comparison between projects that targeted at extremely different types of users and were at different scale extends existing research that was primarily focused on either commonalities or an individual case. This comparison is by no means intended to judge which community is better; instead, our goal is to understand how different OSS communities can better organize their work through such a comparative analysis.

2.3. 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. Several research papers have described the basic functionalities of those tools (e.g., [32,35,40]).

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 systems also serve as a focal point for communication and coordination for many stakeholders (e.g., customers, project managers, quality assurance personnel within and beyond software development team) [5]. Bugzilla is employed by many large OSS communities, which include Mozilla, GNOME, Netbeans, and Apache [19]. 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.

Improving the design of bug tracking systems has engaged ongoing effort. Despite some incremental improvements, many of the designs remain merely better interfaces to a database that stores all reported bugs [23]. Based on their examination of bug reports from Mozilla and Eclipse, Breu et al. [6] 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. 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 [12] use contributors' reputation on past work and social profile as a preliminary way to infer code quality. Their designs have largely overlooked the different types of users and distinct community contexts.

To enhance the understanding of how technologies can better support OSS peer review processes, our study extends prior research by presenting two design alternatives that were tailored to different peer review processes, which took place in different community contexts with different types of users. They provide a starting point to break the "one-size-fits-all" way of designing bug tracking systems.

3. Methods

Our overarching research objective was to unfold and contrast different OSS peer review practices. As previous studies indicate (e.g., Crowston and Scozzi [10]; Wang and Carroll, [45]), the OSS peer review practice consists of four common activities despite their naming variations. They include submission/bug reporting, analysis/design discussion, fix/patch development, and test/patch review. Our study, therefore, specifically examines the following research questions:

(1) How do different OSS communities submit/report bugs?
(2) How do different OSS communities analyze reported software defects and design issues?
(3) How do different OSS communities (develop a patch to) fix a bug?
(4) How do different OSS communities review/test a patch?

3.1. 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 [6,50,53]. 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 [11]. We intentionally selected two established and large communities for the purpose of codifying the good models of OSS peer review practices. Such codification was anticipated to serve as resources that could inspire other types of OSS communities/systems, like new and unstable projects and smaller projects. Analyses have shown that small OSS projects only have one or two developers and the majority of new projects failed because of their inability to attract a critical mass of stable developers [7]. For projects at that scale, peer review can remain effective even without explicit coordination mechanisms and governance, like what Crowston and Scozzi [10] found after analyzing bug fixing activities in small projects hosted at SourceForge. However, those projects will need control and structure when they have grown to a larger scale [26]. Peer review in large projects, like Mozilla and Python, illustrates how the control and structure are implemented.

An important reason we compare Mozilla and Python is the different users they target, who account for a significant portion of OSS communities. The type of software being developed is a relatively evident feature that can differentiate OSS projects. As a commonly used case sampling criterion (e.g., [10,34]), it has been suggested for observing differences of OSS governance [26] and found to be associated with member diversity influencing OSS peer review processes [46], both of which are crucial to identify the differences of our interest. Mozilla produces 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. 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.

Our deliberate selection of Mozilla and Python also took their different community sizes into consideration. Mozilla is significantly larger compared to Python. For example, Mozilla attracted significantly more commits (~60,000), approximately 10 times, compared to Python and Mozilla had more than 1200 active contributors (compared to around 60 in Python) during 2014. The significant size difference between the two large communities is likely to pose distinct challenges to Mozilla and Python, entailing different ways of organizing peer review.

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. Python has 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, Mozilla describes their work in ECMAScript proposals, W3C/WhatWG working group recommendations, and Request For Comments (RFCs). Similarly, Python creates Python Enhancement Proposals (PEPs) for extensions of the language, such as the introduction of a new feature (e.g. decorators) or changes to the syntax.

3.2. 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. Data in our study were not generated for our research purposes, like responses to pre-defined tasks or investigators' inquiries, but rather the records of public activities and conversations within OSS communities and were collected in an unobtrusive way as part of people's natural work practices. Such type of data has been widely used in OSS research (e.g., [10,28]). More importantly, the goal of our study was to investigate work practices, which involve multiple players and potentially last for a long time. The archival of people's activities and conversations is not confined by time frame and does not rely on participants to articulate the routine aspects or recall old episodes. The openness and reliance on distributed collaboration of OSS further alleviates the common concern that too much of data is private and not accessible. In this sense, the public bug reports may have better validity.

In addition to the focus 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. We relied primarily on artifacts that record actual behaviors, namely bug reports and discussion threads at mailing lists, to draw inferences presented in the paper. The web documents in which the organizations describes their work flow was leveraged only for understanding the study context as well as clarifying terms used in bug reports. Throughout the duration of data collection and analysis, the first author also actively participated in developer and community meetings and IRC discussions in these communities. The participation helped clarify and validate concepts whenever issues arose during the data analysis.

The selection of the data we collected was driven by our research goal—investigating how open source software communities carry out peer review practices. These work practices are rooted in complex community contexts, involve dynamic groups of people who may represent diverse stakeholders, and are likely to extend over a long time period. Such characteristics entail data that are rich enough for in-depth analysis and have captured people's natural work activities. To this end, we selected the archival of public activities and conversations related to peer review within OSS communities as our major data source. This type of data is

not confined by time frame or participants' inability to articulate the routine aspects or recall old episodes. The openness and reliance on online collaboration of OSS further alleviates the common concern that most data are private and not accessible unless they are directly elicited for specific research purposes. Well-recognized OSS studies have also frequently used this type of data (e.g., [10,28]). Furthermore, we triangulated our findings with other data sources through conducting virtual observation on project meetings and informal interviews with community members to enhance the validity.

We retrieved bug reports created between two stable releases of Firefox and Python for analysis. The retrieval was performed on July 28, 2011. This sampling strategy was intended to capture the possible behavioral changes near releases [16]. For Mozilla, 7322 reports were filed between the Firefox 3.5 first official release and the Firefox 3.6 first official release. For Python, 1850 issues were generated between the Python 3.0 official release and the Python 3.1 official release.

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 [9], 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. Oversampling long discussions might overestimate the collaboration level. However, we found that collaboration is prevalent in the projects, whether or not as intense as the long discussions demonstrate. On average, 2 Mozilla members participated in discussing one bug (*median* = 2; *skewness* = *6.765*), contributing 3 comments in total (*median* = 3; *skewness* = *14.305*); 3 Python members contributed to each issue discussion (*median* = 3; *skewness* = *3.007*), generating 4 messages (*median* = 4; *skewness* = *5.669*). 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 followed the grounded theory approach to discover patterns and recurring themes [41]. Previous studies have identified the key activities the OSS peer review process consists of [10,45]. We used this understanding to guide our coding process. Moreover, within each of these activities we used open coding, because there was no existing schema. Although quantitative approaches are advantageous at assessing performance/outcomes or analyzing relationships (especially causal ones) between quantifiable variables, our focus was to understand how peer review was carried out and participants' interactions in consideration of the specific social context. We did not exclude any type of participants from our analysis, including the ones reported bugs, commented on bug reports, submitted code, and committed patches. Such inclusiveness serves our goal of unfolding what people do and how they do it in OSS peer review rather than why an individual behave in a certain way.

The iterative 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. An 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 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. Despite the sequential difference in the coding process that the Mozilla data were analyzed before the Python data, our analyses did not differ in terms of their thoroughness between the two data sets. As with the Mozilla dataset, we went through the entire Python data sample and coded the narrative units of the Python data sample in its entirety. In our analysis, we identified commonalities and differences between the two communities and presented them in the paper. We do not think the order of the coding process affected our findings; we would have arrived at the same conclusions if we had coded the Python dataset first.

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 of not modifying code held true in almost every bug report that had patches (n = 494). Then the authors integrated the codes with observations other than discourses into 4 themes, which are presented in the following section.

4. 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, particularly the contribution volume and the participation scale and distribution of the communities (see Table 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, for which 1850 issue reports were submitted by 829 distinct participants. 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). Moreover, the portion of core developers among all the contributors in the

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

	Mozilla	Python
Bugs reported	7322	1850
Distinct bug reporters	5419	829
Distinct bug contributors (reporters and commenters)	6704	1188
Core developers	176	127
(Ratio among all contributors)	(2.63%)	(10.69%)
Bug reported by core developers	779	487
(Ratio among bugs reported)	(10.64%)	(26.32%)
False positives (non-real bugs)	3418	469
(Ratio among bugs reported)	(46.68%)	(25.35%)
Fixed bugs	498	832

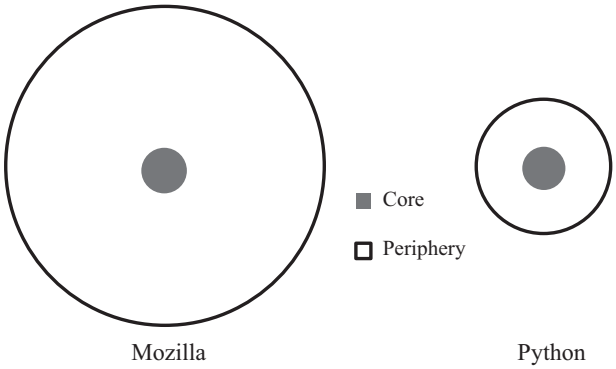

Fig. 1. Comparison of the number of contributors and the portion of core developers in Mozilla and Python (areas of the circles are proportional to the number of people).

Mozilla community (2.63%) was much smaller compared to in the Python community (10.69%) (also illustrated in Fig. 1).

Bugs filtered out, namely false positives, in Mozilla (n = 3418; 46.68%) were much 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 (n = 2680) constituted a significant portion of the Firefox open bugs (n = 3406; 78.68%).

Further quantitative analysis indicates that collaboration is crucial, to some extent, for the peer review process to accomplish its goal in the two communities. 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 ($p_{participant} < .001$, $p_{comment} < .001$) than in Python ($p_{participant} = .054$, $p_{comment} = .088$).

Our subsequent qualitative analysis further identified the varying ways Mozilla and Python communities carried out peer review. These differences encompass variances in each key activity that constitutes the peer review process, from *bug reporting*, *design decision making*, to *patch development and review*, across different community contexts. The variances also include the different *tool uses* involved in these computer-supported processes.

4.1. Bug reporting: Description vs. code

The different practices between the two communities emerged since the beginning of the peer review process, *bug reporting*. Although both descriptions and code are prevalent artifacts shared in Mozilla and Python, participants in Mozilla seemingly tend to report bugs by describing experiences, whereas those in Python often through implementing code (e.g., patches, test cases). Regardless of bug legitimacy, reporters of 276 Firefox bugs created patches for their own reports (3.77%; n = 7322). In contrast, reporters were also patch authors in 260 Python issues (14.05%; n = 1850). Moreover, 6.75% Firefox bug reports had patches (n = 7322), whereas the portion for Python was 40.32% (n = 1850).

The different styles of bug reporting are further demonstrated when reporters articulate their intents for creating reports. 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 doesnt 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)*

In response to the different forms of contributions from the large diverse communities, experienced members appear to share norms and organizational practices with different emphases to help inexperienced participants make quality contributions. Mozilla contributors often taught how to describe problems, such as what types of information reporters 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 often related to coding, 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)*

To leverage enormous incoming contributions, experienced participants also spend efforts triaging bug reports other than coaching peripheral contributors to help peer review move forward.

Triaging involves categorizing and prioritizing bug reports. Confirming whether a reported problem is a real bug accounted for a significant part of triaging efforts in Mozilla, given many reports were false positives (n = 3418; 46.68%). A dedicated group of active contributors performed this task repeatedly, asking reporters to obtain a stacktrace for crashes, check nightly build (the newest code base), or test with safe mode and a clean new profile as shown below.

> "*Try http://support.mozilla.com/en-US/kb/Safe+Mode and a new additional test profile http://support.mozilla.com/en-US/kb/Managing+Profiles*" *(Bug 507708; comment 1; commenter/active volunteer).*

Mozilla had not defined triagers as a formal role in the community, although their work was common in the peer review process. The organization had a quality assurance (QA) team that took on some of those triaging responsibilities. However, most of them relied on volunteers. 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 to Mozilla, the triaging role in Python was defined even more informally, focusing on drawing developers' attention to patched reports rather than confirming symptoms. Beside core developers who acted as triagers spontaneously, only a single individual consistently played this role in our data sample. Five other active contributors who were not core developers participated more like co-developers in implementing code. Triagers were not explicitly recognized as in Mozilla, but rather just granted advanced privileges to the bug tracking systems. Major efforts of triaging were dedicated to adding flags (e.g., affected Python versions) to bug reports and highlighting the ones that already had patches but lacked developers' participation.

> *"Patch is small and simple, can we move this forward?" (Issue 5729; comment 1; commenter/triager; active contributor; Python)*

Aside from those social mechanisms to harness the community contributions of bug reporting, the designs of bug tracking systems are also tailored to address the differences in bug reporting. The most evident different design was the flags the two communities used to categorize open bugs. While Roundup simply labeled the status of all unresolved issues as *open*, Bugzilla divided open bugs into 4 types, *unconfirmed*, *new*, *assigned*, and *reopened*. The status of *unconfirmed* provides triagers with easy access to the group of bug reports that need their care. Python developers can use the keyword *needs review* to highlight the readiness of a patch.

Summary: The different peer review practices begin from bug reporting activities, in which Mozilla reporters mostly describe symptoms whereas a significant number of Python reporters implement code solutions. To help reporters submit quality reports, Mozilla contributors often suggest the types of information that is critical to bug description, while Python contributors offer guidelines for coding. Triaging and its supporting technical infrastructure also have different focuses in response to the different characteristics of bug reports. Mozilla emphasizes confirmation of a bug, whereas Python examines readiness of a patch.

4.2. Design decision: Individual vs. collective decision-making

Another remarkable difference between Mozilla and Python arises when they have to make decisions on design-related issues, either the appropriateness of requested features for project agenda or the effectiveness of proposed approaches for patch design. Although Python has a BDFL who gives a final say over technical disputes, he only participated in a small number of discussions, 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 a common norm (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 for controversial issues. 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" [47], 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. Developers often prefer to take such long discussions and voting outside the bug tracking system, such as at mailing lists. In one Python bug report regarding whether to maintain both the original Python implementation of the IO file and a new C version, two developer commenters could not reach an agreement. They directed the issue onto the Python-dev 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)*
> *". . . + 1 from me for keeping the pure Python version around for the benefit of other VMs as well as a reference implementation." (a message from the email thread with the subject "[Python-Dev] IO implementation: in C and Python?"; core developer).*
> *"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)

On the contrary to Mozilla, Python developers attempt to avoid individual influences, especially when the underlying reasons are not well articulated. The example below shows that after a core developer closed an issue report and set it as rejected, other developers criticized such acts and eventually had the issue reopened and reconsidered.

"what do you mean by 'too trivial' ?
I don't understand why this is now suddenly rejected. Raymond, Guido, and other people have + 1 this on python-ideas.
http://mail.python.org/pipermail/python-ideas/2009-May/004871.html
People have worked on a patch, so I think this is unfair to close it now without more explanations, and just say that 'we agreed at EP'..." (Issue 6095; comment 10; commenter; core developer; Python)

Voting does not happen every time disagreement arises in Python. 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)

The technical designs for decision making also vary between Mozilla and Python. Although Mozilla values individual decisions, Bugzilla has implemented a field in individual bug reports that allows people to add their votes. The total number of votes a bug report receives is defined as a bug's importance. Anyone can create a Bugzilla account and then can vote if s/he thinks the bug should be fixed. Despite the affordance of weighing in on deciding a bug's legitimacy, the module owner still can ignore the votes if s/he has different opinions. For example, an end-user perceived himself powered to influence the decision on a feature. However, the core developers considered this misinterpretation of the affordance.

"Bugzilla works with an element of voting on the issue by it's members, correct? Is that not a democratic element to software development?" (Bug 513147; comment 78; commenter; peripheral participant; Mozilla)
"Nope, there's no "voting" involved at all. Mozilla is a meritocracy, and as such, the module owner/peers have final say over decisions concerning the codebase... If you're referring to the "voting" feature supported in Bugzilla, votes for bugs are almost always ignored and mean pretty much nothing" (Bug 513147; comment 79; commenter; core developer; Mozilla).

Oppositely, Python does not provide any affordances for its voting process despite the role of voting in facilitating its decision-making. Participants embed their votes (i.e., +1, 0, −1) in narrative messages at either the bug tracking system or mailing lists, as earlier quotes of this subsection shows. Additionally, votes are not explicitly calculated but only estimated.

Summary: The peer review practices continue to differ when communities have to decide bug legitimacy and patch design approaches. Mozilla empowers module owners and peers to make the decision when controversy arises, while Python relies on the collective decision from all its core developers through informal voting. With respect to technical support, Bugzilla enables every user to vote but the voting results do not have much impact on the final decision. In contrast, neither Roundup nor Python mailing lists provide any explicit voting features.

4.3. Patch development and review: Designated vs. voluntary responsibilities

The variances of division of labor between Mozilla and Python emerge during the collaboration on developing and reviewing patches. Specifically, the two communities differ in two aspects: one is how they divide responsibilities among patch reviewers; and the other is how participants play the roles of the patch author and the patch reviewer for a bug. As the complexity and the variety of work activities in large-scale OSS communities increase, the responsibilities are defined in an increasingly articulated way. For instance, both Mozilla and Python have non-profit organizations, in which their members fill various positions supporting 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. Mozilla patch authors have to explicitly request the designated reviewers (usually two or three for an individual module) for assessment (i.e., the affected module's owner or peers) in order to integrate their patches into the code repository. 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. Reviewers use "r = " in their comments and set the flag "review + " along with the uploaded patch to indicate that a patch passes review as the following quote illustrates.

"reviewed on IRC and manually tested working, r = me." (Bug 515463; comment 14; patch reviewer; core developer; Mozilla)

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. Patch authors in Python rarely 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 toward 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 difficultly 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)

Another difference lies in the boundary between the role of patch author and that of patch reviewer, even though neither the two communities enforce such boundaries. 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 directly modified the patches, turning themselves into patch authors. They left summaries as comments in the bug reports when the changes were minor. Other times when they did not agree with the design, they created a new patch to implement their own approach. Among all the bugs that went through patch review and eventually got fixed or accepted, 31.70% were the cases in which patch reviewers became patch authors ($n = 388$). Such cases only occurred 8 times in Mozilla. 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 communities both recognize patch authors' contributions but acknowledge in slightly different ways. Moreover, patch reviewers can be easily traced in Mozilla but much less so in Python. The version control system and the bug tracking system in Mozilla both record the name/user ID of patch authors and patch reviewers as metadata for each bug. The bug tracking system in Python functions similarly in regard to archiving patch authors' identities. However, given Python developers often have to commit other contributors' patches to the version control system and only the committers' names/user ID will be automatically captured, they create additional documentations in the code base to acknowledge the original authorship. Patch reviewers are not explicitly specified in Python. Thus, their identities are not annotated in either the bug tracking system or the version control system.

Summary: In the activities of patch development and patch review, core developers in Mozilla tend to not switch to the role of patch authors when they review others' patches, whereas Python reviewers directly revise patches more frequently. The responsibilities of patch review are specified and designated to module owners and peers in Mozilla, while any core developers can volunteer reviewing and approving patches. Additionally, Mozilla captures the identities of patch authors and reviewers as metadata in its bug tracking system and version control system. Python acknowledges patch authors through documentation, while has not provided a deliberate way to track patch reviewers.

4.4. Tool affordance and use: Articulation vs. minimalism

Aside from the different characteristics of the key activities constituting the peer review process, Mozilla and Python are also distinct from each other in terms of the affordances of their supporting technologies and the ways their contributors use these tools. The primary supporting technology is bug tracking systems for both projects, unlike many other small OSS projects that rely on emails to achieve this purpose. The designs of Bugzilla are more biased toward documentation and Mozilla contributors interact with it in a relatively articulate way. In comparison, Roundup features afford actions in the peer review process and Python developers use them in a very flexible fashion. We have described some of the differences in the earlier subsections.

The essential support bug tracking systems provide is similar in Mozilla and Python, including a summary of the key information about bugs schematically (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.

One remarkable difference in the interfaces of bug tracking systems is the *keywords* defined for a bug. The *keywords* field holds keywords pertaining to the categorization of a bug. Bugzilla maintains a very articulated list of keywords, whose length currently has exceeded 500. For instance, keywords related to documentation already include *dev-doc-complete*, *dev-doc-needed*, *user-doc-complete*, *user-doc-needed*, and *doc-bug-filed*. On the contrary, Python keeps a minimal list of keywords, which is as short as 6 keywords. *Easy* (i.e., fixing the issue should not take longer than a day for someone new to contributing to Python to solve), *needs review* (i.e., the patch attached to the issue is in need of a review), and *patch* (i.e., there is a patch attached to the issue) are the straightforward ones calling for specific contributions. Bug reports with these three keywords are also highlighted as search shortcuts in Roundup interface.

Another distinction associates with the flags indicating the progress of peer review. As we discussed earlier, Bugzilla classifies open bugs into 4 groups, *unconfirmed*, *new*, *assigned*, and *reopened* to differentiate the voluminous incoming bug reports. Roundup does not divide open issues; instead, it provides an extra field, *stage*, to engage core developers in taking the peer review process to the next phase. The flags for this field consist of *unit test needed*, *needs patch*, *patch review*, *commit review*, and *committed*/*rejected*.

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)*

Summary: As the primary technical infrastructure for peer review, bug tracking systems are designed and used in a very articulated way in Mozilla, particularly values of the fields in a bug report. In contrast, Python contributors appropriate the tracking system to attract actions from the community with minimal efforts.

5. Discussion

After reviewing publications on OSS from multiple fields in the past decade, Crowston et al. suggested "[f]uture research needs to compare projects ... of varying types in order to advance our understanding of FLOSS development" [11]. Our study is a step toward this goal. Reflections on the classic role models, projects that have established work practices, are well recognized, and keep evolving, provide learning opportunities for other growing communities. We examined the key activities of the OSS peer review process and their supporting technology designs in two OSS projects targeting entirely different types of users and of different sizes. Fig. 2 shows how our findings relate to each other.

Aside from the different types of software types and different sizes of communities, Mozilla and Python also adopt different types of software licenses. However, we did not observe any association between the differences of licenses and the variances of peer review practices. Gamalielsson et al.'s study on the sustainability of OSS projects and their forks indicated that a weak copyleft license was preferred to alternative permissive licenses by a sustainable fork's contributors [17]. Given this findings only derived from one project, whether one type of license has better impacts on projects' sustainability than another is not conclusive. For example, Schweik [39] did not identify any relationship between a project's success and its license type from examining a large number of OSS projects hosted at SourceForge. In our study, Mozilla and Python use their self-defined licenses rather than any existing ones. Mozilla's Mozilla Public License (MPL) is partially copyleft, while Python's Python Software Foundation License (PSFL) is permissive. We did not find that Mozilla's participants expressed or acted more likely to remain in the community or contribute because the license is more "open" in the way Gamalielsson and Lundell [17] implied in their article.

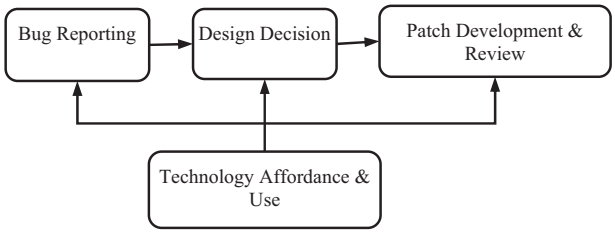

Fig. 2. Relationships between themes of the results.

The two projects serve as alternative designs for both organizing peer review and improving supporting tools. The two communities can learn from each other, while other younger OSS communities (e.g., projects at GitHub) can also adapt these practices to their own contexts. For example, "bootstrap" (https://github.com/twitter/bootstrap) is similar to the Python community in that it is a front-end framework for web development that mostly involves software developers in bug reporting and fixing. On the contrary, Mopidy (https://github.com/mopidy/mopidy), an end-user application, draws many users to report usability issues and bugs, and therefore, it is much more similar to the Mozilla community. As these projects evolve and grow, it is likely that they will encounter issues found in successful communities of similar nature. They could adopt the lessons learned in our case studies of successful open source projects in order to scale. Open source forges, such as GitHub and Sourceforge, may also provide configurable designs accommodating these variations to support different OSS projects. We discuss these implications in this section.

5.1. The emerging role of triagers

Bug reports in pure description or with code entail different ways to categorize and prioritize them in order to tap the massive incoming contributions. Mozilla and Python present two different approaches. In an end-user dominated community like Mozilla, confirming a report a real bug is probably the most critical task for triaging. This helps screening out a large portion of reports that describe problems evidently caused by other software or inappropriate system setting, or have already been reported, or lack information to diagnose. In addition to use highly articulated reporting schema to elicit information from reporters at bug submission, providing resources about how to verify a bug is also a viable solution. In a tech-savvy developer dominated community like Python, triaging should be more focused on suggesting the type of action in need. For example, reports can be classified into the ones that are easy to fix, need discussion, have patches missing test cases or documentation, or have patches ready for review. Such categorization helps direct the attention of core developers and active developers. This difference of triaging priority also complements the variance of issue attribute modification reported in Xie et al.'s study [49].

As our results indicate, a group of active contributors who frequently perform triaging emerged as the amount of triaging work became paramount. These triagers have some technical expertise and are capable of programming, but they rarely contribute code and are largely focused on sorting bug reports. This group of triagers is much larger and more visible in Mozilla, mediating the collaboration between end-users and core developers. In comparison, triaging in Python has not yet become a separate role in the community. Core developers and active developers usually carry out this task while they also frequently create or modify patches as well. This difference may be due to the relatively smaller volume of submitted reports compared to the group size of core developers. As reports from the periphery exceed the time resources developers can delegate to triaging, a specialized group of triagers probably will emerge in Python. Recognizing the contributions of triagers (e.g., Friends of the Tree in Mozilla), and even formalizing the triaging role to give them an identity, which have not been implemented in either the two communities and may suppress their motivations in the long term.

5.2. Decision efficiency and feedback gathering

Getting agreement on a design issue from a large community like Mozilla and Python is almost impossible. Thus, these two communities both grant their core developers to make the call. Although reaching a decision is important, gathering feedback

from argumentation among people with diverse perspectives and expertise is also beneficial. The two distinct rules and technical affordances for decision-making in Mozilla and Python result in different experiences of interacting with core developers. In Mozilla, decision can be made efficiently because it only depends on a very small number of members. Meanwhile, emotional conflicts are more likely to arise and constructive feedback may become difficult to evoke. The affordance of voting provided by the bug tracking system made these negative impacts even worse. In contrast, the emphasis on rationale in Python mitigates such conflicts. No direct affordance of vote counting may even reinforce this emphasis.

5.3. Control and spontaneity

How to define responsibilities of core developers is challenging for organizing patch review. 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. Mozilla divides the review responsibilities at a very fine level (e.g., that of modules and sub-modules), and only allows a very small group (e.g., 1–4) of developers to approve patches for a specific module. Such division reduces the need of coordination and increases 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. In the Python community, everyone shares the responsibility, which increases the bandwidth of reviewer resources but may also cause redundancy or futility of efforts. Depending on the volume of patches and availability of core developers, OSS communities can employ strategies that balance between these two options.

In accordance to the different degrees of control at approving patches, Mozilla and Python employed different technology designs to enhance awareness of patch reviewers. Mozilla enables patch authors to explicitly name specific reviewers through setting flags on their patches, Python does not provide such affordances, but rather its developers proactively look for patches to review. These designs left issues that may need further investigation. For instance, whether the request for patch reviewers should be further constrained to a pre-defined list of developers and how patch authors can identify the appropriate reviewer. For projects like Python in which control is more *ad hoc*, powerful search that helps developers find their interests would be a much more desired feature.

5.4. Documentation and facilitation

The different characteristics of tool affordance and use in Mozilla and Python reflect the differences in their peer review processes. The very articulated design of Bugzilla can be partly attributed to the gap between end-users and developers in regard to how they conceptualize software problems. Such articulation enhances the documentation of the development process and makes bug reports easy to retrieve in the future. However, it also adds overhead to the peer review process and entails great maintenance efforts. Thus, it might be easier to adopt in projects where developers are paid employees like Mozilla. In contrast, Python developers are inclined to annotating bug reports only when they want to engage actions or coordinate workflow. They also keep the tags as few and straightforward as possible. Such flexibility reduces the workload of developers, which saves them time to focus on development work they are interested in. This may fit projects like Python better, because developers are mostly volunteers who are motivated intrinsically.

5.5. Limitations

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, other OSS projects may exhibit differences of additional dimensions when compared with these two cases. Our investigation was focused on bug reports archived by bug tracking systems, but the archived information may be incomplete due to the fact that collaboration in the peer review process may involve the usage of other computer-mediated communication tools [1]. The first author observed a variety of practices other than peer review in these two communities throughout data collection and analysis helped mitigate this gap and allowed us to validate our findings. Other OSS projects that heavily rely on mailing lists to perform peer review may suggest different characteristics of interactions.

Our study is not to explain or identify causal relationships but to qualitatively contrast cases. Thus, it is less concerned with threats to internal validity. For instance, the selection bias refers to the effects introduced by individual/group differences. With respect to external validity, we are focused on large OSS communities. Thus, we suspect that our findings would generalize to the much smaller counterparts, like the projects that have one or two core developers. These two cases were also shaped by their historical trajectories: Mozilla Firefox was transformed from a proprietary software product and developed by a group of developers, whereas Python was created by Guido van Rossum, who remains principal influence on the project. Such particularities may not be directly transferred to other OSS projects that have a different evolution path.

6. Conclusion

In this article, we contrasted peer review practices within two large and distinct open source communities. In particular, we analyzed each of the key activities involved in the OSS peer review process as well as their associated technology affordances and uses. Our investigation provided some interesting findings. First, the different bug reporting styles, *description* versus *code*, afford different focuses of triaging on confirming a bug or highlighting actions in need (particularly patch readiness). The role of triagers also becomes more distinct as the volume of bug reports increases. Second, making design decisions through *individual judgment* or *collective discussions* demonstrate the tradeoffs of design efficiency and feedback gathering. Technical affordance of voting has to match the power distribution among participants in order to mitigate frustration, while also allows inclusive participation and encourages rationale articulation. Third, the different granularity in division of labor, *well specified* versus *ad hoc configured responsibilities*, reflects the different degrees of heterogeneity of technical expertise. To address the tradeoffs of control and spontaneity, designs for enhancing developers' awareness and search capacities require further improvement. Finally, the variance in tool affordance and use, *articulation* versus *minimalism*, present two different ways of design thinking, which depend on the perception of the role of technology in the work process. Articulation helps documentation and knowledge management, while minimalism facilitates collaborative efforts and reduces maintenance cost.

This article contributes alternative designs of social mechanisms and technology infrastructure for OSS peer review practices. It extends previous research that was primarily focused on commonalities across community contexts or a single community. Although two cases can hardly generalize to all OSS projects, they engage users at two sides of the spectrum of technical expertise.

Thus, other projects can adapt those designs according to their positioning at the spectrum. This is not intended to prescribe practices that should or must be adopted by OSS projects, but rather provide learning resources for OSS projects, or even online communities in general, to cope with challenges of harnessing massive peripheral contributions and coordinating efforts of core members. Furthermore, we highlight the emerging role of triagers, which has not received much research effort compared to the core and peripheral participants in virtual communities. We hope our study can foster future investigations on more diverse OSS models.

Acknowledgments

This work is supported by the US NSF (0943023). We thank our partners, the Mozilla and Python organizations for sharing their practices, and also all the reviewers of this article for their suggestions that helped improve it.

References

[1] J. Aranda, G. Venolia, The secret life of bugs: going past the errors and omissions in software repositories, in: Proc. ICSE 2009, ACM Press, 2009, pp. 298–308.
[2] P.M. Bach, R. DeLine, J.M. Carroll, Designers wanted: participation and the user experience in open source software development, in: Proc. CHI 2009, ACM Press, 2009, pp. 985–994.
[3] F. Barcellini, F. Detienne, J.M. Burkhardt, User and developer mediation in an Open Source Software community: boundary spanning through cross participation in online discussions, Int. J. Hum.–Comput. Stud. 66 (7) (2008) 558–570.
[4] A. Barham, The impact of formal QA practices on FLOSS communities–the case of Mozilla, in: Open Source Systems: Long-Term Sustainability, Springer, 2012, pp. 262–267.
[5] D. Bertram, A. Voida, S. Greenberg, R. Walker, Communication, collaboration, and bugs: the social nature of issue tracking in software engineering, in: Proc. CSCW 2010, ACM Press, 2010, pp. 291–300.
[6] S. Breu, R. Premraj, J. Sillito, T. Zimmermann, Information needs in bug reports: improving cooperation between developers and users, in: Proc. CSCW 2010, ACM Press, 2010, pp. 301–310.
[7] A. Capiluppi, P. Lago, M. Morisio, Characteristics of open source projects, in: Proceedings. Seventh European Conference on Software Maintenance And Reengineering, 2003, IEEE, 2003, pp. 317–327.
[8] K. Crowston, H. Annabi, J. Howison, C. Masango, Effective work practices for FLOSS development: a model and propositions, in: Proc. HICSS'05, IEEE, 2005, pp. 197a.
[9] K. Crowston, B. Scozzi, Open source software projects as virtual organisations: competency rallying for software development, IEEE Softw. 149 (1) (2002) 3–17.
[10] K. Crowston, B. Scozzi, Bug fixing practices within free/libre open source software development teams, J. Database Manage. 19 (2) (2008) 1–30.
[11] K. Crowston, K. Wei, J. Howison, A. Wiggins, Free/libre open source software development: what we know and what we do not know, ACM Comput. Surv. 44 (2) (2012).
[12] L. Dabbish, C. Stuart, J. Tsay, J. Herbsleb, Social coding in GitHub: transparency and collaboration in an open software repository, in: Proc. CSCW 2012, ACM Press, 2012, pp. 1277–1286.
[13] L. Dahlander, L. Frederiksen, The core and cosmopolitans: a relational view of innovation in user communities, Organ. Sci. 23 (4) (2012) 988–1007.
[14] N. Ducheneaut, Socialization in an open source software community: a socio-technical analysis, Comput. Supported Coop. Work (CSCW) 14 (4) (2005) 323–368.
[15] K. Finley, Github Has Surpassed Sourceforge and Google Code in Popularity, 2011.
[16] C. Francalanci, F. Merlo, Empirical analysis of the bug fixing process in open source projects, Open Sour. Dev. Commun. Qual. (2008) 187–196.
[17] J. Gamalielsson, B. Lundell, Sustainability of open source software communities beyond a fork: how and why has the LibreOffice project evolved?, J Syst. Softw. 89 (2014) 128–145.
[18] R.A. Ghosh, R. Glott, B. Krieger, G. Robles, Free/Libre and Open Source Software: Survey and Study, International Institute of Infonomics, University of Maastricht and Berlecon Research GmbH, 2002.
[19] T.J. Halloran, W.L. Scherlis, High quality and open source software practices, in: Proc. 2nd Workshop on Open Source Software Engineering, University College Cork, Ireland, 2002.
[20] K. Hamasaki, R.G. Kula, N. Yoshida, A. Cruz, K. Fujiwara, H. Iida, Who does what during a code review? datasets of OSS peer review repositories, in: Proc. MSR2013, IEEE Press, 2013, pp. 49–52.
[21] R. Heckman, K. Crowston, U.Y. Eseryel, J. Howison, E. Allen, Q. Li, Emergent decision-making practices in free/libre open source software (FLOSS) development teams, in: Open Source Development, Adoption and Innovation, 2007, pp. 71–84.
[22] The Institute of Electrical and Electronics Engineers, IEEE Guide Standard for Software Reviews IEEE Std 1028-1997, New York, 1999.
[23] S. Just, R. Premraj, T. Zimmermann, Towards the next generation of bug tracking systems, in: Proc. VL/HCC 2008, IEEE, 2008, pp. 82–85.
[24] A. Ko, P. Chilana, How power users help and hinder open bug reporting, in: Proc. CHI 2010, ACM Press, 2010, pp. 1665–1674.
[25] S. Koch, G. Schneider, Effort, co-operation and co-ordination in an open source software project: GNOME, Inf. Syst. J. 12 (1) (2002) 27–42.
[26] P.B. de Laat, Governance of open source software: state of the art, J. Manage. Governance 11 (2) (2007) 165–177.
[27] C. Lattemann, S. Stieglitz, Framework for governance in open source communities, in: Proc. HICSS 2005, IEEE, 2005, pp. 192a.
[28] A. Mockus, T. Fielding, J.D. Herbsleb, Two case studies of open source software development: Apache and Mozilla, ACM Trans. Softw. Eng. Methodol. 11 (3) (2002) 309–346.
[29] J.Y. Moon, L. Sproull, Essence of distributed work: the case of the Linux kernel, First Monday 5 (11) (2000).
[30] M. Mukadam, C. Bird, P.C. Rigby, Gerrit software code review data from android, in: Proc. MSR2013, IEEE, 2013, pp. 45–48.
[31] E.S. Raymond, The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary, O'Reilly, 2001.
[32] C.R. Reis, R.P. de Mattos Fortes, An overview of the software engineering process and tools in the Mozilla project, in: Proc. Open Source Software Development Workshop, 2002, pp. 155–175.
[33] P. Rigby, D. German, M. Storey, Open source software peer review practices: a case study of the apache server, in: Proc. ICSE 2008, ACM Press, 2008, pp. 541–550.
[34] P.C. Rigby, M.A. Storey, Understanding broadcast based peer review on open source software projects, in: Proc. ICSE 2011, ACM Press, 2011, pp. 541–550.
[35] J.E. Robbins, Adopting OSS methods by adopting OSS tools, in: Proc. the ICSE 2nd Workshop on Open Source, 2002.
[36] F. Rullani, S. Haefliger, The periphery on stage: the intra-organizational dynamics in online communities of creation, Res. Policy 42 (4) (2013) 941–953.
[37] R.J. Sandusky, L. Gasser, Negotiation and the coordination of information and activity in distributed software problem management, in: Proc. GROUP2005, ACM Press, 2005, pp. 187–196.
[38] W. Scacchi, Understanding open source software evolution, Softw. Evol. Feedback: Theory Pract. 9 (2006) 181–205.
[39] C.M. Schweik, Sustainability in open source software commons: lessons learned from an empirical study of SourceForge projects, Technol. Innov. Manage. Rev. 3 (1) (2013).
[40] M. Shaikh, T. Cornford, Version management tools: CVS to BK in the Linux kernel, in: Proc. Taking Stock of the Bazaar: The 3rd Workshop on Open Source Software Engineering, ICSE2003, 2003.
[41] A.C. Strauss, J. Corbin, Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory, Sage Publications Inc., 1998.
[42] M. Terry, M. Kay, B. Lafreniere, Perceptions and practices of usability in the free/open source software (FoSS) community, in: Proc. CHI 2010, ACM Press, 2010, pp. 999–1008.
[43] M.B. Twidale, D.M. Nichols, Exploring usability discussions in open source development, in: Proc. HICSS 2005, IEEE, 2005, pp. 198c.
[44] G. von Krogh, S. Spaeth, K.R. Lakhani, Community, joining, and specialization in open source software innovation: a case study, Res. Policy 32 (7) (2003) 1217–1241.
[45] J. Wang, J.M. Carroll, Behind Linus's law: a preliminary analysis of open source software peer review practices in Mozilla and Python, in: Proc. CTS 2011, IEEE, 2011, pp. 117–124.
[46] J. Wang, P.C. Shih, J.M. Carroll, Revisiting Linus's law: benefits and challenges of open source software peer review, Int. J. Hum.–Comput. Stud. 77 (2015) 52–65.
[47] B. Warsaw, PEP 10 – Voting Guidelines, 2007.
[48] K. Wiegers, Peer Reviews in Software: A Practical Guide, Addison-Wesley, 2001.
[49] J. Xie, M. Zhou, A. Mockus, Impact of triage: a study of Mozilla and gnome, in: ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, 2013, IEEE, 2013, pp. 247–250.
[50] Y. Yamauchi, M. Yokozawa, T. Shinohara, T. Ishida, Collaboration with Lean Media: how open-source software succeeds, in: Proc. CSCW 2000, ACM Press, 2000, pp. 329–338.
[51] X. Yang, R.G. Kula, C.C.A. Erika, N. Yoshida, K. Hamasaki, K. Fujiwara, H. Iida, Understanding OSS peer review roles in peer review social network (PeRSoN), in: Proc. APSEC2012, IEEE, 2012, pp. 709–712.
[52] Y. Ye, K. Nakakoji, Y. Yamamoto, K. Kishida, The co-evolution of systems and communities in free and open source software development, Free/Open Source Softw. Develop. (2005) 59–82.
[53] R.K. Yin, Case Study Research: Design and Methods, vol. 5, Sage, 2003.