The Role of Patch Review in Software Evolution: An Analysis of the Mozilla Firefox

Mehrdad Nurolahzade, Seyed Mehdi Nasehi, Shahedul Huq Khandkar, Shreya Rawal

Department of Computer Science

University of Calgary

2500 University Dr NW, Calgary, AB T2N 1N4

{mnurolah,smnasehi,shkhandk,srawal}@ucalgary.ca

ABSTRACT

Patch review is the basic mechanism for validating the design and implementation of patches and maintaining consistency in some commercial and Free/Libre/Open Source Software (FLOSS) projects. We examine the inner-workings of the development process of the successful and mature Mozilla foundation and highlight how different parties involved affect and steer the process. Although reviewers are the primary actors in the patch review process, success in the process can only be achieved if the community supports reviewers adequately. Peer developers play the supporting role by offering insight and ideas that help create more quality patches. Moreover, they reduce the huge patch backlog reviewers have to clear by identifying and eliminating immature patches.

Categories and Subject Descriptors

D.2.5 [Software Engineering]: Testing and Debugging; D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement

General Terms

Human Factors, Management, Measurement

1. INTRODUCTION

Following the success of FLOSS projects in the last two decades various studies have been performed to gain insight into this model of development [1][9][15][16][18]. The basic principles of FLOSS development are clear enough, but the details can certainly be difficult to define. Open source developments typically have a central person or body that selects a subset of the developed code for the official release and makes it widely available for distribution [1]. The selection takes place after code is exposed to the development community and reviewed by people both inside and/or outside the core development community [9]. Code review not only works as an important quality assurance mechanism in both commercial and FLOSS settings, but also enables learning and knowledge transfer in the software development team. This is especially essential to the FLOSS model where development is usually driven by a virtual team that is geographically distributed over multiple time zones.

In this paper, we examine the process of incrementally submitting and integrating patches into Firefox (one of the core software evolution activities in Mozilla). The review ensures that the patch adheres to the initial requirements, commonly accepted standards, does not introduce inadvertent errors and unwanted side effects to the common code base. "Mozilla community has decided that it can't accept just any change to be integrated into the public central Mozilla code base. If you want your code to become a part of it, you need to follow rules. These rules are not like law, but basically you must convince people that your change is good."[1] The Firefox project is one of the successful projects in the FLOSS world. Firefox is the second most popular browser and its market share has been steadily growing over the last year.[2] The hybrid of open source and commercial characteristics of development in Mozilla has led to a process model that heavily relies on patch reviews performed by module owners, who act as the gatekeepers [19] in the development community.

This study combines quantitative and qualitative research methods on data drawn from the Firefox Bugzilla repository. Products like Firefox that use a mixture of commercial and FLOSS development process have become widespread. This research was designed to help better understand the dynamics of this development model by trying to answer the following research questions: What are the roles involved in the patch review process? What is the process of conducting reviews? When are reviews performed? What do reviewers look at and what they possibly miss?

More specifically, we make the following contributions:

- *Statistical analysis of reviewer behavior.* We found that most reviews are performed by core community members. Additionally, a substantial amount of reviews take place in the first 24-48 hours after patch submission. Interestingly, on average peers conduct their review before module owners.
- *Identification of reviewer focus points and concerns.* We analyzed reviewer comments and categorized different types of feedback provided by reviewers. We found that peers are

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

IWPSE-EVOL'09, August 24–25, 2009, Amsterdam, The Netherlands.

Copyright 2009 ACM 978-1-60558-678-6/09/08...$10.00.

[1] https://developer.mozilla.org/en/Mozilla_Hacker's_Getting_Started_Guide

[2] http://marketshare.hitslink.com/browser-market-share.aspx?qprid =1

more interested in the functionality and usability aspects of the product while module owners are more concerned about the quality and long-term maintainability of the project.

- *Developer and reviewer behavioral patterns.* We identified a number of interesting patterns in developer and reviewer behavior, for example, *Patchy-Patcher* and *Merciful Reviewer* that are described in our findings (Section 3).

The remainder of this Section discusses an overview of the Mozilla software evolution process. In Section 2 the data collection and analysis method is described. Section 3 presents the results uncovered through the study. In Section 4 we discuss our findings and the limitations of the study. A summary of related work is given in Section 5. We conclude in Section 6 by summarizing our findings and presenting ideas for future work.

1.1 PATCH EVOLUTION IN MOZILLA

Mozilla tries to incorporate a hybrid of the quality of the commercial and scalability of the FLOSS development models. Mozilla, unlike the typical FLOSS project, has a relatively large core development group [1]. At the same time, it enforces code ownership similar to commercial projects. This is contrary to some FLOSS projects that rely on unofficial ownership structures, which means you can commit code without being required to seek approval from a source of authority.

One of the challenges in FLOSS development model is managing the loosely coordinated contributions of participants. Some FLOSS projects rely on decision-making mechanisms like voting to manage the chaos of open source development [5]. Mozilla also utilizes voting for incorporation of enhancements and new features. However, the final decision is made by the responsible module owner and is not steered by voting. In other words, as Mike Beltzner the development director of Firefox puts it, "Anyone can propose a change. Anyone can comment on a proposal for change. Anyone can submit a change to the code. Not everyone can approve a change." [12] Module owners in Mozilla are the equivalent of leaders in a typical commercial project. They have a precise understanding of a subset of the product and its dependencies with the rest of the product or related projects. Their objective is to bring order to development by leveraging and coordinating resources. A good example is the Mozilla's "Won't Fix" status, which is used by module owners to mark those bug reports or enhancements that they are not going to fix. This works like a control mechanism for module owners to show what the priorities of the project are and what they do not want to include in the final product. In other words, Mozilla reserves the right for module owner to veto a change request if need be. Of course, this also comes at a price and may cause disputes and disagreements in the community [13].

Mozilla module owners should facilitate good development as defined by the developer community. Code review is used as the basic mechanism for validating the design and implementation of patches. Before code is checked into a source code repository the appropriate module owner and possibly peers must review it. The patch submit-review process involves developer submitting his/her patch to Bugzilla by attaching it to the appropriate bug report. Submitters may submit patches that serve as bug fixes or patches that provide an enhancement or a new feature. Reviewers then read through the submitted patch and comment on it. Mozilla requires patch reviews to be done by module owners and possibly peer volunteer developers. This distinguishes Mozilla from some FLOSS projects that primarily rely on inspections performed by volunteer developers. Based on the feedback received by the reviewers, the submitter enhances the patch. Once the patch is deemed acceptable, it is committed to source code repository by the developer (if he has write privilege to source code repository) or a core member of the module.

Developers should build, run, and test their solutions before submitting the patch. They are encouraged to include tests in their patch as well.[3] After submitting a patch the developer may ask for a code review by changing the status of bug report to *review?* or *super-review?*. Some patches have to undergo two reviews: a regular review and a "super-review". Reviewers are specific to given areas of the code base, but any super-reviewer may review a patch. Each module has an owner and zero or more peers who can perform the review. Review requests are forwarded to the module owner by default, but the module owner may assign one of his peers to perform the review. If there is not a response from the reviewer, it is the responsibility of the developer to contact the module owner and remind him/her to perform the review. If developer does not get a response within a week and believes the patch deserves rapid attention, s/he can ask in Bugzilla or IRC (in #developers channel) who else can review patches for current component and forward his request to them instead. The patch-review process is often iterative, but reviewers never fix code themselves. The reviewer asks for modifications and the developer is expected to apply them and resubmit the patch. Reviewers indicate the approval or rejection of the patch by flagging the patch *review+*, *super-review+*, and *ui-review+* or *review-*, *super-review-*, and *ui-review-* respectively.

When a patch is finally approved it is checked into the product tree. After check-in, the person behind it should be available for the next hour or two in case something goes wrong with his/her check-in. This is roughly the time it takes to get unit test results from all platforms. Every day one person from the Mozilla community is selected to watch over the build tree, make sure unsuccessful builds get traction, call people on the phone, etc. This person is called a "sheriff". For reasons like build breakage, performance regressions, or test regressions, the sheriff could close the tree to further check-ins. The sheriff reopens the tree when he has confidence that the regressions are diagnosed, being fixed, and Tinderbox[4] is clear enough.[5] For performance regressions the sheriff asks people to explain whether their check-ins are responsible for the regression. If somebody could not show that his/her check-in is not related to the regression, the sheriff will back the check-in out.[6]

2. METHOD

Our study relies on data extracted from the Mozilla Bugzilla database. A Bugzilla database collects bug reports that are submitted by reporters with a short description and a summary. Bugzilla also captures the status of a bug, for example, UNCONFIRMED, NEW, ASSIGNED, RESOLVED, or CLOSED. The resolution of a bug is captured separately from its status, for example, FIXED, DUPLICATE, or INVALID. Details

[3] https://developer.mozilla.org/en/Creating_a_patch

[4] https://developer.mozilla.org/En/Tinderbox

[5] https://wiki.mozilla.org/Sheriff_Duty

[6] http://www-archive.mozilla.org/hacking/regression-policy.html

on the life cycle[7] of a bug can be found in the Bugzilla documentation.[8] For our analysis, we developed a program that extracted data from Bugzilla into a local database. To extract data from Bugzilla, its XML export feature was used. Bug report comments and history were retrieved directly from the web interface of Bugzilla. Additionally, we developed a tool on top of our database schema to facilitate qualitative coding and quantitative analysis of the bug descriptions, status changes, and developer comments.

Our research questions cannot be answered by solely conducting a quantitative analysis on the Mozilla bug repository. Identifying an instance of review and the type of feedback provided by the reviewer requires qualitative analysis of the bug report comments. Therefore, we have combined both qualitative and quantitative methods in this study in order to properly analyze the Mozilla patch evolution process. We examined 112 randomly selected bug reports from the Mozilla Firefox project. We retrieved a list of bug reports through the web interface of the Mozilla Bugzilla[9] by specifying Firefox as the product and leaving component, version, severity, priority, and the rest of the fields untouched. We then randomly went through the search results and selected 112 bug reports. We narrowed our focus to those bug reports that entailed at least one patch submission and review to ensure that patch review takes place in all bug reports in the selected sample. Our sampled data set contains 66 bugs, 38 enhancements and 8 new features filed between the years 2002 and 2009.

We found that 67 developers submitted 310 patches. Those patches received 318 reviews by 66 peer developers and 38 module owners (or module owner peers). The median number of patches per bug reports is 2, with a standard deviation of 2.88. Figure 1 shows the distribution of reviews per bug report. For each of the selected bug reports, we carefully examined the fields, comments, attachments, and status changes to identify discussions and events. To answer our research questions we were primarily interested in discussions related to patch submission and review events. Three researchers performed initial coding based on a coding guideline, which was developed by open coding a sample data set. A fourth researcher revised the data set in the end, to increase intercoder reliability.

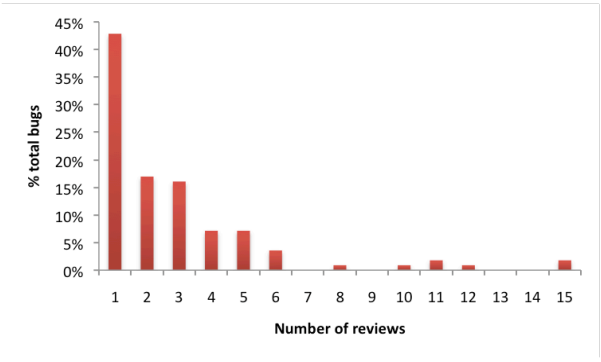

Figure 1. Number of reviews per bug report.

[7] http://www.bugzilla.org/docs/3.2/en/html/lifecycle.html

[8] http://www.bugzilla.org/docs/3.2/en/html/

[9] https://bugzilla.mozilla.org/query.cgi

3. FINDINGS

We performed various observations on the Mozilla patch review related events and found interesting patterns in the way developers and reviewers are submitting, reviewing, and integrating patches into the main development trunk of the project.

3.1 PATTERNS

3.1.1 PATCHY-PATCHER

Submitting a patch does not necessarily mean that it is finished or takes care of the bug report. A patch can be considered finished when the developer formally asks for a review.

"wip 0.1 works, but i need to fix labels and finish fixing test_history_sidebar" (Bug #390614)

A work in progress (WIP) patch may not even build, but can still be used as a platform to discuss the solution. A developer may partially know about the solution, but still might need help from the community for the missing parts. WIP patches are also used as a means of progress reporting. The Mozilla developer guidelines advise developers to work on multiple bugs in parallel. "Difficult bugs may take several days or weeks to complete, plus the time for reviews."[10] Using Bugzilla to keep track of work in progress helps developers not to lose context when switching between bug reports.

3.1.2 NEWCOMER

The dynamics of immigration of newcomers [4] to FLOSS projects has been studied in the past. Mozilla comes with a pool of documentation to help new developers get oriented with community development standards and practice. However, as expected not everyone goes through all the documents before filing a bug report or submitting a patch. Moreover, we observed that there is a discrepancy between the accepted notion of practice and the documented version. It is not unusual to see developers stuck wondering what the next step is, or doing something based on their own intuition that can of course be misleading.

"Should I attach a new patch for new review?" (Bug #416728)

For example in response to the comment above, the reviewer replies that there is no need for new review; but unless he is able to check the new patch in, he should attach a new patch.

3.1.3 MERCIFUL REVIEWER

The Mozilla review guide states that module owners and super-reviewers use the *review-* and *super-review-* flags respectively to communicate that patch failed the review. Our data set contains only 42 instances of *review-* while the number of review failures we counted was 56. This discrepancy between the two figures shows a tendency in reviewers not to use *review-* flags. In those situations, the reviewer leaves a comment on the patch and resets bug status from *review?* to nothing.

3.1.4 DOUBTFUL REVIEWER

If a patch entails changes to existing functionality of the system, the decisions regarding new functionality is left to the community. This is when the bug report assignee, peer developers, and module owner(s) engage in a discussion regarding different aspects of

[10] https://developer.mozilla.org/en/Mozilla_Development_Strategies

new functionality. However, we noticed cases when the bug report does not attract much of community attention and the reviewer expresses his/her uncertainty of the proposed solution.

"Let's put this in for beta, and make sure we blog about the change a little,and see the reaction." (Bug #412862)

The approach developed by reviewers in this case is to accept the patch and wait for the community's reaction to the new/changed functionality when they see it in the new builds (late feedback). Alternatively, a reviewer can delay his verdict until peer developers step in and express their opinion (early feedback).

3.2 OBSERVATIONS

3.2.1 ASSIGNMENT

A Developer can submit a patch to a bug report that is not assigned to him. Yet, most patches are developed by assigned developers. Merely 36 out of 310 (11.6%) patches in our data set were developed by peer (non-assignee) developers.

3.2.2 CORE DEVELOPERS

Our data set contains patch contributions by 67 developers who have contributed 310 patches to 112 bug reports. But, the distribution of developer contribution to patch development is not even. Top developers (that make up 25% of the developer population) have contributed the majority of the patches (64.5%). We also verified how many bug reports each developer has contributed to. Likewise, the top developers have contributed to the majority of bug reports (55.6%) by submitting patches.

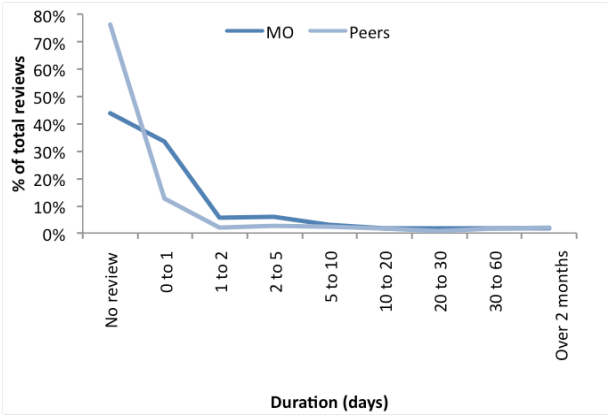

Figure 2. Patch Review Time.

3.2.3 MODULE OWNER REVIEWS

Checking into most (but not all) of the Mozilla tree requires another level of pre-check-in code review. This level of review is done by a group of strong developers and is referred to as "super-review".[11] Our data set contains 38 reviewers that did a total of 198 code or UI inspections. The Mozilla reviewers are expected to provide some sort of response within 24 hours.[12] Figure 2 shows the timing of patch reviews in our data set. While module owners do not review 43.8% of patches, 33.4%, 5.5%, and 5.9% of module owner reviews happened within 24 hours, 2 days, and 5 days after patch submission. We noticed a minor UI enhancement patch that waited 435 days to be reviewed. The problem is finally resolved when one of the developers notices it and brings it to community attention.

"I'm not sure of the procedure here, but a patch has been waiting for review for nearly a year now, and nothing has happened." (Bug #182928)

Of the total of 198 review decisions in our data set, 86 are acceptance and 112 are rejection. Table 1 shows the different feedback types provided by module owners and peers. Analysis of the reviewer decisions shows that the primary reasons for rejecting a patch are implementation and design issues. Documentation, coding standards, and functionality are almost weighed the same by reviewers when rejecting a patch. However, this cannot be interpreted as reviewers lean towards the implementation and design quality of Firefox than the documentation and coding style quality of it. Reviewers even notice minor programming malpractices like bad naming of variables and functions, block indentation, and inconsistency between code and comments.

Table 1. Reviewer feedback types

Feedback	Module Owner	Peer Developer
Implementation	63	45
Functionality and Usability	6	31
Documentation	9	1
Coding Standards	7	3
Performance	1	2

3.2.4 PEER REVIEWS

Most submitted patches (76.1%) get no peer developer review and only 17.3% of patches get only one such review. 12.4% of patches are reviewed by peers within 24 hours (See Figure 2). Our data set contains patch review contributions by 66 peer developers. Here again like patch development the contribution is not evenly distributed between peers. A core group of peers (23%) did the majority (63%) of the reviews. The analysis of the list of top peer reviewers revealed the fact that they are all core developers of the community. Peer developers primarily express their opinion on implementation, functionality/usability, and design aspects of proposed patches (see Table 1). Documentation and coding standards receive relatively much lower attention as compared to former concerns. Peer developers are more out spoken when they see something wrong with a patch, otherwise they rather stay silent.

"I'm actually pretty sure this is the wrong fix" (Bug #464792)

80% of comments by peer developers are either negative or partly negative.

"This method name [IsChildrenVisible] is a little off... maybe "AreChildrenVisible", or "HasVisibleChildren"?" (Bug #323492)

[11] http://www.mozilla.org/hacking/reviewers.html

[12] https://developer.mozilla.org/en/Code_Review_FAQ

A reviewer may need further detail regarding the patch implementation from the developer in order to provide his/her feedback. Likewise, the developer might have problem interpreting reviewer feedback and therefore may need to ask further questions to clarify the situation.

"Those lines aren't in that function; I'm unsure what this comment is about." (Bug #450340)

Discussions around review may also attract peer developers to get involved and express their opinion on developer or reviewer comments, which might affect the new patch that developer is going to submit.

3.2.5 PEERS VS. MODULE OWNERS

The majority of bug reports (45.9%) get only one review that is a module owner review. At the same time, module owners decide by themselves in 89.4% of cases, because no peer developer commented on the patch before them. On the other hand, our data set shows that on average peer developers tend to review developed patches before module owners. We also verified what motivates patch developers to resubmit a patch. Resubmission happens when a module owner, the patch developer, or a peer developer (ranked respectively) rejects a patch or finds room for improvement in it.

3.2.6 RUBBER STAMPS

According to the Mozilla review guidelines, if the type of correction required is small and simple enough that a review is not needed then the reviewer can "rubber stamp" the patch. A rubber stamped patch is flagged *review+* but the reviewer also supplies the list of minor fixes expected.

"This extra </handlers> makes this not work, r=me with this removed" (Bug #346079)

The developer is expected then to make the corrections, submit a new patch, and check-in the new patch.

"Yikes, had a bad copy and paste in that one" (Bug #346079)

We counted 31 instances of rubber stamp given in our data set in which minor implementation (18), coding standards (7), and documentation (6) corrections were requested by reviewers.

3.2.7 MULTIPLE REVIEWERS

In certain cases a patch requires more than one review in order to be checked into source code repository. A patch that changes code in more than one module must receive a *review+* from each module owner. If the first reviewer feels that the patch would benefit from additional reviews, they should request a second review from an appropriate person. Also, significant user interface & experience changes should get ui-review from someone in the UI group. In our sample, more than one module owner reviewed 23 patches, one module owner inspected 145, and no module owner reviewed the rest of them.

3.2.8 UNDISCOVERED ERRORS

Code review is not expected to find all errors in the code [7][2]. Mozilla accepts the fact that reviewers may miss things during their review. 8.4% of patches in our data set passed review and went into source code repository but later were backed out or replaced with another patch. We verified that performance issues and regressions caused as a result of merging current product tree into main tree are the primary errors that remain undiscovered during review.

"Based on the site breakage, re-opening and suggesting we back out this change." (Bug #412862)

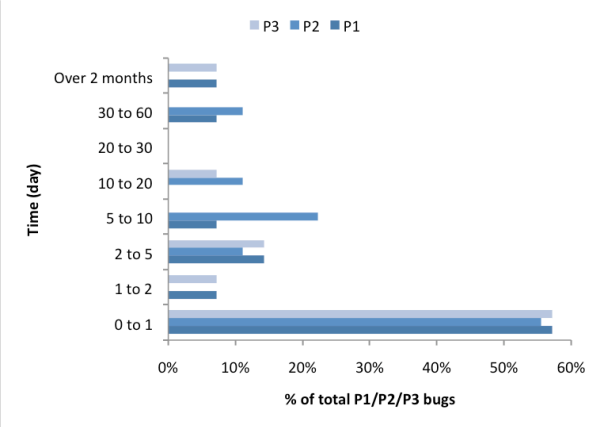

Figure 3. Patch review time by bug report priority.

If the undiscovered error causes a build breakage, test or performance regression then it is easily identified during automated builds. Otherwise, it can make its way into next release until someone notices and files a new bug report for it. Bug reports that are suspected to be a regression are marked with the keyword "regression" in Bugzilla. If one or more suspected patches that caused the regression are identified, the two bug reports are marked as related. Additionally, the original bug report is reopened to further pursue the problem.

"Adding bug 418643 to the dependency list which has been introduced this problem 3 month ago." (Bug #477739)

3.2.9 DEVELOPMENT EFFORT

The chance to get a patch into source code repository is 53.55%. In other words, on average one of every two patches developed is thrown away by the developer or fails the review. We did not analyze the characteristics of patches that get accepted, but a study like Weißgerber et al. [17] would be interesting to conduct in the future.

3.2.10 BUG PRIORITY

High priority bug reports are expected to receive higher developer and reviewer attention. The idea behind assigning priority to bug reports is to show the level of interest in resolution of those bug reports. Developers are considering bug reports based on their priority; 65% of P1 (highest priority) bug reports get their first patch within 24 hours, while 22% of P2 and 13% of P3 bug reports get their patch in similar time. Unless the developed patches are reviewed as soon as possible, faster resolution of higher priority bugs cannot be achieved. Figure 3, shows the review time of P1, P2, and P3 bug reports in our sample. There is no significant relationship between the time reviewers consider patches and bug report priority. While 57% of P1 patches received a review within 24 hours, also 55.5% and 57% of P2 and P3 patches respectively received a review within 24 hours. Hence, reviewers are examining patches, irrespective of their bug report priority.

4. DISCUSSION

Figure 4 provides a conceptual model of the patch evolution process space in Mozilla. The process is shaped around a problem statement (bug report), that is either a defect found in the product or a new enhancement or feature to be incorporated. The resolution of the problem happens through developing a solution, which is delivered in the form of a patch. The problem and the associated solution are tightly tied together and can influence one another. The three parties involved (developer, peer, and module owner) work closely to define the problem and candidate solutions, further refine the selected solution, and finally resolve the problem (close the bug report). Hence, the set of activities taking place in the relationship of the above parties can be described as define, refine, and resolve.

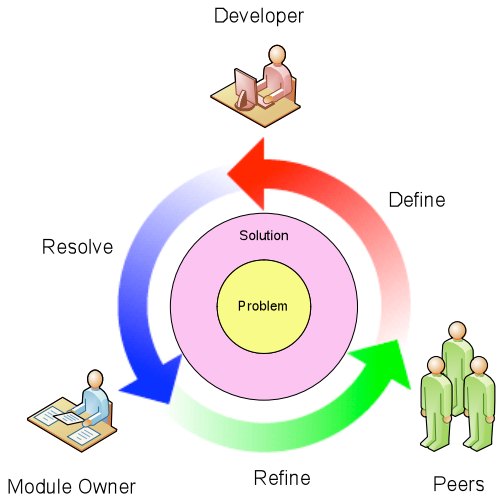

Figure 4. A conceptual model of the Mozilla patch evolution.

4.1 DEFINE

In Mozilla, there is a strong correlation between assignment and patch submission. This is quite contrary to the common view held in FLOSS development in which everyone contributes a patch to the problem and the best patch is chosen by voting or is selected by a member of the core development group. Peer developers tend to invest their effort in engaging in discussions with the assigned developer before and after solution development, rather than trying to develop alternative solutions themselves. In other words, they help developers understand the problem and solution spaces. They provide alternative explanations for the problem, identify related problems, propose alternative solutions, verify the developed solution, and spot misalignment between the developed solution and the original problem.

If a developer is not competent in the field of the problem or does not have the required information to provide a solution, he will turn to peer developers for advice. Alternatively, assigned developer may act as mediators between parties interested in solving a particular problem. The mediator [15] negotiates between those competent parties who can provide insight and those who are willing to produce a patch and drives forward the process until the solution is developed. This is one of the occasions where the *Patchy-Patcher* pattern of development may be seen.

4.2 REFINE

Module owners are usually overwhelmed with the number of patches they have to review and bug reports they have to comment on. Peer reviews can be very helpful to patch developers in finding their mistakes and misalignments with community standards early on. Although non-member peers do not contribute to patch reviews very much on an individual basis, the combined contribution of them is still substantial. Peers comment on a little less than a quarter (23.9%) of submitted patches. It may not seem like a great help to module owners, but the reality is that peers also contribute before patch submission by helping the developer better define the problem/solution. If peers do not find something wrong with a patch, then there is no reason to leave a comment. In most FLOSS communities the absence of negative voice is treated as agreement with a proposed solution. Our result, showing 80% of peer comments are either negative or partly negative about the submitted patch, is backing up this view. On the other hand, although peers do not review patches as much as module owners do, their effect on identifying and eliminating immature patches is still valuable. Module owners only review 33.4% of patches within 24 hours, but adding up the 43.8% of patches that are not reviewed by module owners (because they are WIP patches or the developer or peers found something wrong with the patch and eliminated it) and the number of patches reviewed within the next 24 hours, the total of patches examined within 48 hours would be 82.8%.

We noticed that there are noticeable differences in the contribution of peer developers and module owners to refining the solution. Peer developers seem to be not willing to review a patch when a module owner has already done so. Rather they prefer to express their own view on module owner's comment by either questioning the feedback provided or trying to clarify it for the developer by providing further context or examples. Module owners, on the other hand, prefer to wait until peer developers first review a patch. If the peer comments result in a resubmission of the patch then that saves them one review. After all, reviewers are the expensive resources of the community, any reduction in reviews that does not lead to a reduction in the number of defects found will result in cost saving.

Implementation and design aspects of developed patches receive high attention from both peer developers and module owners. However, peer developers are providing more functionality and usability feedback to patch developers than module owners, and at the same time they are commenting less on documentation and coding style aspects of the developed patches. Comparison of the top developers list and top peer reviews list shows that, except a few exceptions, the most active peer reviewers are also active developers themselves. Reviewing the code is a more tedious task than commenting on the functionality and usability of the implemented solution. Besides, the primary interest of the external developers, who are users of the system [5], is the functionality and usability aspects of the product. Our analysis shows that the *Doubtful Reviewer* pattern usually happens when changes are being made to functionality and usability of the system. These problems are not necessarily of the kind of nature that module owners are an expert in. This is when the community has to step in and express its preferences [12]. In this regard, a peer focusing more on functionality and usability aspects of the product, at the cost of disregarding the maintainability of it, is still beneficial.

Inevitably, the module owners have to make up for that by focusing more on the quality and long-term maintainability of the

project. In addition, the combined number of coding standards and documentation rubber stamps given by reviewers is still less than rubber stamps given for minor implementation issues. Reviewers are more careful with a change that might disturb the long-term maintainability of the product. Deviations in coding standards and documentation can easily creep into the code base, but minor implementation issues are in good chance to be discovered by automated tests or by the community. Module owners and peer developers are complementing each other. They refine the developed solution based on their interest/concern in the overall process.

4.3 RESOLVE

Our findings are similar to previous findings from FLOSS and commercial [1] projects that show the core development group provide most of the functionality developed. It is reflecting the part-time nature of external participation as compared to the full-time commitment of core developers. Although most external developers do not participate frequently in development on an individual basis, but the combined contribution of them is substantial. Back in 2007, Firefox was estimated to have 1000 contributors that made 100 contributions daily [12]. 37% of the code contributed to Firefox between November 2006 and April 2007 came from the community [12]. Module owners do rarely participate in development, but they have a group of peers that are part of the core development team and do most of the development. While we expected the top developers to be either the module owners or their peers, we found developers that are actively contributing to development and are not listed as module owner peers. This can be attributed to the special policy that Mozilla has in order to "find and elevate smart contributors" [12] to take on the role of module owners or their peers in the future. Despite the extensive control of module owner on the Mozilla development process, they still handle external developers with great care. Although, we did not find any sign of disgrace or discontent in developer's behavior upon receiving *review-* in our data set, but one can expect to see such a reaction from time to time. However, module owners seem to have developed this strategy over the years not to deter new or casual developers from contributing to the community. After all, it is commonly believed that "Teams with practices to attract contributions from more developers will be more effective" [11]. Similarly, we noticed that despite the clumsiness of the *Newcomers*, which introduce delays and complications to the process, the community still copes with them patiently.

In addition to mutual respect, trust plays an important role in the relationship between module owner and the developer. Rubber stamping a patch results in skipping the review step of the new patch, which increases the speed of the patch review process and decreases the workload of the reviewer. However, rubber stamps are not given to all developers, as one of the module owners puts it in the developer forum "It's more likely to happen with a known contributor, because it's far too common for people to screw up changes like this (making code changes in addition to the comment change, screwing up the diffing, etc). You'd think this would be simple, but apparently for some people it's not."[13] Module owners rely on assigned developer to manage the lifecycle of a bug report from the beginning to the very end. The bug report resolution does not happen by submitting a patch. Developers are expected to be present and offer their services during and after the patch is checked into source code repository and automated builds are run.

4.4 THREATS TO VALIDITY

Of course our study also suffers from several limitations. Aranda and Venolia [8] found that "Bug reports are strongly dependent on social, organizational, and technical knowledge that cannot be solely extracted through automated analysis of software repositories." Similarly in our study, an instance of review is considered to be the explicit feedback provided by the module owner or toggling of review flag. However, developers and reviewers rely on communication channels other than Bugzilla to discuss patch and review related topics. Therefore, those instances of review that have taken place over email, IRC, or IM communications between patch developers, peers, and module owners have not been considered. Despite the above limitation, we have considered those review instances that the developer implicitly refers to an instance of communication between him and the module owner that motivated him to submit a new patch.

Our sample data set contains only 112 of thousands of bug reports residing on the Mozilla Bugzilla server. Our sample size might be small, compared to samples used in related MSR work, but our study primarily relies on qualitative analysis of the sampled bug reports. Unlike quantitative analysis, conducting qualitative analysis on a larger sample requires considerable amount of effort. The natural step to address the above concerns would be to talk to the Mozilla developers involved and have them verify our findings.

The generalizability of the results presented here can be evaluated by examining bug reports and patch evolution processes in other FLOSS communities with different characteristics. The research reported here relies on reading sampled bug reports and information acquired from the Mozilla community web site, developer forum, and a few published talks and interviews with key community members. Other sources of information like the code repository, developers IRC channels, chat history, and additional methods like interviewing can be used in the future qualitative examinations of patch review process allowing analysis of information not accessible using our present method.

5. RELATED WORK

The code review process has been discussed both in the context of commercial and FLOSS projects. Mäntylä and Lassenius [14] studied the type of defects that were discovered by the code reviewers and found that 75 percent of defects did not have any effect on the visible functionality of the software, but they "improved software evolvability by making it easier to understand and modify". Our findings also show that module owners closely look at non-functional aspects of the developed patches. Those non-functional defects are hardly tracked by automated tests and should be discovered through module owner inspections.

The development process of Mozilla and other FLOSS projects have been investigated in many studies. Mockus et al. [1] studied various aspects of the Apache web server and the Mozilla browser and compared them with commercial projects. They come up with several hypotheses based on the analysis of the Mozilla data and provide a description the development process. Asundi and Jayant [9] described a generic patch submit-review process in FLOSS

[13] http://groups.google.com/group/mozilla.governance/browse_thread/thread/f01ea12ff3c36522/98955c0068c9d923?hl=en

projects. They found that although the patch review process is not the same across various FLOSS projects, the core members across all projects play a crucial role. Crowston and Scozzi [10], like the former two studies, found that there are striking differences in the level of contribution to the open source development process. The most active users carried out most of the tasks while most others contributed only once or twice. The contribution power law distribution has similarly been observed by Wilkinson [6] in an analysis of four systems including Mozilla's Bugzilla. Identically, we observed the same phenomenon in developer and reviewer contributions.

Sandusky and Gasser [18] use data drawn from Mozilla to study the role of negotiation in software problem management in the context of FLOSS projects. Our qualitative analysis of bug reports in order to identify patch submission and review discussions is similar to theirs. Halverson et al. [3] did a study on the Mozilla development community and presented two prototypes to aid coordination and management needs of software development work. They also identify a few social and technical patterns of behavior in the development process. Rigby and German [15] studied four open source projects including Mozilla and found some commonalities in the review process of them: each project had a coding standard; projects required contributors to update documentation; and they emphasized that patches should be separate and no patch should add large functionality, because small patches are easier to review. Rigby et al. [16] provided a description of Apache review process and showed quantitatively that Apache development relies on frequent reviews of small pieces of functionality.

6. CONCLUSION

Our goal in this study was to better understand the patch evolution process of the Mozilla development community. We quantitatively measured parameters related to the process, explained the inner-workings of the process, and identified a few recurrent patterns in developer and reviewer behavior. Most development and peer reviews in Firefox come from a group of developers who make up the core development group. Assigned developers are primarily responsible for bug reports and are supported by peer developers and module owners. Peers play a key role in the process by providing ideas before a patch is developed and reviewing developed patches before module owners and finding and reporting back errors. This results in decrease in module owner workload. On the other hand, while module owners are concerned about the long-term maintainability, peers seem to be interested in the functionality and usability of the product. This preference also benefits module owners because feedback from the community helps them better decide on these aspects of the product.

Despite its limitations, a number of interesting questions have been raised by our work that can be investigated in future research. We noticed that the level of attention to bug reports is not evenly distributed. The popularity phenomenon has also been observed by Wilkinson [6] in four online systems including Mozilla's Bugzilla. What types of bug reports attract more community participation in the form of discussions before and after patch submission? Likewise, there are bug reports that receive no community attention. What type bug reports are resolved without receiving any community attention beyond the assigned developer and the module owner? An interesting research question to be investigated in the future is: How the community discussions before patch submission affect the patch development and review process? In order to assess the effectiveness of the review process, further research is needed on the nature and treatment of undiscovered errors. Is there a common pattern in occurrence of undiscovered errors?

7. ACKNOWLEDGMENTS

The authors of this paper feel indebted to Jonathan Sillito for the insightful and fruitful discussions throughout this research. We also wish to extend our thanks to Frank Maurer, Tom Zimmermann, David Ma, and the unknown reviewers for their constructive comments on earlier versions of this paper.

8. REFERENCES

[1] A. Mockus, et al., Two case studies of open source software development: Apache and Mozilla, ACM Transactions on Software Engineering and Methodology (TOSEM), v.11 n.3, p.309-346, July 2002.

[2] B. Boehm, V. R. Basili, Software Defect Reduction Top 10 List, Computer, v.34 n.1, p.135-137, January 2001.

[3] C. A. Halverson, et al., Designing task visualizations to support the coordination of work in software development, Proceedings of the 20th anniversary conference on Computer supported cooperative work, November 2006.

[4] C. Bird, et al., Open Borders? Immigration in Open Source Projects, Proceedings of the 4th International Workshop on Mining Software Repositories, May 2007.

[5] C. Gacek, B. Arief, The Many Meanings of Open Source, IEEE Software, v.21 n.1, p.34-40, January 2004.

[6] D. M. Wilkinson, Strong regularities in online peer production, Proceedings of the 9th ACM conference on Electronic commerce, July 2008.

[7] F. Shull, et al., What We Have Learned About Fighting Defects, Proceedings of 8th International Software Metrics Symposium, Ottawa, Canada, pages 249–258. IEEE, 2002.

[8] J. Aranda, G. Venolia, The Secret Life of Bugs: Going Past the Errors and Omissions in Software Repositories, Proceedings of the 31st International Conference on Software Engineering (ICSE'09), May 2009.

[9] J. Asundi, R. Jayant, Patch Review Processes in Open Source Software Development Communities: A Comparative Case Study, Proceedings of the 40th Annual Hawaii International Conference on System Sciences, January 2007.

[10] K. Crowston, B. Scozzi, Coordination Practices for Bug Fixing within FLOSS Development Teams, Proceedings of the First International Workshop on Computer Supported Activity Coordination (CSAC 2004), April 2004.

[11] K. Crowston, et al., Effective work practices for FLOSS development: A model and propositions, Proceedings of the 38th Annual Hawaii International Conference on System Sciences, 2005.

[12] M. Beltzner, Embracing the Chaos: Designing For and With Community, Session in Web 2.0 Expo, April 2007.

[13] M. Conor, The Life Cycle of a Bug, Mozilla Video Presentations, https://developer.mozilla.org/En/Video_presentations, Feb 2007.

[14] M. V. Mäntylä, C. Lassenius, What Types of Defects Are Really Discovered in Code Reviews?, to be appeared in IEEE Transactions on Software Engineering, 2009.

[15] P. C. Rigby, D. M. German. A preliminary examination of code review processes in open source projects, Technical Report DCS-305-IR, University of Victoria, January 2006.

[16] P. C. Rigby, et al., Open source software peer review practices: a case study of the apache server, Proceedings of the 30th international conference on Software engineering, May 2008.

[17] P. Weißgerber, et al., Small patches get in!, Proceedings of the 2008 international working conference on Mining software repositories, May 2008.

[18] R. J. Sandusky, L. Gasser, Negotiation and the coordination of information and activity in distributed software problem management, Proceedings of the 2005 international ACM SIGGROUP conference on Supporting group work, November 2005.

[19] T. J. Allen, Managing the Flow of Technology, MIT Press, 1977.