Bug-Patching for Mozilla's *Firefox*

Jean-Michel Dalle, Matthijs den Besten, Héla Masmoudi and Paul David

The case study of the Mozilla project is focused on the organization of quality control and quality assurance in a distributed innovation environment, and focuses on the coordination of the detection and correction of operating defects ("bugs") in Mozilla's Firefox web-browser. Reductions of the likelihood of product malfunctions over a widely varying array of use-contexts is an important and quantifiable dimension of quality improvement in the computer software industry, and is particularly critical for the success of the characteristic distributed production mode found in projects that are developing "free" (in the sense of *libre*) and open source software systems. (It is now conventional to refer to the individuals and organisations working in this way as "FLOSS" developers, and FLOSS communities.)

Figure 1 schematically illustrates the centrality of *bug resolution* – or the sub-process of "bug-patching," as it is colloquially termed – in the generic mechanism through which software reliability is improved. The quality improvement process begins with *bug detection*, proceeds to *bug resolution*, comprising both discovery of the source of the malfunction, 'the bug', and its correction ('fix'), and finishes with the integration of the new piece(s) of code ('patches') into the next version of the software that will be released.

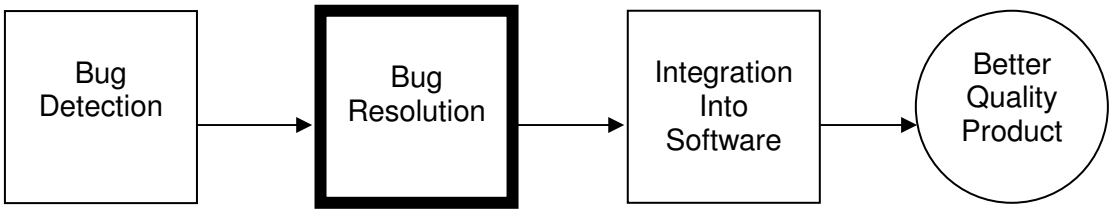

Figure 1: Bug-patching and the Software Quality Improvement Process. Source: Kuan (2003)

1 What are the key issues of interest, and how are they approached? An overview of the case study

The main goal in this study is to advance our understanding the processes of bug-patching in open source projects as one socio-technical mechanism for organizing a recurring task in "distributed problem-solving" that has now come to be adopted widely, although implemented with possibly many local variations. We have chosen to focus on the specifics of one such implementation, by examining the way in which the Mozilla community responds to bug reports about the Firefox browser.

Analytical work by economists (e.g., Bessen, 2002; Arora, Caulkins and Telang, 2005), and technical studies by software engineers (e.g., Paukins et al.,1995; Banker and Slaughter,

1997 has revealed reasons why, in the case of computer code, it will be optimal for producers to allocate resources to extensive post-release repair of defects based on feedback from user-experience. It is important to recognize that this is a feature that software does not share with the run of complex products that are constituted of physical systems, but which may extend to a wider range of information-goods that can be implemented by digital encoding. In both the economist's theoretical models, and in the engineering cost studies, however, it is typical to pre-suppose that the actually patching activity itself can be executed out with equal efficiency in the use of resource inputs (however they are supplied) when it is centrally managed prior to, or following the product's released to a distributed population of users. Empirical information could be brought to bear on that assumption by comparing the experiences of commercial software companies that experimented with both approaches, but this would entail a proprietary producer to adopt the early-and-continuous release strategy for a large software program – and has yet to be done. The existing studies, moreover, do not tell us about the practical organizational requirements and the effectiveness of *decentralised* bug-patching systems that have to mobilize and coordinate dispersed and independent resources to concurrently deal with numerous reported defects in parallel.

Therefore, what we are able to learn from this case study should be of practical interest from a variety of viewpoints:

- First of all, the participants in open source development project themselves may not be fully aware of the processes that channel their individual and collective behaviors, and those in positions to affect the procedural routines that are adopted by these project in its utilization of Bugzilla, and other, analogous bug-tracking platforms, might well benefit from our research into the micro-dynamics of resource allocation in bug-patching and it’s associated performance effects.
- Secondly, there also may be lessons in these findings for commercial software firms, and other enterprises that are beginning to move toward more "agile" code development, involving shorter release cycles for the components of digital products with modularized designs.
- Thirdly, beyond the orbit of the producers of software systems and similar, complex information products, there are the interests of the end-users. As more and more web-users adopt Firefox as their default browser, Apache web-servers, and applications that run on Linux, there is a greater need to understand the nature of the processes through which these open source programs are maintained and what features of that process and its implementation tend to gear it toward meeting their needs, rather than the needs and interests of the development communities.
- Finally, and more specifically, stakeholders in companies that have invested in supporting Firefox, or in other enterprises that directly and indirectly depend upon its reliability as a tool in the hands of users of, and contributors to web content, have an interest in appraisals of the strengths, limitations and potential "improve-ability" of the Firefox project's routines for "in-the-field -quality-assurance."

In this overview of our study it will be useful to start by underscoring the special importance of bug-patching in the context of FLOSS software production, and the broader significance of the light that an empirical approach to understanding the workings of this process in the particular case of Firefox may be able to shed on the broader phenomena of IT network-enabled distributed problem-solving. That is the subject of the following section (2) of this “brief”.

Next we describe (in section 3) the “bug-patching cycle” that is supported technically by Bugzilla, an IT platform tool that is used to coordinate communications and coordinate actions among the distributed problem-solving resources that the Mozilla community is able to mobilize.

Section 4 presents some of the issues that are involved in our study's focus on one metric of a bug-patching organisation's "problem-solving performance" -- the time-duration of the bug-resolution process cycle, and, in the case at hand, its distribution for a large sample of the operating-detect reports received from Firefox users. We note out that it would be both desirable and possible to make this metric the basis for comparative studies of performance in FLOSS, and in proprietary (closed-code) software production organisations, which, give suitable data could be carried out along the lines reported here.

Section 5 summarizes salient findings from our investigation of the way that the platform provided by Bugzilla is being used in patching bugs in Firefox, that is, the "routines" (or stabilized patterns of collective behavior) that the organization follows in responding to defect reports from users of the browser. By mining and analyzing micro-level data mined from the project's bug-tracking archive, it has been possible also to find statistically significant patterns of association between certain "response routines" and our chosen measure of effectiveness, the expected duration (or its complement, the expected speed) of the bug-resolution process. We also report on some illuminating behavioral correlates of the comparatively small category of instances in which reported bugs were successfully resolved only after an exceptionally prolonged cycle, sometimes involving repeated and inadequate patching.

Because understanding the circumstances in which distributed problem-solving mechanisms "don't work well" can be just as important as understanding why the do work, we look closely the instances where resolution of the reported bug remained elusive, and comment in the concluding section (6) on some implications of that facet of our statistical findings.

2 The "bug-patching process" and the criticality of its effectiveness for FLOSS development projects

One of the notable features of the development process that characterizes large "free and open source software" projects, and which distinguishes it from the traditional mode of production that typically is followed in proprietary (closed) software production, is the practice of "early and frequent release" of successive versions of the code. This allows each updated version of the program to be quickly accessed and used widely from an early stage in the project's development. Thus, open source projects generally do not continue to elaborate all the contemplated features of the program under development and submit it to extensive internal testing before inviting selected outsiders to use it in "beta-test" form prior to a further round of bug-fixing in order to quality the package for release to the market. The managers of commercial software companies naturally are conflicted in this process, while they recognize that a bug-ridden product is likely to be poorly received by the mass of prospective purchasers, they tend to chafe at the delays entailed in finding and fixing bugs, as this postpones the start of recouping the sunk costs incurred in the design, code-writing and code documentation stages—thereby raising the projects financing costs. Moreover, long delayed releases can risk losses of sales where competitors are able to field rival products and enjoy first-mover advantages.

The advantages of the early release policy for FLOSS projects, and its functionality in recruiting volunteer developers have been much discussed, but it is clear that a corollary of the commitment to that way of working is the likely exposure of users of the early software releases to the unwanted presence of many more defects (bugs) per line of code. Consequently, the rapidity and effectiveness with which those problems are addressed is likely to strongly influence the program's attractiveness to expert users in the first instance, and, subsequently to the perceived reliability of the product for a wider population that will

include less adept users. Furthermore, in the initial stages of the project the ability to attract and retain a growing number of user-developers with code whose reliability is “workably high” and rapidly being improved can be critically important for the formation of a developer community that will contribute to its further elaboration and maintenance.

To be sure, reliability, and user perceptions of reliability are not less important in the marketing strategy of proprietary (commercial) software producers, but their traditional organizational solution for achieving this has been to use a necessarily more limited set of employed expert programmers and selected outside developers in order to achieve an acceptable quality standard before going to the market. That internally managed approach offer advantages screening for expertise, and in scheduling and coordinating the work of paid employees. But the open source approach, by mobilizing a potentially far larger number of user-testers and bug-fixers, offers the advantages of subjecting the new code to a more heterogeneous array of user-practices and a greater diversity of use-contexts, as well as greater average speed in resolving a multiplicity of bugs by having them worked on more-or-less concurrently. This approach is likely to expose a wider assortment of bugs (including those involving incompatibilities of interoperation with other programs) to early detection, and to elicit a greater array of specialized bug-finding and patching skills.

3 Bugzilla, the “bug-tracker” its use in the bug-patching procedure

Bugzilla is one of the by-products of Netscape’s entry into the world of open source development and its transformation into the Mozilla web-browser project. Bugzilla is a bug-tracking program, a platform tool that facilitates the coordination of distributed bug discoveries (detection), problem assessments, and radiation responses in the form of patch-submissions. This program for Bugzilla was written to replace the in-house system that Netscape previously had use to handle defect reports and feature requests. The decision to put this system on-line and thus accessible to the entire web was a direct test of the famous maxim of Eric Raymond (1999) that “with enough eye-balls all bugs are shallow.” After that move, everyone who encountered a problem with the Mozilla browser could report it to Bugzilla and participate in on-line attempts to resolve the problem.

Although Bugzilla is used by all Mozilla projects, Firefox depended on it more than others. First of all, Firefox focussed on repackaging and consolidation rather than adding extra features and was less of a pure development effort. Secondly, developers like Blake Ross of Firefox constituted a new generation of programmers who had learned about the internals of Mozilla through Bugzilla whereas the first generation of Mozilla programmers had first dealt with the source code and then learnt about the bug tracker. Last, but not least, more than other Mozilla projects, Firefox was explicitly aimed at outside more mainstream users and a system like Bugzilla constitutes an important interface between developers and users.

A bug-tracking program is but a tool, and a tool by itself is not enough, because the way that it is used by the human agents also needs to be organized, or self-organized, and the “routines” followed for that purpose may involve the variation in the ways that signals received from defect-reports are filtered and translated to new signals to which bug-fixers will respond, with greater or lesser effectiveness.

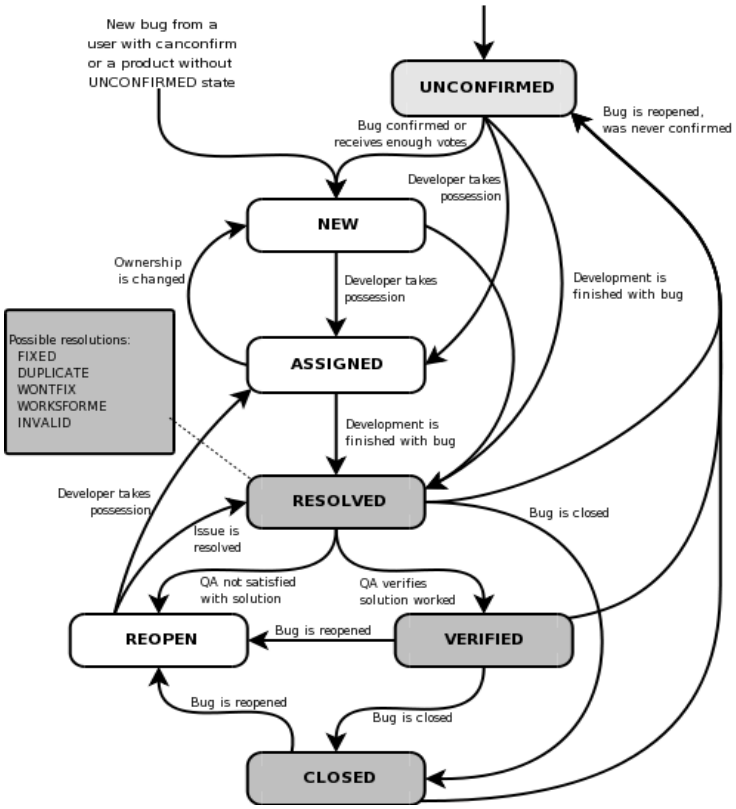

Figure 2: Flow-diagram of Defect Tracking and Bug Resolution with Bugzilla. *Source:* Bugzilla User Guide

Figure 2 is a schematic outline of the process of bug resolution that is adopted by project that employs Bugzilla. When someone reports a bug, this bug will be labelled either “unconfirmed” or (confirmed as) “new”. The criteria used in that designation are within the control of the project, and in the context of application in the Mozilla project, to be specific, if the reporter was an outsider with little or no involvement in development the bug’s initial status would be “unconfirmed”, where as the label “new” would be affixed to the bug it was reported by an insider. Having entered the process, the bug first is *triaged*. That is, the severity of the bug is assessed and a priority is assigned. In addition a developer takes possession of the problem, and deals with the report. Next, the bug is *treated*: The cause of the problem is identified and patches are proposed. Once a solution is accepted, the bug is declared “resolved” and after it has been verified that the solution (patch) works satisfactorily when it is integrated into the code base, the bug is declared “closed.”

Note that the same process that is used to track defect reports also can be used to track feature requests and issues that would appear on the project’s to-do list in general. As such, Bugzilla is one of the systems that facilitate change management. At the same time, the details of the bug resolution process different from project to project, depending on local culture and probably even the personality of the developer in charge. Note also that the whole process is archived. This allows developers to identify similar bugs and it allows analysts to track how the project as an organization learns over time.

The intricacies of these socio-technical interactions call for and also make possible a more detailed, micro-level look at the response-patterns that might systematically affect the speed of successful resolution responses to bug-reports -- our selected dimension of organizational performance. But, first, it is necessary to consider the nature of that metric for “quality improving performance.”

4 Performance metrics for quality improvement: the distribution of "bug-resolution times"

In an ideal world, we could compare the "quality" of open and closed source programs head-to-head. But quality is a multi-dimensional attribute of a product, and particular of a complex artefact such as a large software program, and to define and measure it unambiguously is not a simple matter. For instance, a survey of empirical studies by Wheeler (2003), found various dimensions of software quality – including market share, reliability (uptime), performance (speed), scalability, security, and total cost of ownership – but no agreement among the studies as to which of these dimensions really matter (Kuan 2003).

In addition, within each of the many conceivable and practical metrics that address the foregoing dimensions of software quality, there typically is more than one dimension of the quality attribute in question, and several options for their measurement. How do we choose which dimension of quality to measure, or how to weight different dimensions? Some users want easy-to-use software, while others care more about the program's speed, still others care whether it is robust in concurrent interoperation with other programs. Defining some comprehensive quality measure, which would involve a weighting of different dimension is therefore a formidable challenge, and even finding a metric for a single quality dimension is difficult.

But it is possible to avoid having to define and measure levels of quality and still arrive at an assessment organisational performance in effecting software *quality improvements.* This trick involves choosing as a performance metric the comparative speeds at which problems for users that arise from defects (bugs) in a given body of code can be successfully resolved. The distribution of the lengths of time that the remediation process takes is obviously of interest for users of the program, and it subsumes the frequency with which reported bugs are successfully resolved.

Comparison of rates of improvement for similar software programs where different bug-patching procedures were utilized would be feasible using this measure, and these would be quite illuminating even if a static measure of quality could be defined. Such statistical comparisons across software programs can and have been made that, matching open source and closed source products, as they are relatively straightforward to implement if one is able to obtain access to records of "service requests" and their disposition by the organisation involved, or "bug-tracking" data.

Such a comparison is illustrated by Figure 3 which compares the baseline hazard rates for successful bug-patching in the cases of an open source and a closed source (commercial) web-server program. The figure is reproduced from Kuan's (2003) pioneering empirical study, which statistically fitted hazard functions to bug-tracking data obtained from each of the production organisations. The "hazard" here represents the estimated likelihood (on the vertical-axis) of a reported bug being fixed at elapsed time t , being measured from the date of the first report, conditional it not having been fixed prior to that date. The "baseline hazards" are have been estimated by regression methods using "controls" for the level of "priority rating," and the level of "severity" of the defect that was assigned on the basis of the reports received by users. The baseline hazard for the case of the Apache (open source) web server is shown by the dark heavy line, which lies above that for the proprietary closed source program. This indicates that, for bugs of a given priority and severity, the resolution

process was faster for the open source program than for its closed source counterpart.

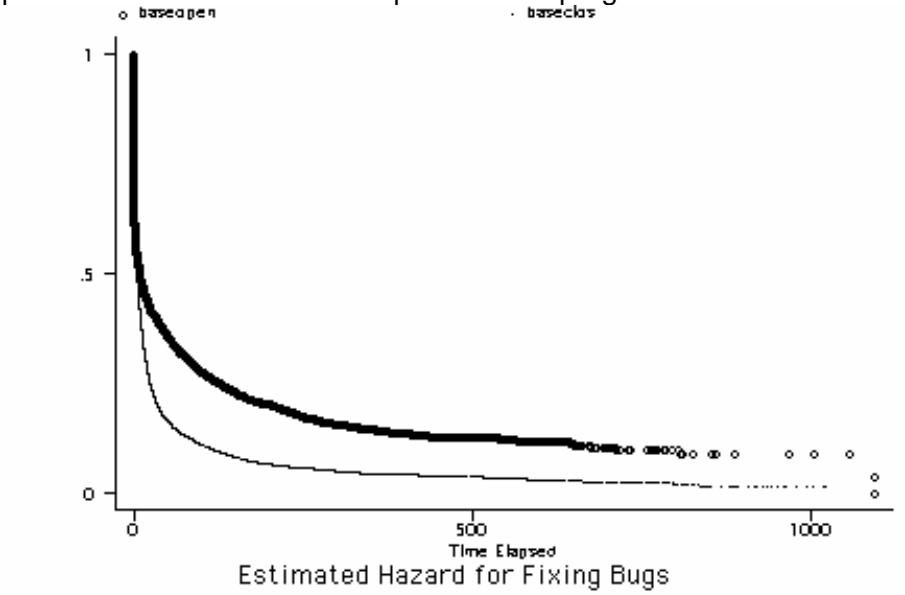

Figure 3: Baseline Hazard Rates for Open and Closed Source Web Servers. Source: Kuan (2003)

But, that is by no means the general state of affairs: for example, Kuan (2003) reports that that in a paired comparison of Gnome with a proprietary closed source graphical user-interface program, open source bug-resolution rate was consistently slower. The two kinds of development processes are really quite different, and release policies, as well as the considerations entering into the assignment of priority and severity score on the basis of bug reports cannot be assumed to be identical. In the absence detailed information about such matters, which would need to obtained through interviews with expert informants from each project, and lacking bug tracking data from proprietary software company archives, the present study has not undertake comparative performance assessments, and is focused instead on a deeper exploration of the way in which the outcomes in terms of bug survival durations (or equivalently in bug fixing speeds) are affected by the way that members of the Firefox developer community responded, in each case, to the information in the bug reports submitted to Bugzilla by users of their browser.

5 An analysis of data from the bug-patching process: methods and findings

The analysis reported here is based on two samples of bugs that we drew from the 40,000 or so that have resulted in a change to the Firefox code base. The first sample comes from the early stages of the project, when Firefox was still called Phoenix, in the interval between releases 0.1 and 0.5. The second sample relates to a later, more mature phase of the project, between release 1.5 and release 2.0. For present purposes, we ignore bugs where the severity was judged to be “enhancement”, as we want to focus on bug reports rather than feature requests.

The histogram in Figure 4 shows the frequency distribution of the survival durations (measured in block of days from the date of the initial bug report) for our entire sample of Firefox’s bugs. It is apparent from visual inspection that well more than the majority of these bugs had been “fixed” with six months of their first appearing in the system. But the tail formed by bugs that resisted successful patching is not only quite long but ample, for there

are roughly 2500 bugs, or more than 6 percent of the total, that have survived for at least 6 years. One would like to know whether those long-lived bugs were inherently difficult to find and fix, so that they should properly be thought of a having “resisted” patching, or whether they were deemed not very important because they were affecting few users, or were more of an annoyance than a source of damaging malfunctions. To look more closely into these underlying circumstances, we collected corresponding data on the course of “treatment” received by every one of the individual bugs in our sub-samples.

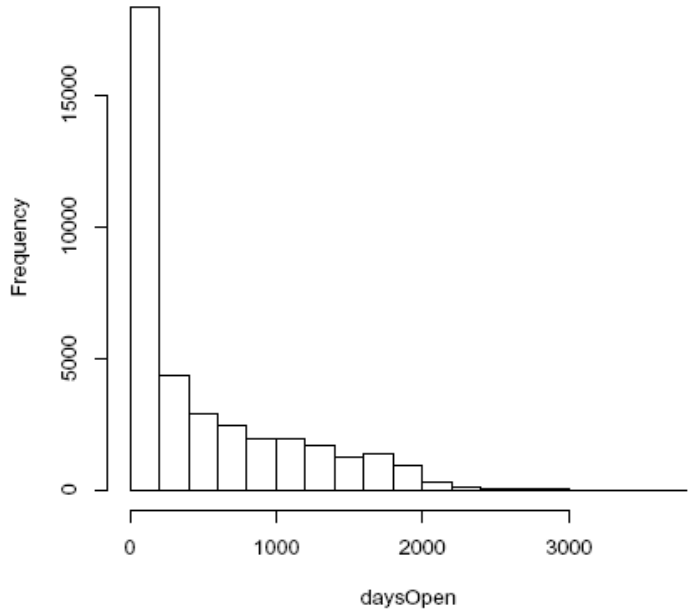

Figure 4. Histogram of the survival duration distribution of Firefox bugs reported to Bugzilla

FLOSS developers, according to Sack et al (2006), coordinate their activities almost exclusively in three information spaces: “the implementation space, the documentation space, and the discussion space.” Our data collection effort on the Firefox project’s conduct of bug-patching activities extended into each of these spaces, at least partially. It covers the implementation space in so far as we only consider bugs that are mentioned in commits to the official code base that were kept by the Mozilla concurrent versions system (CVS). For those bugs, in so far as they are associated with a file of which at least one revision is part of a Firefox release, we retrieved the bug-reports and the logs of changes to those bug reports kept by the bug tracker system Bugzilla. Thus, we are covering the documentation space as well. We also cover the discussion space since mailing-list style comments relating to the bugs are included in the reports. However, if Blake Ross(2006) is right in his assessment that Internet Relay Chat “was a big means of communication and especially with Firefox early on”, then we might have a problem here, because, to our knowledge, archives of those chats have not been kept. On the other hand, so far, we haven’t done the text mining to which the chat archives and bug comments lend themselves.

Instead, for the present, we have focused on a range of more readily quantifiable indicators. From the CVS (concurrent version system), we retrieved the following data: for each bug the number of different authors referring to the bug in their commit-comments; the number of files touched by these commits; the number of distinct comments and the number of commit; and, finally, the number of lines of code added and deleted through the commits.

From the bug-reports, we retrieved the number of persons copied in the bug resolution discussion; the number of other bugs the bug depends on or blocks; the numbers of attachments, patches and comments and the number of distinct contributors to the discussion; and, besides, the number of bugs that were identified as a duplicate of the bug

reported. In addition, we retrieved the priority assigned and the severity estimated for each bug.

With help of the change-log of the bug-reports, it was possible to retrieve the time that passed between the opening of the report and the assignment of someone in charge of the resolution process; we also retrieved the number of times the bug was (re-)assigned; whether its priority was incremented or decremented or had been changed more than once; whether its severity increased or decreased or changed more than once — where the importance was judged to be in order of trivial, enhancement, minor, normal, major, critical, blocker.

Finally, we determined whether the initial status of the bug report was New or Unconfirmed; whether the bug was reopened at least once; the number of report edits by the bug-reporter, and the number of report edits by the last person in charge of its resolution.

With this data in hand, we carried out multiple regression studies of six samples: (1) A sample of bugs related to Firefox in the early phases of its development; (2) a sample of bugs related to Firefox in a later phase of its development; (3 and 4) for both of those samples a sub-sample of bugs started their life as New; and (5 and 6) and another sub-sample for those bugs that started their life with the status: Unconfirmed.

The difference between New and Unconfirmed is due to the fact that only a subset of the members of the Firefox community have the so-called CanConfirm privilege, according to which their bugs start as New and do not need confirmation — which is precisely needed otherwise to move from Unconfirmed to New. This privilege is granted by cooptation, based on the past track record of developers asking for it, and for the sake of the analysis, we therefore consider that it characterizes members of the core of the community, as distinguished from those more peripheral members who have not been granted this privilege.

The three findings emerged most saliently and robustly in regard to the “routines” – or systematic patterns in the responses to bug reports – and their effects upon the speed of successful bug-fixing. Firstly, bug treatment behavior in the project was not temporal stable: in the early phases of Firefox was different from treatment at a later stage. In particular, the difference between the typical response to bug reports that originated from developers on the project’s periphery (“outsiders”) and the general run of bug-reports became significantly more pronounced in the more recent samples.

Second, bug reports from “outsiders” who did not have author to designate their initial status as other than Unconfirmed took longer to find a successful resolution and were more likely to remain “un-fixed”, in contrast with those that began their life on Bugzilla as Confirmed.

Thirdly, factors such as the objective technical complexity of the bug-patching problem, and the level of effort devoted to the contextualisation the reported defect play significant roles determining the speed with which a bug is typically fixed, and on balance both are associated with the further prolongation of successful resolutions.

A specialised investigation was focused on factors that might affect performance in the handling of slippery problems, i.e. of bugs that leak, and survive beyond the first patch to the main code branch that aimed to kill them and remained un-fixed for extended time periods – “superbugs”, as they might be called. It has been found that the use of specific fields of the bug management tool to orient the attention of developers to instances of patching failures could positively influence the performance of distributed development team in their handling such difficult problems and, specifically, shorten the average time before these bugs could be permanently “fixed.”

6 Closing comments: on the influence of FLOSS cultures and the problem of directing collective priorities in distributed bug patching

The insider-outside distinction here is striking in several respects. It suggests that to be on the periphery of the project means that defects one encounters when using the program are not likely to be addressed seriously and remedied, which, at least superficially fails to fit the picture that is conveyed by the "onion model" of the social interaction structure of FLOSS projects. The "inner core" of the developer community, the insiders are thought to be occupied mainly in design of the code's architecture and difficult technical question of programming approaches, with the bulk of the core occupied in code development. Bug reports and bug-fixing activities are envisaged as flowing in from the periphery where they capture the notice of core members who "take possession" and supervise the steps needed to mobilize the attentions of specialist bug-fixers. But, it seems that there has been a tendency toward the differentially strong filtering of messages from "outsiders," which carries the implication that problems affecting less technically adept users – the mass of whom are not known, or heeded by the project's core of developers – are less likely to be attended to.

Distributed problem-solving for FLOSS-reliability improvements, because it is largely self-organized and dominated by the interests and needs of the more expert developers, may then be less responsive in fixing the problems encountered by inexpert end-users of the software – even when they are caused not by the absence of helpful features, but by real defects in the code. In a commercial software vendor it would be more automatic for managers to be concerned with, and attempt to deliver more responsive bug fixing to the portions of the market where the customer base was thickest, namely among the population of less expert users. An alternative, more centralized management approach to bug-patching in FLOSS projects might be capable of emulating such a policy by according greater weight to the problems being reported from the periphery. That might be effected by altering the mechanism through which "confirmed" status was accorded to bugs on the basis of who had reported them, or by manipulating the criteria for assigning priority scores. Such a strategy, if anyone were able to acquire sufficient authority to implement it, say, by altering the allocation of "CanComfirm" privileges, would quite possibly be more effective in promoting the long-term growth of a user-base for FLOSS products. The experience of Firefox lends substance to this suggestion.

Bugzilla was used by Mozilla from the start and has been adopted by many open source projects since. The Firefox subproject also used Bugzilla, but appears to have done in a fashion that departed somewhat from previous practice. In particular, Blake Ross and other early drivers of Firefox had clear ideas as to how the bugs in Bugzilla should be triaged:

> We're making a product for mom and dad. You still have a voice here, but some of the features that you think we should add may not be the ones that they want to use. So you have to take our word for it that, even though 500 of you want something right now, you may actually be in the minority of a much larger group that we're pursuing that's going to be silent during this phase of development.

So, Ross and colleagues explicitly announced that they would ignore or downgrade some of the reports filed on Bugzilla because they feared that otherwise priorities would be skewed in the wrong direction as bugs would predominantly be filed by power-user who like "alpha-geek" features more than mom and dad would. Some of this is reflected in patterns that we were able to detect in samples of bug-reports from early and later stages in the development of Firefox as it appeared that bugs filed by outsiders were treated differently from bugs filed by insiders.

Anecdotally, the new spirit is reflected in the treatment of a bug like bug 213186. In the old spirit of hacker-humor, very early versions of Firefox had the phrase "Cookies are delicious delicacies" in the preferences panel of the browser. Bug 213186 is a request to remove this

phrase and replace it with something more appropriate. The fact that “delicious delicacies” was eventually replaced by “pieces of information stored by web pages on your computer” is a sign of the emergence of the new mom-and-dad-friendly orientation – even though it took more than a year after the request to come to that resolution. But the indications from our statistical studies that the differential treatment received by bug that accepted as “new,” vis-à-vis those reported by outsiders and hence labeled “unconfirmed,” and the association between the initial status of “unconfirmed” and longer waiting-times for a successful resolution, might be a signal that the older culture of Mozilla is reasserting itself. Whether the momentum of adoption by now has become sufficiently strong that this sort of reorientation in the bug-patching process will curtail Firefox’s ability to continue attracting a growing number of mom-and-dad users remains an open question, and one whose answer could be problematic for other FLOSS projects with similar aspirations to seriously challenge the commercial software vendors on their home turf.