Radicality and the Open Source Development Model

Christopher Oezbek
Freie Universität Berlin
Berlin, Germany
oezbek@inf.fu-berlin.de

Florian Thiel
Freie Universität Berlin
Berlin, Germany
thiel@inf.fu-berlin.de

ABSTRACT

Background: The Open Source development paradigm has matured sufficiently to give greater importance to questions on how to change the architecture and development processes of individual projects.
Objective: We explore the abilities of Open Source projects to perform radical vs. incremental changes and suggest ways to improve them.
Methods: We worked qualitatively and performed first a content analysis study of thirteen medium-scale Open Source projects. Using impression from this observational study we conducted an active case study with one large-scale Open Source project to achieve a radical change.
Results: We found evidence that Open Source projects prefer incremental changes for software design and implementation as well as for the introduction of processes and tools. Our results include preliminary explanations for this preference such as legacy constraints and structural conservatism.
Limitations: Further work is necessary to validate results widely and add a quantitative assessment to the results.
Conclusions: We conclude that project leaders and innovators need to limit the radicality of the means to the goals they want to achieve in their projects, and give some advice on how to do so.

Categories and Subject Descriptors

D.2.9 [**Software Engineering**]: Management; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*

General Terms

Management, Human Factors

Keywords

open source process, incremental, evolutionary, radical, innovation introduction

Submitted to FLOSS Workshop 2010, 1-2 July, Jena, Germany

1. INTRODUCTION

Over the last ten years our understanding of the Open Source development paradigm has increased greatly. From the licensing terms conceived by Richard Stallman [24], over the various aspects of the development process such as bug-tracking [6], lean Internet-based communication [27], distributed source code management [8], role-advancement [11] and software design [1], to the detailed study of the larger flag-ship Open Source projects such as the Apache web server [15], the Linux kernel [16], the Mozilla browser [21] and the OpenBSD operating system [12], the question on what constitutes Open Source development and how it is achieved is understood much better today.

Given this wider understanding of the Open Source paradigm as arising from loosely-coupled development by globally distributed volunteers using mostly informal means of coordination, it becomes more and more interesting to assess the implications of using the development paradigm from different perspectives. One such perspective could for instance be on security achieved in Open Source production [19] or on sustainable development over longer periods of time [4]. In this paper, we look at the ability of Open Source projects to achieve changes to their software product and development process. This perspective is of interest when considering that requirements for a particular software product and the technologies and processes used in software development are subject to rapid changes. If Open Source development is meant to be a viable mode of software production, it must be able to handle such changes just as it must be able to demonstrate its ability to produce secure software.

Our starting point was to ask if Open Source projects can deal with radical changes to their implementation, architectures and processes when prompted by disruptive changes in user needs, software landscape or available technology. To explore this question, we adopted two research methods: Firstly, we used qualitative content analysis on mailing-list data from thirteen Open Source projects which we gathered during a previous study on software development innovations in Open Source projects [17].[1]

The regarded projects were medium in size with 3-15 core members and were analyzed for their activity in the complete year 2007 for a total of 33,027 e-mails in 9,419 threads to uncover incidents in which those projects attempted re-implementations of considerable size, other radical code changes or radical changes of their processes and tool-usage.

Secondly, we performed an Action Research case study [7] with two separate Open Source projects to explore how our ideas about incremental and radical change would work in practice.

[1]Originally this data had been analyzed using Grounded Theory methodology [25], but for this analysis a less stringent method was used.

In the following text, issues regarding the software and the software development process will be kept mostly separate. Changes concerning software are commonly referred to using terms such as refactoring or reimplementation, while changes to the development process are called innovations.

With this separation in mind, this article proceeds in three steps. First, discussion is focused on re-implementations using the mailing-list data and related literature (see Section 2). Second, the case study of incrementally achieving a radical change to the code is discussed in Section 3. Third, the focus is shifted on radical changes to the development process in Section 4. Finally, we conclude with some results in section 5.

2. SOFTWARE DEVELOPMENT

The initial interest in the question about on degree of radical changes an Open Source project can achieve was raised by discussions in a sample of thirteen projects studied by one of the authors.

The maintainer of the Bugzilla project questioned the viability of continuing development using Perl as the programming language [bugzilla:6321][2]. In this discussion, which was heated and in stretches very personal [bugzilla:6394], one core-developer brought up the decisive reason against such a radical rewrite: He recounted the story of how Bugzilla had already faced the choice between rewrite and incremental repair before, when development on version 3.0 started and history had shown that the incremental repair succeeded, while the rewrite failed [bugzilla:6767].

Similar points were raised in three other projects on four other occasions by leaders or senior members of the respective projects: (1) During a discussion about the tasks to assign to students for participating in the Google Summer of Code program[3], the maintainer of the project ArgoUML praised agile methods and incremental development and cautioned against giving any task to a student, which would replace existing code rather than improve on it to achieve new functionality [argouml:4912]. The maintainer's primary argument was that a radical change would likely lead to so much additional work beyond the student's engagement to bring the replacement up to par with the existing solution that the project could not afford such at the moment. (2) In the project gEDA, the preference for incremental development was raised twice in the context of setting course for the future development of the project as the only viable way to move forward [geda:3004,3979]. The maintainer of the project mandated that all radical changes must be broken down in "a set of controlled small step refactoring stages" [geda:3016]. (3) In the project U-Boot, one developer proposed to switch to a date-based version naming scheme, because "u-boot has been around and refined for quite some time now and the changes have become a lot more incremental" rather than "earth-shattering" [uboot:31353]. This proposition was initially rejected, but eventually accepted in 2008 as this realization had spread in the project.

Taking these impressions and looking into the literature on Open Source development we can find accounts and analyses of several failures to achieve reimplementation:

Østerlie and Jaccheri give the most thorough account of such a failure when they describe the trouble the Gentoo distribution underwent when trying to re-implement their package manager Portage from 2003 to 2006 [18]. With three major attempts to replace the existing system by a more robust, better structured and faster code base within three years, yet no success on this goal, they find strong evidence that Open Source projects should prefer evolutionary over revolutionary development strategies.

In their paper the authors provide us with four reasons why the reimplementation failed: (1) There is over-indulgence in discussion which drains scarce development resources akin to a Garbage Can in organizational decision making [5]. (2) The failure to provide a prototype version that can serve as a starting point for mobilizing the community or — in the words of Eric S. Raymond — which provides a "plausible promise" that "convince[s] potential co-developers that it can be evolved into something really neat in the foreseeable future" [20]. (3) Competition for resources by (a) other reimplementation efforts and (b) the day-to-day business of dealing with bug-reports by users and attracting new developers. (4) The inability to balance the need for a stable starting point to achieve a rewrite and the need for progress on mainline development.

The authors in their concluding section condense these reasons down into the following substrate: Reimplementation is "limited by the installed base" and only possible "through a continuous negotiation with the installed base" about what is possible to say, expect and do [18]. The failure to re-implement thus is seen primarily as a failure to provide a transition strategy which takes the realities of the existing situation into account.

A similar account is given by Conor MacNeill, one of the maintainers of the project Ant, in his discussion of the history of Ant [14]. In his cases, there were multiple competing proposals for rewriting Ant as a version 2.0, none of which eventually succeeded. He argues that these proposals were typical results of second-system effect [3], in which the developers became aware of the deficiencies of their initial implementation and asked for too much in terms of new features and architectural capabilities. The project was able to resolve the splintering in different proposals eventually, but even then it proved impossible for the reimplementation to keep pace with the development of the trunk and settle the resource competition. Eventually it was accepted that incremental changes were more likely to result in the desired architecture and features [14]. Today Ant is at version 1.8.0, having achieved what was originally planned for version 2.0 [14] and much beyond, paying tribute to the realization that the way forward was an incremental one and beneficially so.

Jørgensen, as a third example, surveyed the Open Source operating system FreeBSD on their development process and found it to be highly incremental and bug-driven, while the development of radical new features appeared to be difficult [12]. Two reasons were uncovered: (1) Radical changes render the code-base of the project unusable for long times, making intermediate releases following the "release early, release often" [20] principle impossible.[4] This lack of releases has two negative consequences: First, the perception of progress in the project by users will decrease, making the project seemingly less attractive to users, which in turn will demotivate developers. Second, maintenance effort in the project will increase, as bugs need to be fixed in two (or more) diverting code bases, stretching the limited developer resources. (2) In radical feature development, the number of subtle bugs which emerge from architectural complexities rises. This makes parallel debugging break down because the number of developers who find the

[2]Citations such as [bugzilla:6321] are hyperlinks to e-mails from the respective developer mailing-list and are numbered in the order they were posted to the mailing-list archive Gmane.org.

[3]The Google Summer of Code program sponsors students to work on Open Source projects over the summer.

[4]This is also a problem for agile development and its emphasizes of no Big Design Up-Front [13] which might make more radical changes to the architecture necessary as the project progresses. Using a comprehensive test-suite as a "safety net" will probably be the reason why agile development is able to handle bigger changes with more ease [13].

"shallow" bugs (in Raymond's terms) is becoming too small [12].

Similar to this last point, Benkler makes a theoretical argument about the importance of modularity, granularity and cost of integration as limiting factors in collaborative production [2]. He argues that for Open Source to be successful in attracting participants, the independence of individual components must be maximized and the granularity of tasks for each such module and the cost for integrating them into a product must be minimized [2]. Radical features as described above are hurting granularity and exclude contributors with small scopes for activity [23].

One danger of reliance on evolutionary development is discussed by Weber based on a path dependence argument [26]: Open Source development has been able to overcome the path dependent lock-in caused by proprietary software such as Microsoft Windows, by providing a cheaper alternative and collecting contributions from users. But would Open Source projects be able to break their own paths as caused by evolutionary processes, based for instance on an inferior architecture without hierarchical control? The projects in the sample studied for this article seem to answer this question with yes by assuming a positive out-look of what can be achieved using evolutionary refactorings of even big concerns.

3. A REMEDY?

Judging by the results presented in section 2, open source software projects prefer incremental modification and also have a better chance of actually integrating changes of that nature. This observation leads to the natural question of how to turn large-scale, radical changes into manageable, evolutionary ones.

In a former case study we looked at how the development community of Open Source projects handles a proposal for an architectural change related to software security that comes with on outline describing an incremental, evolutionary path for achieving the change.

We chose the popular online publishing applications WordPress and Mambo, because we assessed via code review that the protections against SQL injections and Cross-Site-Scripting were not using known-good architectures nor libraries. To be able to benefit from secure abstraction libraries, the architecture had to be changed in a way that unified access to the SQL (for protection against SQL injection) backend and the creation of HTML output (for protection against Cross-Site Scripting).

To overcome the resistance to a radical change to existing architectures, we then devised an incremental update path: Architecturally unsound code locations would be annotated using a simple coding scheme, including a name for the issue and an estimation of the effort needed to repair this code location. A typical annotation would look like this:

```
// @RawSQLUse, trivial_implementation
```

We assessed that almost all of the annotated code locations could be fixed independently. By splitting the architectural change into manageable work packages we hoped that it would appeal to a group of developers who were interested in getting the security record improved and who could then pick individual annotated locations and fix them.

Proposing this innovation—first to the project maintainers and (after receiving positive feedback) then to the community at large—to two Open Source projects failed and provided further insight into the reasons why some types of changes might be difficult in the Open Source development paradigm. Most importantly, we found that the presence of independent third-party projects (TP) that build on another project (P) and which are important for the popularity of P prevent changes to P if they break compatibility with TP (in this case, plugins for WordPress). This is especially problematic if some TP depend directly on the inner workings of P since this makes **all** changes of the latter potentially harmful to TP.

Without a well-defined interface, the breadth of the feature-use by third-party projects was intransparent. We call the concept behind this impediment for (radical) change *Legacy Constraint* (inability to change the current state because of a dependency) caused by a *Missing Interface* (failure to define contracts for use).

We also discovered an apparent *Structural Conservatism* in one project, where (even incremental) architectural change was rejected out of fear that getting used to the new architecture would be too much of a burden for developers.

A similar *Legacy Constraint* could be observed in the abovementioned discussion in the project Bugzilla: One mailing-list participant reminded developers that many companies were selling customized versions of Bugzilla to customers with specific needs. These customized and non-public versions were reliant on incremental updates, which can be merged painlessly, and would likely fork the project if radical changes occurred [bugzilla:6354].

After so many negative examples of re-writes failing, one example should be given in which a rewrite was successful, yet only after a substantial amount of time had passed. In the example given above, the maintainer of the ArgoUML project had cautioned against assigning tasks in the Google Summer of Code to students, which aim at rewriting existing code. Yet, this advice was not followed by the core-developer who still offered to mentor the task. When a student was found to work on this task [argouml:4987], the outcome predicted by the maintainer occurred: The student was able to rewrite the existing code, but failed to get it integrated into the trunk, because legacy constraints were caused by the need to support users who were still using data based on the old implementation [argouml:5636] [argouml:7492]. From the time the Summer of Code ended in August 2007, almost 18 months passed before the new implementation was announced in a release [argouml:7905, 8209].

Why did this reimplementation eventually succeed? First, the core-developer who mentored the change put sufficient energy into seeing the change be integrated [argouml:5686,7911] and thus defeated the primary argument of the maintainer who had assumed a lack of resources. Second, an incremental update-path was chosen in which the new implementation of the feature was integrated along-side the existing one [argouml:5686] for bleeding edge users [argouml:6254]. Third, there was absolutely no concurrent development on the legacy feature [argouml:5507,4903], leaving it entirely broken [argouml:6955]. It is easy to argue that parallel development on the existing feature could likely have out-paced the reimplementation in a similar way that it had caused the reimplementation in Ant and Portage to fail. Fourth, the Google Summer of Code paid the student to work intensively on the task for three months, thus raising the student's possible task granularity to a level where the reimplementation could be achieved. Fifth and last, in stark contrast to the above case with WordPress and Mambo, sufficient interface contracts were in place so that the *Legacy Constraints* were limited to the need to support the old file format.

4. RADICAL INNOVATION

As a last step in this exploration on radical changes in Open Source projects we want to now look at changes to the development process, to used tools and methods. Such changes to the way software is developed — in contrast to changes to the product — are interesting because they might affect the capabilities of Open Source projects to produce software in an efficient and high-quality way. For instance, one could consider the introduction of a new bug-tracker into a project which can be easier understood by non-

technical users. This may raise the amount of feedback about usability issues and technical problems which are easily overcome by power-users but a large problem for non-technical users.

To approach this aspect of change capability in Open Source projects, we want to first discuss what radicality means in the context of innovation introductions. The first and most obvious defining aspect of a radical innovation introduction is the degree to which it affects changes in the project. Such changes might have an effect on:

(1) *Social Structure*: Consider as an example the innovation of self introductions, which was established in the project Bugzilla. In this innovation, new developers joining the mailing-list are asked to introduce themselves by sharing some social and occupational facts about themselves [bugzilla:6549]. This innovation is radical in the social dimension, because people are asked to reveal hitherto private information about themselves [bugzilla:6554].

(2) *Suddenness*: The suddenness by which a change is introduced in a project affects the perceived radicality of the introduction. If large changes are spread out over a long time, they are likely to be felt as incremental and evolutionary. Combining both aspects, one could say that radicality is the first derivative of change over time.

(3) and (4) *Scope* and *Effort*: The concept of radicality seems to have surprisingly only minor conceptual associations to the perceived scope of the innovation, i.e. in particular the question to whom and in which situations it might apply, and to the amount of effort associated with it [xfce:13027]. For instance, in the case of the self-introductions at Bugzilla, the criticized radicality of the innovation in social matters touching privacy [bugzilla:6554] was limited by making participation voluntary [bugzilla:6549] and giving participants control about the extent to which they want to share "some information about themselves" [bugzilla:6555]. Effort was similarly small, consisting in most cases of only a single e-mail to be written. This explains why over the observed months after the innovation was introduced, 16 participants introduced themselves to the list.

A good introductory episode to explore the concept of radicality in conjunction with innovation introductions can be found in the project GRUB. The project had long suffered from the lack of a working bug-tracker and project members were repeatedly discussing remedies. Over the course of the debate, a total of five proposals were made which can be ranked by increasing radicality: (1) Not radical at all is the proposal to stick to the status-quo of not using a bug-tracker and keeping the existing flow of bug-reports being sent to the maintainer or the mailing-list [grub:3934]. (2) The team members could restart using the existing bug-tracker [grub:4047], which was at the time of the debate filled with bugs for a legacy version of the software product. In this option the existing bugs would have to be confirmed for the current version or closed. (3) The project considered altering the existing bug-tracking software to distinguish current and legacy bugs and thus avoid the work to confirm or reject the legacy bugs [grub:3391]. (4) Some members suggested moving to an entirely novel bug-tracking system such as Bugzilla [grub:3902], which already included the capability of managing several software versions. This option would provide additional features over the existing bug-tracker. (5) It was suggested to use an integrated bug-tracker on top of a new version control system [grub:4082,3934]. Systems in this later category, such as Trac[5], simplify bug-tracking procedures by giving developers the ability to control the bug-tracker via commit-messages and enhance the bug-tracker using links to version control.

[5]`http://trac.edgewall.org/`

Considering the radicality of the propositions, the first one is of course without any radicality as it maintains the status quo. The second one can be assessed as modifying only data of the project; the third is modifying infrastructure instead by adapting the existing bug-tracker; the fourth is modifying infrastructure and data (as a migration of data is needed into the new system) and the last is modifying both data and infrastructure of the bug-tracking data and also data and infrastructure of the source code management system. While all proposals are favored at one or another point in time during the discussion, it is the least radical proposal which achieves any change that is ultimately executed by a core developer [grub:4093].

Unfortunately, the discussion in the Grub project, does not give much an indication of why deviations in certain dimensions are perceived as more radical than those in others. For instance why is modifying data perceived as less radical than the change to the software? Only the reasons for favoring the existing bug-tracker over a new one (with or without SCM integration) [grub:3273] and a rationale for not using the bug-tracker at all [grub:3934] are extensively explained by the maintainer. In the first case of using a new bug-tracker, the maintainer argues that the burden of maintaining a server over the years is unreasonable in comparison to using an existing hosting platform [grub:3273], an argument which connects radicality to long-term effort required. For the second case of not using any bug-tracker at all, the maintainer argues that bug-trackers are unsuitable for discussion in comparison to e-mail and provide no advantage over a wiki in terms of taking note of important tasks to be done [grub:3934]. This rejection is thus not connected to radicality but rather to the perceived capability and utility of solutions and in both cases rather a questionable judgment.[6]

Nevertheless, we propose the following hypothesis:

> Open Source projects will prefer innovations that incrementally improve on the existing situation in preference to radical innovations that cause major disruptions.

Such a preference of course does not mean that an incremental innovation is automatically likely to succeed or an radical innovation automatically rejected. In the project Bugzilla for instance, the maintainer proposed a highly incremental change to the development process in which developers could additionally and voluntarily ask for a design review before starting to spend time implementing a solution which might be rejected later on due to design issues [bugzilla:6943]. This introduction fails, despite being timid in the consequences it would have had and its potential to save a lot of time in cases where a solution is later rejected.

Finally, we want to inspect the innovation introductions which seemed most radical at first and in which projects switched from a centralized to a decentralized source code management system such as Git. Changes to the version control system are radical at first sight because they invalidate tool setup for all participants, potentially make new processes necessary, affect hosted software, render long-honed skills and knowledge partly obsolete, require changes to existing data and in the case of distributed version control can even affect the power relationships between project members. The last point needs some elaboration: Distributed version control systems are potentially disruptive to the power balance in the Open Source work-flow because every interested person can obtain an identical copy of the project repository and potentially supplant the existing flow of contributions coalescing into the project's

[6]It is both commonly suggested to avoid discussion in bug trackers [9] and well-known that Wikis are not very well suited for managing structured information [22].

software by social convention alone [8]. Structural power relationship as known from centralized version control, in which commit and meta-commit rights are handed out by the project-core selectively [10], can be invalidated and replaced by reputation and trust between all interested parties.

How could the projects which introduced the distributed version control system Git achieve such a radical change? The following reasons stand out:

- Execution of the migration is performed by the project leadership in person in all projects [rox:9371, geda:4322, kvm: 1399]. By executing the data migration, clean-up and server setup within very short amount of times (within one day from decision to finished execution in ROX, sixteen days in gEDA, and without any prior announcement of intent in KVM), the projects are quickly confronted with a new status quo.

- The adoption of the technology and adaptation of auxiliary processes and tools on the other hand is spread-out considerably over several months. For instance contributions were still received using the old system initially [rox:9404], tools were converted only as needed in the context of the next release [rox:9543,9434] and processes were slowly adjusted to support the new realities [uboot:25853].

- Several projects used SCM adapters to enable the use of a new technology prior to the migration [geda:2893] and use of a corresponding old technology after the fact [geda:4322]. Spreading out opportunities for learning and at the same time preserving the ability to contribute for all participants in this manner can reduce much of the radicality associated with an sudden change. It seems that some developers find their personal migration not difficult and quickly manage it [grub: 4134, geda:9424], but others struggle [geda:4464,4457].

- All projects that performed the switch in 2007 performed it partially on central components first and then only gradually finished the migration of less important parts of the project. This often duplicated maintenance effort but made the execution of the switch much more manageable for the maintainer.

- Not all projects switched to a distributed work-flow of pulling changes, but some retained centralized operation in which commit-rights are used to access one official repository [geda: 4335].

In particular, adapters and partial migrations sufficiently stretch out the introduction over time and limit their scope to take off the edge of these introductions. Furthermore, the effort required to "make the switch" only needs to be invested once in this case. The ongoing cost for the project members is zero unless the switch to a new version control system would make development processes more complicated. In addition to reducing radicality in such ways, the innovators needed in particular decisiveness (to overcome discussion) and strong capabilities in execution (to migrate data and set-up systems) to be successful.

5. CONCLUSION AND LIMITATIONS

In summary, this article has shown evidence for the preference of incremental change processes in Open Source projects and hypothesized a similar preference in the context of innovation introductions. It can be argued that the volunteer nature of Open Source participation limits the capabilities of a project to enact radical changes both in design, code, process and tools.

The discussed possibilities for reducing the radicality of code changes (e.g. change annotations) and for the radicality of innovation introductions (e.g. adapters) appear to appeal to projects' preferences for incremental solutions. Further studies are required to establish even clearer criteria for radicality and to validate the applicability of these criteria to Open Source development projects in general.

6. ACKNOWLEDGMENTS

We would like to thank Martin Gruhn, Lutz Prechelt and Stephan Salinger for advice and discussion, and the participants in the Open Source projects we studied for this article, in particular from WordPress and Mambo, who engaged with us in implementation of the proposed remedy.

7. REFERENCES

[1] F. Barcellini, F. Détienne, J.-M. Burkhardt, and W. Sack. A socio-cognitive analysis of online design discussions in an Open Source Software community. *Interacting with Computers*, 20(1):141–165, 2008.

[2] Y. Benkler. Coase's penguin, or, Linux and The Nature of the Firm. *Yale Law Review*, 112(3):369–446, Dec. 2002.

[3] F. P. Brooks. *The mythical man-month: essays on software engineering*. Addison-Wesley, Reading, MA, 1975.

[4] M. Ciolkowski and M. Soto. Towards a comprehensive approach for assessing open source projects. In *Software Process and Product Measurement*, volume 5338/2008 of *Lecture Notes in Computer Science*, pages 316–330. Springer, Berlin / Heidelberg, 2008.

[5] M. D. Cohen, J. G. March, and J. P. Olsen. A garbage can model of organizational choice. *Administrative Science Quarterly*, 17(1):1–25, 1972.

[6] K. Crowston and B. Scozzi. Bug fixing practices within Free/Libre Open Source software development teams. *Journal of Database Management*, 19(2):1–30, 2008.

[7] R. Davison, M. G. Martinsons, and N. Kock. Principles of canonical action research. *Information Systems Journal*, 14(1):65–86, Jan. 2004.

[8] B. de Alwis and J. Sillito. Why are software projects moving from centralized to decentralized version control systems? In *CHASE '09: Proceedings of the 2009 ICSE Workshop on Cooperative and Human Aspects on Software Engineering*, pages 36–39, Washington, DC, USA, 2009. IEEE Computer Society.

[9] K. Fogel. *Producing Open Source Software: How to Run a Successful Free Software Project*. O'Reilly, Sebastopol, CA, USA, 1st edition, Oct. 2005.

[10] T. J. Halloran and W. L. Scherlis. High quality and open source software practices. In J. Feller, B. Fitzgerald, F. Hecker, S. Hissam, K. Lakhani, and A. van der Hoek, editors, *Meeting Challenges and Surviving Success: The 2nd Workshop on Open Source Software Engineering*, pages 26 – 28. ACM, 2002.

[11] C. Jensen and W. Scacchi. Role migration and advancement processes in OSSD projects: A comparative case study. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 364–374, Washington, DC, USA, 2007. IEEE Computer Society.

[12] N. Jørgensen. Putting it all in the trunk: Incremental software development in the FreeBSD Open Source project. *Information Systems Journal*, 11(4):321–336, 2001.

[13] M. Lindvall, V. R. Basili, B. W. Boehm, P. Costa, K. C. Dangle, F. Shull, R. T. Tvedt, L. A. Williams, and M. V. Zelkowitz. Empirical findings in agile methods. In D. Wells and L. A. Williams, editors, *XP/Agile Universe*, volume 2418 of *Lecture Notes in Computer Science*, pages 197–207. Springer, 2002.

[14] C. MacNeill. The early history of ant development. Personal Blog. `http://codefeed.com/blog/?p=98`. Last visited 2010-03-01, Aug. 2005.

[15] A. Mockus, R. T. Fielding, and J. Herbsleb. A case study of Open Source software development: the Apache server. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 263–272, New York, NY, USA, 2000. ACM.

[16] J. Y. Moon and L. Sproull. Essence of distributed work: The case of the Linux kernel. *First Monday*, 5(11), Nov. 2000.

[17] C. Oezbek. *Introducing innovations into Open Source projects*. Doctoral thesis, Freie Universität Berlin, to appear in 2010.

[18] T. Østerlie and L. Jaccheri. Balancing technological and community interest: The case of changing a large Open Source Software system. In T. Tiainen, H. Isomäki, M. Korpela, and A. Mursu, editors, *Proc. 30th Information Systems Research Conference (IRIS'30)*, number D-2007-9 in D-Net Publications, pages 66–80, Finland, Aug. 2007. Department of Computer Sciences, University of Tampere.

[19] C. Payne. On the security of Open Source software. *Information Systems Journal*, 12(1):61–78, Feb. 2002.

[20] E. S. Raymond. *The Cathedral and the Bazaar*. O'Reilly & Associates, Sebastopol, CA, USA, 1999.

[21] C. R. Reis and R. P. de Mattos Fortes. An overview of the software engineering process and tools in the Mozilla project. In C. Gacek and B. Arief, editors, *Workshop on Open Source Software Development*, pages 155–175, Newcastle, United Kingdom, Feb. 2002. University of Newcastle upon Tyne.

[22] R. Schuster. Effizienzsteigerung freier Softwareprojekte durch Informationsmanagement. Studienarbeit, Freie Universität Berlin, Sept. 2005.

[23] C. M. Schweik, R. English, and S. Haire. Open Source software collaboration: Foundational concepts and an empirical analysis. National Center for Digital Government Working Paper Series 2, University of Massachusetts Amherst, 2008.

[24] R. M. Stallman. *Free Software, Free Society: Selected Essays of Richard M. Stallman*. GNU Press, Oct. 2002. With an introduction by Lawrence Lessig.

[25] A. L. Strauss and J. M. Corbin. *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*. SAGE, 2nd edition, Sept. 1998.

[26] S. Weber. The political economy of Open Source software. UCAIS Berkeley Roundtable on the International Economy, Working Paper Series 1011, UCAIS Berkeley Roundtable on the International Economy, UC Berkeley, June 2000.

[27] Y. Yamauchi, M. Yokozawa, T. Shinohara, and T. Ishida. Collaboration with lean media: how open-source software succeeds. In *CSCW '00: Proceedings of the 2000 ACM conference on Computer supported cooperative work*, pages 329–338, New York, NY, USA, 2000. ACM.