Collaboration, Conflict and Control

Proceedings of the 4th Workshop on Open Source Software Engineering

May 25, 2004
Edinburgh, Scotland

Editors

Joseph Feller
University College Cork
Cork, Ireland
jfeller@afis.ucc.ie

Brian Fitzgerald
University of Limerick
Limerick, Ireland
bf@ul.ie

Scott Hissam
Software Engineering Institute
Pennsylvania, USA
shissam@sei.cmu.edu

Karim Lakhani
MIT Sloan School of Management
Boston Consulting Group
Massachusetts, USA
karim.lakhani@sloan.mit.edu

Table of Contents

"Collaboration, Conflict and Control"

Welcome to the 4th Workshop on Open Source Software (OSS) Engineering. OSS is now well recognized as an important research area within both the Software Engineering and Information Systems fields, as evidenced by the growing number of related journal papers, conference papers, workshops and seminars. As part of this community of research, the *Open Source Software Engineering* workshop series has consistently attracted high quality position papers and large numbers of active participants from academia, government and industry. The position papers and presentation slides from each workshop are hosted at the *Open Source Resources* website (http://opensource.ucc.ie), and are downloaded by hundreds of readers each month.

The first workshop on open source software engineering, "Making Sense of the Bazaar", demonstrated the existence of a lively and motivated community of software engineering practitioners and researchers interested in OSS. The goal of the second workshop in the series, "Meeting Challenges and Surviving Success", was to nurture this community, and to focus the discussion on both the short and long term sustainability of the OSS phenomenon. Based on the discussion at the second workshop, "Taking Stock of the Bazaar", the third workshop, focused on the diverse array of techniques – as well as supporting tools and social contexts – which can be observed in the domain of OSS. We believed such a "stock taking" exercise would lay the foundation for more a fruitful exchange of ideas between open source software and other forms of software engineering.

The overarching theme of this year's workshop is "Collaboration, Conflict and Control." Specifically, the goal of the workshop is to bring together researchers and practitioners for the purpose of discussing the platforms and tools, the techniques and processes, and the organizational structures that are used to support and sustain:

- Communication and collaboration within and between developer and user communities.
- The resolution of conflict within development projects. This is to be addressed on many different levels, from deciding on improvements to be included in a particular release, to reconciling the goals of not-for-profit community organizations and their industrial collaborators.
- Effective leadership and control of development activities, and of the evolution of specific products. Again, this is to be addressed on many different levels, from the management of code commit privileges in a single project, to the roles played by non-governmental standards builders (e.g. the Internet Engineering Task Force and the World Wide Web Consortium) and governmental policy makers.

This year, we're pleased to include 20 position papers – from 40 authors in 12 countries – in our workshop proceedings and on the workshop website at http://opensource.ucc.ie/icse2004. We extend warm thanks to all of our authors and look forward to the best workshop yet!

Joseph Feller
Brian Fitzgerald
Scott A. Hissam
Karim R. Lakhani

How can Academic Software Research and Open Source Software Development help each other?

Vamshi Ambati
Institute for Software Research International
Carnegie Mellon University
vamshi@andrew.cmu.edu

S P Kishore
Language Technologies Institute
Carnegie Mellon University
skishore@cs.cmu.edu

Abstract

In this paper we discuss a few issues faced in coordinating, managing and implementing academic software research projects and suggest how some of these issues can be addressed by adopting tools and processes from Open Source Software Development. At the same time we also discuss how a few issues in Open Source Software Development (OSSD) projects can be addressed by adopting processes from Academic Software Research.

1. Introduction

Academic Software Research (ASR) has made substantive contributions in varying degrees for the growth of Software and Technology in diverse areas like data mining, bioinformatics, astronomy, natural language processing, medicine and others. With the wide spread of the Internet, many universities and researchers are working on collaborative projects from geographically distant areas. A typical example is the Universal Digital Library [7] project where universities and researchers from India, China, and the United States are collaborating on massive distributed sub-projects like document acquisition, digitization, quality control etc. However, characteristics of Academic Software Research like rapid prototyping, less documentation, frequent inflow and outflow of project members (e.g., students graduating, researchers changing positions) though useful and productive in collocated projects, are beginning to create problems in the academic community as the projects scale and move into geographically distributed environments. The characteristics of Academic Software Research and the issues mentioned above are essentially deviations from the conventional software engineering practices and it is worth investigating different paradigms of software development and imbibing processes and practices into the software development cycle of academic software research.

In this paper, we take the position that a few problems of communication, coordination and quality that creep into Academic Software Research as they scale and distribute geographically can be solved by adopting practices and tools from OSSD and that issues like entry barrier in the developer community for a new comer and gap between Developer and non-developer community exhibited in some OSSD projects can be addressed by adopting practices from academic software research.

This paper is organized as follows. Section 2 discusses characteristics and issues that crop up in Academic Software Research. In section 3 we discuss Open Source Software development as an emerging paradigm in Software Engineering and the issues faced by OSSD. Section 4 suggests tools and processes that can be adopted by Academic Software Research while section 5 discusses how OSSD can benefit from processes adopted from Academic Software Research. Finally, we conclude by pointing out a necessity for new tools that are needed for the benefit of both the academic and open source communities.

2. Academic Software Research (ASR)

We define Academic Software Research as a team based activity in which students or researchers in Academic institutions involve themselves in building large scale software systems with the guidance of faculty or senior researchers. These projects are mostly funded and usually managed by a single institution including various other departments in it .With the advent and extensive use of internet for software development, the scenario is changing and projects are now involving groups from globally distributed institutions. However, to make development easier these projects are always divided into sub problems and the researchers are allowed to pick the problem that they are interested in. As a result a single person or a small group gets to work on each of the sub problems identified.

Although results are promising in academic software, academicians complain that in massive distributed projects involving various academic

institutions issues of collaboration, communication and control creep in which if handled appropriately would enable the successful completion of projects in a cheaper and elegant way. Another issue that affects academic software research adversely is projects in which an active developer opts out of the project suddenly or leaves the institution after the completion of his academics. There is a lot of contextual information and assumptions involved in the project, made by that developer at various moments of time and all these are now lost and the code is the only artifact that remains. This makes the learning period of a new comer longer and sometimes the code becomes so obsolete and un-understandable that the new comer opts to redo the code himself. This introduces an unexpected delay in the project and can also cause unforeseen problems to the entire project if this group is a critical component for the project as a whole. Some of the other issues which are indirectly related to the already mentioned are:

1) Documentation in research projects is very sparse. Research involves results and achieving these results becomes the primary goal during the developmental phase and so documentation is usually done at the end of development. But research projects often don't work on a deadline. And even if they did work that way, it is very unlikely that the person who starts the project is the same as the person who finishes it. As a result, in the due course the documentation part is lost which makes things tougher at the end.
2) In Academic software research the person who works on the module is the person who knows the most about it and the bugs in it. Bug tracking is very specific to the group or developer. Other groups are only concerned about the results of the research and not how it works. As a result, when the project scales these bugs grow and lead the researcher into performing quick fixes which turn out to be hazardous to the quality and the long term goals of the project.

Methods and Tools which address these issues should evolve. A few suggestions are made in the final sections of the paper, but there is a requirement for more of them. We strongly feel that if these issues are addressed completely, academic software research can be more productive and produce a double fold of the existing outputs.

3. Open Source Software Development

The open source model of software development has gained the attention of the business, the practitioners and the research communities [4]. It is characterized as a fundamentally new way to develop software [3].

Open source developments typically have a central person or body that selects some subset of the developed code for the "official" releases and makes them widely available for distribution [1]. Tasks are not assigned; any person interested in a particular aspect of development can choose to participate and contribute to it. There is no explicit system-level design or even detailed design [3] other than in the heads of a few set of core developers. Communities for different phases of software development are formed. Every interested and motivated person joins the community that suits his level of interest and expertise. Usually the set of core developers is very small and they have the editing and commit privileges over the modules in the code. Then there are several other communities each consisting of potentially large numbers of volunteers for bug-fixes, testing, bug-reporting, documentation, release management etc. The communities are globally separated and co-ordination between them takes place in ad hoc ways [1]. Code is written with more care and creativity because developers are working only on things for which they have a real passion.

Though the quality and dependability of today's open source software projects have proved to be roughly on par with commercial developed software, we feel there are several areas where there are opportunities for improvement. For example often new developers have to understand the code and community practices and culture in order to contribute to the project. And there isn't much documentation as most of the OSS projects can not afford extensive documentation. Therefore it remains an obstacle for the initiation and infusion of new members into existing communities. Another significant issue is that the feedbacks of non-developers are not often heard on the mailing lists. This we feel is because the feedbacks from non-developers are treated as unpractical by the developers. Bringing awareness of the project and work culture and creating familiarity with tools among the huge user community seems a challenge. If it can be achieved we would see more users joining the developer community and this would result in improved and constructive interactions between the different communities.

4. OSSD helps ASR

In order to address issues in academic software research mentioned in section 2 we reverted to the two most popular Software Engineering paradigms in the present world – Proprietary and

Open Source Software development, to see if any of them have a closest possible solution.

For years now, Proprietary Software development has successfully implemented methods of enforcing opt-out rules or agreements or by producing extensive documentation to avoid issues that might arise due to persons leaving projects abruptly. Such methods clearly can not be adopted in Academic software research as no one can be bound to agreements in Academia. Also, the communication methods of Proprietary Software development are person to person communications and do not scale as size and complexity quickly overwhelm communication channels [1]. Ad hoc communication is always necessary however, as a default means of overcoming coordination problems [1].

We then turned to Open Source Software Development which bears common similarities with Academic software Research [2]. Observing the characteristics of Academic Software Research and the characteristics of OSSD mentioned in the previous sections we support this similarity and propose that academic software research can benefit the most by adopting methods and practices from OSSD.

Adopt existing tools: OSSD has created for itself a set of software engineering tools with features that fit the characteristics of open source development process [5]. It has addressed similar issues of collaboration and communication that Academic Software Research is facing through these efficient set of tools. A comprehensive list of tools that were used in most successful Open source projects and Research projects in various phases of software development is presented in table below. Also, the first step in adopting OSSD processes and methods would require the adoption of these tools [5].

Table 1 Some common tools used in OSSD

Version control	CVS, Subversion, WinCVS, ViewCVS
Issue tracking	Bugzilla, Gnats,DebBugs,Bonsai
Communication	Mailman, IRC, MajorDomo, Ezmlm
Build systems	Ant, Make,Autoconf
Design and code generation	ArgoUM ,XDoclet, Castor
Testing tools	DejaGnu, Tinderbox, JUnit,Lint, CodeStriker
Collaboration Environments	SourceForge , SourceCast

Peer reviews*:* Open source software development has benefited from "peer reviews". They have been recognized as a widely used and important quality assurance process in OSSD. A peer review is an informal review where someone other than the author, either collocated or distributed, checks the code with purpose of finding defects. It also maintains a healthy and competitive environment amongst the developers. In Academic Software Research a particular group of developers or in most cases just one developer is given the responsibility of a sub-problem and then the solution is expected of him. This would sometimes lead to group specific or person specific results. Introducing a "peer review" in Academic Software Research would result in many people to get familiar with the work done in the sub-problem and would assure quality and maintainability to the code.

Community development: In OSSD there are groups of members called "communities" for various phases of the software development. There is a small set of core developers who have the editing and commit privileges and decides what goes into the code. This is the developer community. There are other large communities for each task of bug-fixing, testing, bug-reporting, documentation, release management etc. This helps the developers to stay concentrated on their work and also enables extensive testing and efficient bug fixing. This may not be feasible in Academic Software Research but a reasonable way of implementation of the concept of a "community" in Academic Software Research could be that every sub-group working on a sub-problem could act as a community to the other sub-group.

Several projects at Carnegie Mellon University (CMU) have benefited from the open source practices and methodologies. RADAR [6] project funded by DARPA is a very good example of one such project. Interactions and communications of different groups take place over the mailing lists which not only enable ad-hoc communication but also act as a log of the interactions. Every leader of a subgroup in RADAR is also a member of one or more groups and acts as a reviewer for that group, ensuring the quality across the sub projects.

5. ASR helps OSSD

In this section let us look at how we can attempt to solve some of the issues of Open Source Software development mentioned in section 3. We feel that a significant portion of the issues mentioned arise due

to the fact that as projects scale in size like the Apache and Mozilla, it becomes extremely difficult for a new comer to join any of the non user communities. As a result there is a huge gap between the experts in that community and the new comers. We feel that the way Academic Software Research solves this issue can be helpful to the Open Source Software Development and we also feel that adopting this does not violate the characteristics of Open Source Software Development.

In Academic Software Research whenever a new comer comes into the project he looks for people who can teach him and these are often the senior researchers or faculty in that project. In OSSD there has been activity in the mailing lists which creates an asynchronous mode of learning. But since code developers are mostly involved in other activities most of the questions are left unanswered. So, bringing in the notion of a community of teaching assistants would solve this problem to a considerable extent. Among many motivations for which developers work on OSS projects fame, fun and learning are a few. A good way of learning is by teaching. The members in such a "teaching community" would be people who are in OSSD just for learning and are self-motivated and willing to teach what they have learnt. The result of growth of such a community would make the non-developers more knowledgeable and also creates a way for more users to join the developer community which is good for further progress in the project.

6. Need for new tools

We need more tools to adequately address the issues in both Academic Software Research and Open Source software development. A potential artifact that we have figured out in any software development involving research is an intermediate result. The problem of a researcher or a developer leaving the project can only be addressed in its full if we can completely know and understand these results and the various issues that the developer treaded upon while achieving them. Results indicate the progress of the project. The research community needs tools that keep track of results and the configurations of the project that lead to such results.

Although there is a wide range of tools available in OSSD, in order to support many other software engineering practices like requirements management, project management, metrics estimation, scheduling, program analysis and test suite design etc [5] we still need more of them. The open source community and the Academic Software Research community should involve in building these tools for the benefit of the two communities.

7. Conclusion

We have tried to raise issues in Academic Software Research and suggested that they can be addressed by adopting from OSSD tools for development and collaboration and processes and methodologies like "peer reviews" and community development. We also discussed issues in OSSD and suggested a "teaching community" as a possible solution for some of the issues.

8. Acknowledgements

We are indebted to the valuable suggestions and comments of Dr. Raj Reddy (CMU), Dr. William Scherlis (CMU), Dr. Anthony Tomasic (CMU) and Timothy J Halloran (CMU).

9. References

[1] Mockus, A., Fielding, R., & Herbsleb, J.D. Two Case Studies of Open Source Software Development: Apache and Mozilla (2002). *ACM Transactions on Software Engineering and Methodology, 11, 3,* pp. 309-346.

[2] Nikolai Bezroukov „Open Source Software Development as a Special Type of Academic Software Research" URL:http://www.firstmonday.dk/issues/issue4_10/bezroukov/index.html

[3] P.Vixie, "Software Engineering," in Open Sources: Voices from the Open Source Revolution, C Dibona, S Ockman and M.Stone, Eds. Sebastopol, CA:O'Reilly, 1999, pp.91-100.

[4]. Capiluppi A., Lago P., Morisio M., 2002, "***Characterizing the OSS process***", at the 2nd Workshop on Open Source Software Engineering, Int. Conf. Software Engineering, May 2002

[5] Jason E Robbins (2002) "Adopting OSSE Practices by Adopting OSS Tools". Chapter in Perspectives on Open Source and Free Software,J.Feller, B.Fitzerlad et al.

[6] The RADAR project (CMU) http://www.radar.cs.cmu.edu/

[7]The Universal Digital Library Project http://www.ulib.org/

Migrating a Development Project to Open Source Software Development

Wolf-Gideon Bleek, Matthias Finck
Department of Computer Science
University of Hamburg, Germany
{bleek, finck}@informatik.uni-hamburg.de

Abstract

The CommSy-system is a web-based community system, which has been in development since 1999 at the University of Hamburg. It has initially been developed by students and researchers in their spare time. Its last organizational setting was a publicly funded research project, which allowed for full-time and part-time developers. As that project has come to an end, we are aiming at an open source project to ensure continuity by providing a frame for people from different organizations. In this paper we discuss the characteristics of this specific project and of other open source projects to identify a strategy for migrating that particular project to open source. We outline the actions taken to migrate the existing project to open source software development and raise questions concerning the necessary characteristics of an open source project as well as whether the actions will suffice or not.

1. Introduction

Open source software is an increasingly popular form of software used as well as developed. The open source label is becoming a synonym for good quality on many levels. We feel that some of the promises of open source software development may be suitable to help sustain an existing project. In our case (study) we are discussing the conversion of our project, which started in 1999 and is now at the verge of becoming an open source project. This paper describes the setting that has been established over the years and how the transformation process has been initiated. First, we will present our understanding of an open source project and software development process and draw on essential characteristics that might help employing an open source migration. We will then describe the CommSy-project so far, followed by actions taken to migrate the existing project to an open source project. We state the goals connected with these actions and sketch further measures necessary and outline challenges as well as problems that have arisen so far or are likely to come up in the near future. We conclude with an outlook on future work and possible research questions are stated.

2. Key Characteristics of an Open Source Project

Open source projects have become very popular in recent years. Even formerly clearly commercial software is developed in open source projects these days [8], [9]. Current literature presents different views on open source software development [12], [2], [13].

Common to all open source software development projects is that the latest source code of the software is available for free to anybody. A license specifies in which way the software and the source code may be used. Enhancements or corrections to the current version are considered "contributions" to the project and are incorporated coordinately. Becoming a member of an open source project is based on the fact that one contributes actively to the project by e.g. fixing a known bug or implementing a feature presented on a "roadmap".

An open source project is usually managed by volunteers. Most of the coordination and communication work is done with electronic media, e.g. email, online discussions forums, and web-based bug-tracking systems [6], [6].

From the process point of view open source software development is regarded as agile software development [1]. This is mainly because of four reasons: First, the amount of documents produced in advance of and during development is relatively small. Code and prototypes need to speak for themselves. Second, the team structure is flat, and even though the number of participants may be large, the actual number of people working on the same area of the software is fairly small. Third, the assignment of tasks is ad-hoc (on a favor basis) and the tasks are small. Fourth, the release cycles are short (less than half a year, or even shorter than 3 months).

To characterize a software development as an open source process we feel that it must fulfill three essential characteristics:

Openness: The project must be open to new participants, e.g. new developers can get involved. Furthermore, openness means that anybody can use the product by simply installing it. It also means that the process itself is open to changes, which leads to the next point:

Agility: The development process itself must be agile in the sense that the process is carried out in short cycles and may be changed as needed.

Distributed: The participants of the development process are not all located at the same place.

3. The CommSy Project

CommSy stands for community system and is a web-based groupware system to support the communication and coordination in working and learning groups, which has been developed at the University of Hamburg.

In the last three years CommSy has been used predominantly in university contexts to support project-based learning. It supports communication (for example news and discussion forums) and the exchange of working materials (with e.g. file uploads and online documents) as well as organizing the project (aided by dates and groups). CommSy has been used in a variety of teaching fields including history, languages, education, economics, and informatics and it consists of three key features [10]:

(1) CommSy Project Rooms are designed for closed learning groups of approximately 10 to 30 members. These groups normally work for a limited period of time. To support learning group activities, CommSy offers several groupware functionalities like news or events that can be announced, discussion forums, personal homepages, where members are able to present themselves to the group and materials that can be written in a cooperative way, collected and classified by the users. A Project Room is an integral part of the CommSy Common Room.

(2) The CommSy Common Room is an archive "in progress," designed to support teaching and learning individuals and groups over a longer period of time (cf. [11]). The Common Room offers a listing of courses and extracurricular activities in current and earlier semesters, including information about contact persons, a list of Project Rooms and archived study material. Such material can be, for example, the result of a project, a research paper, a book reference, and it can be stored as file attachments.

(3) To enter either a Project- or the Common Room, CommSy offers a single entry point, the so-called CommSy Portal. In addition to its function as a convenient access to the system, the portal provides information for inexperienced users.

Starting in 1999 the CommSy system has been developed in funded and unfunded student projects and by researchers. Due to the characteristic of the development process as a student project, the process was designed to be open to any person who was interested in co-operation. Main parts of the source code were programmed with agile software development methods. For example, in 2000 the development team used the sprint technique of the extreme programming method “scrum” ([1] p. 32).

Since 2001 a publicly funded research project called WissPro[1] residing at three universities has formed the organizational frame for CommSy development and provision. The software development process has been established as an inherent part of the research project, and all members of the development team were paid by WissPro funds. The CommSy development process is based on a participatory and evolutionary design process, which is similar to the STEPS model (cf. [4], [3]).

As a consequence of modifying the organizational frame, key characteristics changed. Due to the growing team size, the stronger bonding, and the larger amount of available time for developing CommSy, the development process changed as follows:

- The openness of the development process for participants who were not involved in the WissPro-project decreased as a result of the larger programming tasks and the existence of full-time developers.
- Due to the close collaboration of the development team (located in three neighboring rooms), most design decisions were made in face-to-face meetings without (extensive) documentation.
- The agility of the process decreased and the time span of developing new releases increased because CommSy development became a paid full-time job, bigger releases were planned.

However, some characteristics of the CommSy development process stayed the same in different organizational settings:

First, there were always people with broad range of skills in the development team including social sciences and education science as well as computer science.

Second, the development process was characterized by an extensive feedback loop to ensure a co-operative learning and design process.

Third, many design decision were made in team discussions without using extensive kinds of documentation. The development team tested different techniques related to UML and tools for automatically documenting its source code, i.e. phpdoc, but for the technical support of the coordination and the documentation of the development process only CommSy itself was used in the majority of cases. The CommSy development team established

[1] WissPro stands for (German) “Wissenprojekt”, which was a publicly funded research project by the Department of Education and Research (BMBF).

a project room called CommSy-CommSy that was used to upload important documents or to fix a date for example.

As we were forced to adjust the project's setting we intended to employ an open source strategy while at the same time preserving key characteristics of our development process. Important cornerstones of our development process were the interdisciplinary team structure, the various levels of openness, and the collaborative development.

4. Actions Taken to Migrate the Project to Open Source

To better understand the migration process, we will first lay out the reasons for switching to open source. Then we point out the goals connected with our migration process. Third, we describe in detail the actions taken especially new ways of communication that were established. Furthermore, we discuss possible role changes that have occurred.

Our reasons for changing a closed-source-project to an open-source-project were located on different levels. As pointed out in section 3, the WissPro-Project constituted only a limited organizational frame due to the limited funding of the BMBF. By the end of 2003, more than 6 people lost their employment and funding for student contracts was also no longer available. People were urged to work in new jobs. However, everybody involved in the project was eager to continue developing the system. This changed the structure of the team significantly because not everybody was able to work primarily on the development of the system during work time. Furthermore, people were scattered now in different organizational frames. We therefore had to open the development process to allow for people from different organizations to participate. In addition to that, the support for the vast amount of users had to be reorganized because there was no organization left to offer a single point of reference. On top of that, the former research project demanded a sustainable solution for providing the system permanently at least at the University of Hamburg.

Besides considering other options, we have decided to follow the path of open source software development. Our expectations regarding that change are as follows:

- Sustaining the coordinated development after the end of the research project.
- Embracing all former members of the development team even though new organizational boundaries exist.
- Introducing (further) applicants to the development team, which might not be collocated.
- Enlarging the development team through publicity.
- Bundling activities from different projects concerning CommSy development.
- Ensuring quality of the development process to create quality software.
- Incorporating new requirements.

To achieve an open source process model as described in section 2, we took a set of actions. First, we evaluated a number of open source license models and chose the GPL, effective April 1st 2003, as the underlying license. Second, we moved the development platform to the freely available SourceForge network to document the change from closed source to open source even though only internal developers were active at that time.

The next steps in our migrating process were to move components of our development process from our local platform to SourceForge successively:

- Using the bug-tracker at the SourceForge platform,
- Establishing the feature request and the task tracker at the source forge platform, and
- Shifting the project documentation to SourceForge

Then we started to separate the presentation of CommSy from WissPro by focusing on the software and its status as open source software. During WissPro we always presented CommSy and its development process as a part of the WissPro project. The nearer the expiration date of WissPro came, the development team presented CommSy as a development project of its own.

All the precedent steps were taken within the organizational frame of the WissPro research project. The following steps were taken after the WissPro project had ended. One of our main goals was to open the development process for former members working in new organizational frames on one hand, and on the other hand for new members being interested in the CommSy development. The motivation opening the development process for new team members was triggered by:

- The motivation from inside the team to enlarge the number of developers and;
- Requests of users to take part in the development process.

To support the team opening process we have shifted our process communication from predominantly direct communication in meetings to communication via electronic channels. Examples of actions taken are:

- The establishing of public accessible mailing lists like commsy.discussion@lists.sourceforge.net.
- The opening of the CommSy-CommSy to all people who are interested in the CommSy development

- The reduction of personal meetings to one meeting in two months.

5. Challenges and Problems

So far, we have discovered the following areas of challenges and problems in the migration process:

- Organizational frame;
- Team structure;
- Culture, and;
- Coordination work.

On the organizational level, we now have the situation that most contributions to the project are not paid. However, some work is paid due to partial funding or people employed by the university doing the work. The challenge lies in explaining to all participants why some work is paid while other work is not. Additionally, time management can no longer rely on standard work weeks, as people's main occupation is no longer CommSy development. The new challenge is to find and communicate a new development rhythm that is fast enough to meet outside expectations and not too fast to include everybody willing to contribute.

Regarding the team structure, we can now observe that external contributors demand their role in the development team and at the same time people working relatively close can provide different levels of intensity being engaged in the development. The challenge connected to this new team structure is to integrate different paces and equally valuing their levels of contribution.

Culture is an essential part of a development project. Part of the established culture was a clearly co-operative and discursive process. Long-term members fear that by integrating an uncontrolled new number of contributors some cultural values may no longer be shared. A natural divergence is created by paid and unpaid contributions. Justification and value of unpaid work needs to be clarified. How can co-operative development and discursive evolution of design decisions be preserved, even though the number of developers increases significantly and they are distributed?

On the level of coordination work the new challenge will be communicating with a large number of associated active developers and users. The need for coordination is much higher than before and transparency about what is going on in the development process is necessary for other developers as well as users.

Additional problems are as follows:

- How are new participants integrated in the existing team? How can they learn the project culture? How do they get information about team structure and established conventions?
- The development process slows down significantly due to the required amount of coordination and communication work. How can users and developers be satisfied with a slow development progress? How can one highlight the results and progress of the development?
- How does the development team ensure that no developer will be lost in the process?
- How does the development team ensure that new people are constantly attracted to the project?

6. Future Work and Research Questions

In the future a major task will be to observe and to analyze the consequences of our actions taken to migrate the CommSy project to an open source project. Due to the fact that the actions taken in the last year until today we only have little experience to estimate the success of these actions. We can only assume that the migrating process will succeed because we have some evidence for that. On the other hand, we also have evidence for missing special characteristics of an open source process we may never (want to) achieve.

For example, the openness of our project, meaning that anybody can use the product simply by installing it and that the process itself is open to changes seems to be fulfilled, but how far the openness for new participants can be assured, is still an open question. Our current team structure and our established ways of cooperation open up the project for further participants in general, but it will still be difficult for a new member who is in a remote location from the development team to participate in the process.

An important research question seems to be how these applicants for an open source development team could be involved and be integrated. An evaluation of some typical open source projects that are established for a longer period of time based on this question will be part of our future work. The results of this evaluation can be used to promote our migration process.

Another research question is how one can systematically migrate an existing software development process to an open source process. It seems to be an important part of our future work to analyze the actions we have taken to support the migration process to find out which are an appropriate instrument and which are not. We have to analyze in how far our project settings differ from other project settings and which consequences we may draw for the actions taken to support the migration process.

The result should answer the question what it takes to initiate an open source project.

7. References

[1] Abrahamsson, Pekka; Salo, Outi; Ronkainen, Jussi; Warsta, Juhani 2002. VTT Electronic, Espoo. 107 p. VTT Publications: 478, ISBN 951-38-6009-4; 951-38-6010-8

[2] Feller, J. and Fitzgerald, B. *A Framework Analysis of the Open Source Software Development Paradigm*. 21st Annual International Conference on Information Systems, Brisbane, Australia, 2000.

[3] Floyd, C. A. Systematic Look at Prototyping. In: Budde, R.; Kuhlenkamp, K.; Mathiassen, L.; Züllighoven, H. (Eds.): *Approaches to Prototyping*. Springer Verlag, Berlin, Heidelberg 1984, pp. 1-18

[4] Floyd, C., Reisin, F.-M. and Schmidt, G. *STEPS to software development with users*. In ESEC '89, Lecture Notes in Computer Science, no. 387, Berlin, Heidelberg. Springer, 1989, pp. 48 - 64.

[5] Hightower, R. and Lesiecki N. *Java Tools for Extreme Programming*. New York, Wiley Computer Publishing, 2002.

[6] http://sourceforge.net/ (last visited march, 8th 2004).

[7] http://www.bugzilla.net/ (last visited march, 8th 2004).

[8] http://www.eclipse.org/legal (last visited march, 4th 2004)

[9] http://www.linux.org/info/gnu.html (last visited march, 4th, 2004)

[10] Jackewitz, I., Janneck, M. and Strauss, M. CommSy: Softwareunterstützung für Wissensprojekte. In Pape, B.; Krause, D.; Oberquelle, H. (Eds) *Wissensprojekte - Gemeinschaftliches Lernen aus didaktischer, softwaretechnischer und organisatorischer Sicht.* Münster u.a.. Waxmann Verlag, 2004.

[11] Jackewitz, I., Janneck, M., Krause, D., Pape, B. and Strauss, M. Teaching Social Informatics as a Knowledge Project. In van Weert, T. J.; Munro, R. K. (Eds.) *Informatics and the Digital Society*. Boston u.a., Kluwer Academic Publishers, 2003, pp. 261 - 268.

[12] O'Reilly, T. *Lessons from Open Source Software Development.* Communications of the ACM Vol. 42 (4): 1999, pp. 32-37.

[13] Raymond, E., S. *The cathedral and the bazaar : musings on Linux and Open Source by an accidental revolutionary*, Bejing, 1999.

[14] Scacchi, W. *Understanding the Requirements of Open Source Software Development*. IEE Proceedings on Software, 149(1), 2002, pp. 24-39.

Communication and Conflict Issues in Collaborative Software Research Projects

Cornelia Boldyreff, David Nutter and Stephen Rank
Faculty Of Applied Computing Sciences
University of Lincoln
{cboldyreff,dnutter,srank}@lincoln.ac.uk

Abstract

The Open Source Component Artefact Repository (OSCAR) was developed under the auspices of the GENESIS project to store data produced during the software development process. Significant problems were encountered during the course of the project in both the development itself and management of the project. The reasons for and potential solutions to these problems are examined with the intention of developing a set of guidelines to enable participants in other collaborative projects to avoid these pitfalls.

We wish to make it clear that we attach no opprobrium to any of the participants in the GENESIS project as many of the issues we outline below have solutions only visible with hindsight. Instead, we seek to provide a fair-minded critique of our role and the mistakes we made in a fairly typical two-year EU research project, and to provide a set of recommendations for other similar projects, in order that they can (attempt to) avoid suffering similarly.

1. Introduction

The GENESIS [7] platform is an Open Source software engineering environment designed to non-invasively complement an organisation's existing development practices. The GENESIS project initially developed all of the components that comprise the platform as a closed-source research project, however the intention from the start was to release the software as Open Source [3]. One component, developed at the University of Durham and now maintained and evolved in new projects at the University of Lincoln, is OSCAR [10, 4, 5]. This tool is intended to support the storage and retrieval of large collections of heterogeneous software artefacts. The other components of GENESIS include a work-flow management system and a project management tool, including a metrics generation and browsing tool. The consortium itself consisted of four development (academic) partners and two industrial partners.

Numerous problems occurred in each phase of the project, some of which affected the success of the OSCAR tool within GENESIS and some which have impacted upon the chances of successfully using OSCAR after the end of the GENESIS project. These problems were either common in other research projects or caused by circumstances peculiar to the GENESIS project.

Problems that are also found in other software development research projects include the effects of short timescales and rapid staff turnover. Even simple mistakes may not be rectified in time to prevent adverse effects later on in the project and the loss of access to key developers, often PhD students, fixed-term contract researchers, or students employed for short-term summer projects, and the ensuing loss of their knowledge may hamper ongoing maintenance of the software produced. Multiple re-writes of existing functionality--rather than steady evolution--are also common in research projects. Certain pieces of academic software have been re-written several times instead of being evolved from their original source code. An example of this approach is the development of the CodeWalker visualisation tool [9]. Often this approach is used because the original software cannot easily be adapted to new research needs, or the program comprehension overhead precludes re-use. Re-writes may occur for other reasons as well; for example the replacement of the Berkeley Packet Filter in OpenBSD with an alternative with a less restrictive license.

The development practices within research projects are often chaotic. For the most part, this is not a problem locally as development teams are small and thus communication overheads and conflicts within teams are minimized. When collaboration between multiple partners is required, an agreed process, even if unwritten, is necessary for effective collaboration.

2. Issues Within The GENESIS Project

A major flaw was over-ambition; the research goals of OSCAR were both lofty and difficult to achieve and evaluate successfully in a short timescale. The mere presence of lofty goals is not in itself a problem since they do ensurethat there are possibilities for future research and that these possibilities are under consideration in the earliest phases of the project. However, researchers as software developers should be very careful about the goals they seek to implement in the

context of their immediate project. Of course when applying for research funds, proposals must convince their reviewers that the proposed research will tackle difficult problems and explore innovative solutions, possibly involving speculative elements.

OSCAR also aimed to solve numerous problems experienced by our industrial partners, instead of a single problem at first and subsequent attempts to solve others. Consequently the initial requirements were very broad and complex. A concerted effort to decompose these requirements into simpler, smaller sub-requirements would have made the chosen rapid-release approach feasible. As it was, to satisfy even one requirement necessitated a significant amount of work on infrastructure, design and development before the code to address the requirement could be written. The requirements placed upon OSCAR by the other components in the system meant that significant functionality was required before OSCAR could be integrated with those components.

Finally, all parts of the GENESIS projects failed to provide suitable prototypes or mock-ups at the earliest possible phase of the project meaning the industrial partners were often unclear about what each platform component was supposed to do or how they would interact with it. Simpler individual requirements within OSCAR would have made implementing a very, very basic prototype early on in the project much easier.

While most of the features were desirable if OSCAR was to be a proper production tool, they were not necessary for a proof-of-concept and should have been relegated to later phases of the project. Following the principle of "release early, release often. . . " [11], a better approach would have been to drastically simplify the core of the initial version of OSCAR, aiming to have a very simple (even simplistic) version available as soon as possible, enabling the OSCAR team to fulfil the second half of that dictum: ". . . And listen to your customers". Good advice for all development teams is to play the expectations game; promise very little and over deliver. Doing the converse and thus disappointing one's consortium partners causes tension. Satisfying the project's reviewers may be in conflict with this strategy however, if it is not explicitly clarified.

One significant problem peculiar to the GENESIS project was the lack of decision on a common platform for development and deployment of the GENESIS environment. This issue also manifested itself in the choice of development tools by each partner. OSCAR in particular was developed exclusively in a Linux environment for much of the project in order to take advantage of the useful Free software tools available such as ready-made if simple version control in the form of CVS and accessory packages such as ViewCVS and StatCVS. The systems used by the OSCAR developers were ignored and perhaps not so readily familiar to the development partners at other institutions. Perhaps as a result of this, the attempt to set up a centralised CVS repository for the whole project failed when most developers ignored the repository and continued using their own methods and tools.

Problems of quality in the external dependencies of OSCAR were also apparent. In particular, the jCVS [8] code used to connect to a CVS pserver installation was poorly documented and difficult to use, resulting in major delays when implementing the CVS interface component and an extensive code review with associated refactoring. Instead, a simple interface class using execution calls to an external CVS binary would have performed the tasks successfully without requiring such a significant coding effort.

Compounding the quality problem, key external dependencies of OSCAR were found not to run under Windows after several months of development. These components included the CVS pserver tool and the PostgreSQL database. Despite this problem, the OSCAR code was flexible enough to be ported to Windows and MySQL very quickly, albeit at the cost of some features and significant expenditure on maintenance of the two separate database back-ends.

Conflicts also arose between the versions of certain dependencies that the GENESIS components had. For example, the OSCAR code relied on a very modern version of the XML parser tools, requiring a complex set-up process to install them where the JDK would use them. The versions required by OSCAR at one point conflicted with the requirements of other components of GENESIS, requiring a work-around.

The most important lesson here is that projects should agree on the components to use and the features they must have, including what platform they should run on if appropriate before development starts in earnest. These decisions should be taken at an early stage, even if the chosen platform is not to the liking of a minority of development partners. In the case of the GENESIS project, Windows would have been a suitable consensus platform as only the OSCAR team preferred Linux as their development environment. If at all possible use of multiple platforms should be avoided as using multiple platforms is unwieldy, especially in a demo session!

Though much interest in the OSCAR concept and our prototype was shown by potential users outside the institutions participating in the GENESIS project, our software dissemination activities were very poorly organised. Presentations at conferences often attracted potential users, andon occasion our web presence elicited interest from companies that were interested in using the system. Therefore, if attracting external users is a goal, we think that the development of an information pack including a gold CD, or at the very least a flyer, should proceed alongside the software development instead of being an activity conducted in the final stages. The information pack need not contain software initially, but a collection of all the papers, presentations and other useful

documentation would be very helpful to potential users. The web presence itself was limited, and the job of maintaining it was never really carried out. In order to attract users, it is key to show that the project is active by having a frequently-updated web presence. As academics, however, we were given credit for our publications, not for our web sites!

Finally, there was the failure to ensure sustained collaboration between partners in the consortium. The OSCAR development team was often excluded from decision making since all the other development partners were located close together in the same country, meaning they could meet frequently to discuss development issues. Much of these discussions and decisions had no impact on the OSCAR component, but this lack of communication did not foster good working relationships. The OSCAR team could only meet the other partners infrequently due to the difficulty of all the partners getting to the same location. Therefore, the bulk of the development partners were able to identify integration risks far faster and earlier than the OSCAR team managed to do. These failings are not attributable to the personnel involved: geographical distance has a deleterious effect on coordination and productivity in software engineering projects [6]. Many open source projects, however, do manage to avoid or work round these problems.

3. Resolving Conflict: Our Experiences

Within the OSCAR team the use of unit testing was extremely helpful; they fulfilled the need to test an API and provided comprehensive yet simple example programs for most parts of the system. However, relying on test-cases alone to document one's system is extremely poor practice, whatever the current XP adherents may advocate!

The introduction of elements of XP philosophy (pair programming, API agreement by test cases) and the use of code reviews were found to be effective at promoting communication. The largest subsystem that was reviewed and refactored was the CVS interface code. Other subsystems that were refactored include the Workspace management subsystem, the user session code, the graphical client and the introduction of a logging aspect throughout the OSCAR code.

Components were reselected and repurposed to address issues related to the complex and inefficient implementation code. For example, our custom data-binding code developed to transform artefacts into XML and vice-versa was replaced by the Castor framework to improve quality and flexibility. Also, the meta-data storage component was supplemented with the CVS repository so the unreliability or possible non-availability of the RDBMS would not affect a running OSCAR instance. Thus it became possible to run OSCAR without PostgreSQL if necessary.

Since continuous integration with the rest of GENESIS did not occur, an internal customer for the services of OSCAR for testing purposes was required. The CoDEEDS project needed a data storage component to contain the details of software components, design constraints and decisions taken to satisfy those constraints with the available components in a particular software system [1,2]. Though integration with CoDEEDS identified many problems with OSCAR, these problems caused delays in the development of the CoDEEDS system while its developer waited for bugs in OSCAR to be fixed. Ideally such a collaboration shouldhave begun at the inception of both projects rather than towards the end since otherwise the problem of balancing the ongoing development of each project with the need to maintain stable software or requirements for the other project to work with gave rise to conflicting demands on the OSCAR developers. Balancing the differing needs of each partner is difficult; industrial partners require dependable software to solve problems whilst academic partners wish to explore novel software solutions.

Risk management plans and scheduling are very important for academic projects since their schedules are so tightly constrained that even a single poor design decision can massively impact the outcome of the project. Once again, we advise that sub-projects such as OSCAR play the expectations game; promise very little and attempt to deliver more.

4. Conclusions

Our research development work was more or less demand-driven by the other partners implying that software would be exchanged regularly between partners. Unfortunately, release management and thus the prospect of early integration with the other components of GENESIS was poor, partially due to the conflicting configuration management philosophies of the OSCAR team and other development partners. For example, the OSCAR team prepared nightly builds with the intent that the other partners would download the nightly builds to try them each day. However, the other partners preferred a managed release program with software released in full at particular intervals with the bugs in each release fixed independently of the ongoing development effort.

Ideally consensus on the management of the release process or a conscious decision not to bother should have been achieved early in the project to the satisfaction of all parties. Additionally it is unreasonable to expect that small development teams should support more than one stable release of the software at once. The best solution is probably a compromise; a stable release should be maintained where possible and a nightly build produced to track the cutting-edge development. Additionally, the developers using OSCAR should have anticipated the need to be somewhat adventurous rather than expecting all bugs to be discovered, isolated and fixed by the OSCAR developers alone.

The choice of tools in the wider GENESIS project also caused certain conflict and collaboration problems for the

developers of OSCAR and other system components. The main tool for development was the web-based SiteScape Forum which, similar to the BSCW environment, allows documents to be uploaded, placed under version control and comments attached to them. Though useful when preparing monolithic documents or similar deliverables, this tool was not suitable for managing source code. Consequently, all the partners developed their own ways of managing the source of their components leading to conflicts and difficulty when releasing software.

Since an effort had been made to provide a single tool for all users, with some success for certain types of artefact, the lack of a suitable configuration management system was an unfortunate omission.

The development of OSCAR was hampered in the main by poor collaboration and communication within the immediate project team and the wider project. The irony of a project dedicated to creating collaborative software being unable to collaborate effectively has not escaped the authors, though the poor support offered by the SiteScape system and e-mail for collaborative code development in contrast to the effective support they offered for collaborative document development is at least partially to blame. Misguided or careless choices of implementation technology also delayed the project extensively. The ready availability of supposedly re-usable Open Source components is a double-edged sword; while they may save implementation time they may also waste it if of poor quality. Finally, the nature of short-term research projects themselves also contributed to the problems; recovering from bad decisions was that much harder as there was little free time or resources to use when repairing the damage.

The largest failing of the GENESIS project was the seeming inability to retain external users, despite a large number of enquiries into the software. In part this may be explained by the lack of maturity of the project's software, but mainly by the fact that dissemination of the software as Open Source was relegated to the end of the project; almost an afterthought. The interest, solicited and unsolicited that GENESIS recieved from users indicates that a real need for this software exists and that the project carelessly failed to take advantage of this.

Interaction with the wider community was consequently limited, even though one of the express goals of the project was to engage the Open Source community with the product. The overall project strategy may have been at fault: initial closed development was intended to "seed" external activity by providing already mature software to the community but instead made the project appear moribund much of the time. Using SourceForge or a similar site from the beginning may have helped visibility and provided better common tool support.

5. References

[1] C. Boldyreff and P. Kyaw. A framework for developing a design evolution environment. In Proceedings of the 27th Annual International Computer Software and Applications Conference (COMPSAC). IEEE, 2003. Dallas, Texas.

[2] C. Boldyreff, P. Kyaw, D. Nutter, and S. Rank. Architectural framework for a collaborative design environment. In Proceedings of Second ASERC Workshop on Software Architecture, 2003. Banff, Canada.

[3] C. Boldyreff, J. Lavery, D. Nutter, and S. Rank. Open Source development processes and tools. In J. F. et al, editor, Proceedings of the 3rd Workshop on Open Source Software Engineering, pages 1518, 2003.

[4] C. Boldyreff, D. Nutter, and S. Rank. Active artefact management for distributed software engineering. In Workshop on Cooperative Support for Distributed Software Engineerng Processes, Proceedings of COMPSAC 2002, pages 10811086. IEEE, August 2002.

[5] C. Boldyreff, D. Nutter, and S. Rank. Open source artefact management. In J. F. et al, editor, Proceedings of the 2nd Workshop on Open Source Software Engineering, pages 3 10, 2002.

[6] J. A. Espinosa, R. E. Kraut, S. A. Slaughter, J. F. Lerch, J. D. Herbsleb, and A. Mockus. Shared mental models, familiar-ity, and coordination: A multi-method study of distributed software teams. In Proceedings of the 23rd International Conference in Information Systems (ICIS), pages 425433, Barcelona, Spain, 2002.

[7] M. Gaeta and P. Ritrovato. Generalised environment for process management in cooperative software engineering. In International Computer Software and Applications Conference, volume 26, pages 10491059, Oxford, England, August 2002. IEEE.

[8] The jCVS project. http://www.jcvs.org, 2003.

[9] M. Lanza. Codecrawler: Lessons learned in building a software visualisation tool. In G. Canfora, M. van den Brand, and T. Gym'othy, editors, Proceedings of the Seventh European Conference on Software Maintenance and Reengineering, pages 409418. IEEE Computer Society, 26-28 March 2003.

[10] D. Nutter, S. Rank, and C. Boldyreff. Architectural requirements for an Open Source Component and Artefact Repository System within GENESIS. In Proceedings of the Open Source Software Development Workshop, pages 176 196. University Of Newcastle, February 2002.

[11] E. S. Raymond. The Cathedral and the Bazaar: Musing on Linux and Open Source by an accidental revolutionary. O'Reilly and Associates, 1999.

Contributing to OS Projects. A Comparison between Individual and Firms

Andrea Bonaccorsi
University of Pisa
bonaccorsi@sssup.it

Cristina Rossi
Sant'Anna School, IIT-CNR
cristina.rossi@iit.cnr.it

Abstract

This paper studies the contributions software firms make to Open Source (OS) projects. Our goal is to ascertain whether they follow the same regularity of pattern seen for individual programmer An exhaustive empirical analysis was carried out using data on project membership[1] *, project coordination and the contributions made by 146 Italian firms that do business with OS software. We compare our findings with the results of the surveys taken on OS programmers. The availability of the data gathered by Hertel et al. ([10]) on 141 developers of the Linux kernel allowed a direct comparison to be carried out between the two sets*[2] *.*

1. Introduction

The decentralized structure of the Open Source production mode makes it impossible to ascertain how many developers contribute to the movement. Agents can join the community of developers of an Open Source project freely, admission to the group only being subject to making a demonstrable contribution of some valuable code ([1]). At the same time, they can resign from the project whenever they want and for whatever reason they wish ([2]).

Open Source projects have a number of features in common with the pattern of contribution followed by individual programmers. Several surveys ([3], [4]) identified empirical regularities in the distribution of the main metrics in project contributions, i.e. project size (number of developers per project), project membership (number of projects each developer joined) and the contribution effort of each participant in terms of man/hours.Three main factors emerge which point to a highly skewed distribution of these variables. First of all, most of the development activity is concentrated on a small number of projects. Secondly, few agents take part in a large number of projects. Finally, the distribution of individual contributions within projects is spectacularly skewed. Most of the programming effort is carried out by very few programmers. This challenges the commonly held image of the Open Source community as a relatively flat network of interacting peers. Several researchers have examined this issue referring to the theory of power law distributions ([5][3] . In particular Zipf distributions were identified in project size, membership and contribution, as well as downloads, mailing list posts and CVS commits.

When a project grows in popularity it becomes more and more attractive for developers to join it. The opposite happens with unpopular projects that do not attract a large base of programmers. Individuals are more likely to expect to enhance their reputation ([7]) if they write valuable code seen by a large community, including large software firms. As a result, the dynamics of participation may share the properties of winner-take-all processes. Virtually all this evidence has been collected by looking at individual programmers. However, Open Source Software (OSS) is increasingly developed by firms that base their business on the legal and social rules of the community. They supply products under OS licenses and develop software according to the OS disclosure model. Little is known about the contribution of firms to OS projects. This paper is a contribution to fill this gap.

Following the approach used in previous surveys on Open Source developers, we prepared an on-line questionnaire; a Website was set up containing all the information on our study and the link to the data-gathering system. The relative novelty of the OSS phenomenon means that there is not yet any complete directory or list of firms involved in it and new firms are getting involved in it each year. So we approached an initial short-list of firms, asked for their help in referring to other firms active in Open Source and stopped when no new referral was generated (snowball procedure). Therefore our sample is not statistically representative of the Open Source firms' universe. However, given the

[1] *Project membership* refers to the number of projects a given firm is now joining or has joined in the past.

[2] We thank Professor Hertel for making his data available and Alessandro Scateni who built up the database

[3] Empirical regularities of this sort shape a wide range of natural and social phenomena [6].

exploratory nature of this study, this approach was considered methodologically appropriate[4] . We managed to contact 275 firms, obtaining 146 valid answers, which represents a good cross-section of the Italian firms operating in the supply-side of the Open Source market.

These firms produce and/or sell software under one or more Open Source licenses recognized by the Open Source definition. Most respondents offer services such as installation (80.1%), support (82.9%), maintenance (76%), consultancy (84.9%) and training (64.4%). 64.5% of firms surveyed began adopting the Open Source technology after 1999, while 35.5% used it before that date. Their average size is 17.3 employees (median value: 8.0), of which about 70% are software developers. Most firms are very young, 6.9% were born after 2002, 47.9% between 1998 and 2001. In general, their founders are entrepreneurs with technical background (28.8%) or with mixed technical and commercial background (65.1%). Their size, origin and business models makes them representative of a new wave of software firms that entered the market after the wording of the OS definition. Interestingly, they grow very rapidly (turnover rate of growth +121.3% over the 1999-2002 period) and have positive expectations about the potential market for Open Source solutions. On the average OSS represents 46.5% of the turnover. Although they share one of the typical features of Italian firms across all industries (namely the attitude to stay small), we do not believe their behaviour is heavily influenced by idiosyncratic national factors.

2. The activity of firms in OS projects

Table 1 shows various measures of firms' involvement in developing OS projects. Firms were asked how many projects they joined by registering on line and how many they coordinate by assuming the role of initiator or manager, in the last twelve months (year 2002) and since their initial involvement in the OS community. Further, they were asked to estimate how many Lines of Code (LOCs) they contribute as a percentage of the total size of projects, and how many patches and modules of their were accepted as official contributions. The data show that the activity of firms is quite limited. On average, they took part in fewer than 4 projects since taking their first steps in the Open Source community while the median value of projects participated in is 1. Similarly to the results of the surveys made on individual developers, all the variables observed display highly skewed distributions (table 1).

Almost half of the sample (49.6%) is not currently engaged in Open Source projects.

Variable	*Acronym*	*Min*	*Max*	*Mean*	*Std. Dev.*	*Median*	*Skewness*
No. of projects the firms joined since the very start of their OS activity	*ALL_A_PM*	0	50	3.8	7.8	1	3.5
No. of projects the firms joined last year	*C_PM*	0	20	1.6	2,8	1	3.7
No. of projects the firms coordinated since the very start of their OS activity	*ALL_A_CP*	0	28	1.1	3.4	0	5.9
No. of projects the firms coordinated during 2002	*C_CP*	0	7	0.5	1.2	0	3.5
% of Line of Codes (LOCs) the firms contributed to each project on average	*%_LOCs*	0	99	10.56	23.5	0	2.5
Contributions by the firms incorporated in the official versions of the projects	*N_C_OV*	0	300	6.9	36.9	0	6.7

Table 1. Descriptive statistics.

About 68% of the firms have participated in no more than 2 projects and many of them (46.2%) have never joined one; only 7.7% have been involved in the development of more than 10 programs (figure 1).

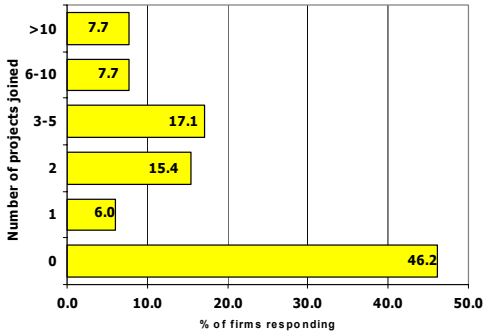


Figure 1. No. of projects the firms joined since the very start of their OSS activity

The large majority of respondents (72.9%) have never carried out coordination tasks (figure 2).

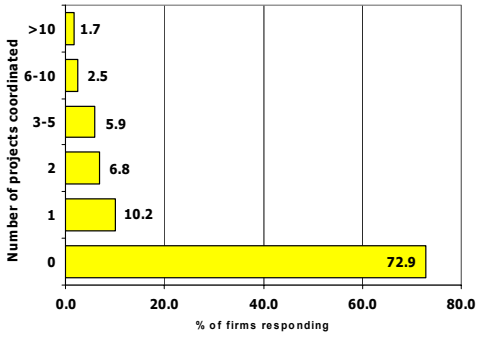


Figure 2. No. of projects the firms coordinated since the very start of their OSS activity

[4] Details on the data gathering procedures may be requested from the authors.

Only 26 firms (21.5%) are currently coordinating a project, with the majority of these (53.8%) just one. The limited involvement of respondents in Open Source projects is seen also in projects' contributions. If leading authors are classified as firms who have supplied more than 50% of a project's overall LOCs, then only 7.6% of them can be so classified (figure 3).

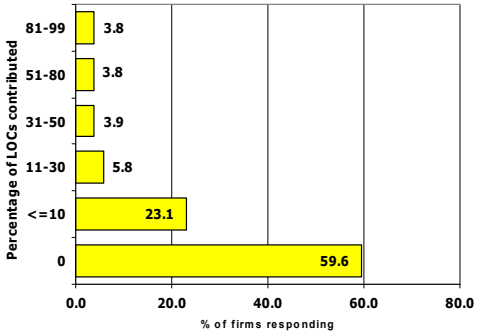


Figure 3. %_LOCs: % of LOCs the firms contributed to each project on average

Few have written pieces of code that have been accepted for the official version of a project. The mean value of variable N_C_OV is 7 but it is strongly influenced by three outliers who provided hundreds of accepted contributions[5] . This apart, the mean value becomes 1 (table 1).

In short it is possible to conclude that firms' development activities are more likely to be limited to merely adapting Open Source programs to suit their customers' requirements, giving less importance to circulating these ad hoc solutions within the community. Moreover like individual developers, firms probably prefer to join large, successful projects with a leadership already consolidated. These projects provide widely available software products, released in official and stable versions, which enable firms to provide their users with better services. Contributing to successful projects improves corporate image, giving a favourable impression to clients and potential venture capitalists. Finally, since most firms entered the OS arena just few years ago, they have not yet gained enough reputation among Open Source developers to be appointed to the coordination of a project.

We also collected data on why firms have decided to adopt a business model based on Open Source software. Each incentive is measured on a Likert scale ranging from 1 (not at all important) to 5 (very important)[6].

The number of contributions made by firms and accepted for official versions is positively correlated with the learning incentive and all the social motivations. The Open Source production mode places a huge volume of source code at the disposal of whoever wants to modify, debug or just study it. Firms that exploit this immense learning opportunity have greater chances of improving their programming skills and consequently are more likely to write pieces of code that the community will consider valuable enough to be accepted into the official versions of the programs. Moreover, interaction within projects is enhanced by positive social motivations. Firms that gift code and conform to the Open Source values have stronger linkages with the community than those who are there only to exploit new business opportunities. The former devote much more effort to OS activities and this has positive impact on the number of contributions accepted. The negative correlation between the economic motivation and the number of projects that firms are currently coordinating is also interesting. It seems that some firms exploit the innovative opportunities given by OSS without actively contributing to it. Insofar as their licensing behaviour conforms to the rules of the community, these firms seem to be accepted as legitimate members.

In order to examine closely the link between the motivations behind and level of OS activity, firms have been divided into two groups on the basis of the scores they assigned to the motivational variables, i.e. low score (1 or 2) or high score (4 or 5). In this way we may correct for a central tendency bias in reply to Likert-type scales. Table 2 summarizes the results. Only mean values displaying statistically significant differences between the two groups (Mann-Whitney test) are reported.

The various metrics of the level of activity show different patterns in the two groups, depending on the incentive type. Firms that attach much importance to purely economic considerations, have less experience in coordination activities. They are acting mainly out of extrinsic motivations such as promoting innovation or gaining freedom from the licensing and price policies of the large software companies and want to keep the

[5] Respectively 300, 200 and 100. The agent ranking fourth has only 12 accepted contributions.

[6] The taxonomy of motivations is as follows [Feller]: Economic motivations: because Open Source software allows small enterprises to afford innovation; because we want to be independent from the price and licensing policies of large software companies; because in the field of Open Source good IT specialists are easy to find; because opening our source code allows us to gain a reputation among our customers and competitors. Social motivations: because we agree with the values of the Open Source movement; because we want to place our source code and skills at the disposal of the Open Source community and hope that others will do the same; because we think that software should not to be a proprietary commodity. Technological motivations: because contributions and feedback from the Open Source community are very useful to fix bugs and improve our software; because of the reliability and quality of the Open Source software; because we want to study the code written by other programmers and use it for developing new programs and product; to obtain products not available on the proprietary software market.

cooperation link with Open Source developers active in order to obtain the feedback and contributions that allow them to lower developing costs.

Motivations	*Metrics*	*N*	*Mean*	*Dev. Std.*	*N*	*Mean*	*Dev. Std.*
		LOW SCORE			*HIGH SCORE*		
We want to be independent of the price and licensing policies of the large software companies	*C_CP**	20	0,7	1,1	76	0,4	1,4
Open Source software allows small enterprises to afford innovation	*ALL_A_CP**	13	2,0	2,4	85	0,9	3,3
We want to place our code and skills at the disposal of the OS community and hope that others do the same	*ALL_A_PM**	*28*	*3,4*	*8,0*	*61*	*4,9*	*9,2*
	N_C_OV+	*24*	*0,6*	*2,1*	*51*	*1,5*	*3,0*
We think that software should not to be a proprietary commodity	*%LOC**	*38*	*8,4*	*23,0*	*38*	*14,2*	*26,3*
	N_C_OV-	*38*	*0,4*	*1,8*	*38*	*1,8*	*3,1*
Contributions and feedback from the OS community are very useful to fix bugs and improve our software	*%LOC+*	*14*	*6,6*	*24,0*	*70*	*10,1*	*22,8*
	N_C_OV+	*15*	*0,1*	*0,3*	*65*	*1,2*	*2,7*
We want to study the code written by other programmers and use it for developing new programs	*ALL_A_PM+*	28	4,1	10,9	59	4,5	7,9

Table 2. Motivations: Mann Whitney tests. +: p value< 0.1; *p value<0.05; - p value< 0.01.

However, these firms can win the trust of developers merely by gifting their code and avoiding hijacking that written by other programmers without devoting resources to coordination tasks. The percentage of LOCs contributed and the number of accepted patches are higher for firms that attach much importance to the feedback from the community. These results corroborate the findings of the correlation analysis about the role played by social motivations in shaping the level of activity of Open Source firms. Firms that assign high scores to social incentives are more likely to win the trust of the community. As a consequence more contributions by them are accepted into official releases. Firms that value the learning opportunities provided by the Open Source mode of production very highly clearly behave in the same way. At the same time, firms that want to fight for software freedom, make great efforts to contribute LOCs to Open Source projects. In this way they increase the code base released under Open Source license schemes.

3. Project activity of firms and individual developers. A comparison.

We compare our data with the findings of the surveys on individual developers ([8], [9], [10], [11], [12], [13]). The samples of some of these empirical analyses are larger than ours. However taking into account every single person working in the firms we surveyed, it can be estimated that data account for more than 1,500 developers[7].

All the studies show that on average the membership of firms in projects is lower than that of individual developers. This holds for both the current and the overall OS activity. In general firms devote less programming effort to the projects in which they take part. Case studies of single successful projects ([9], [10]) display similarities in programming effort as measured by the percentage of LOCs contributed on average by each developer[8] . A much larger value has been instead obtained in our study. However, such comparison is not correct from a methodological viewpoint. The strictly hierarchical organization of successful projects very often makes it difficult for developers to add LOCs to their official versions. Most programmers contribute no line and this significantly affects the mean value of this metric. In the case of the firms we examined, the percentage of LOCs does not deal with a single successful project but refers to project participation in general. Together with a large group of firms contributing no LOCs (59.6%) we found firms providing almost the whole code base of the projects in which they were involved. This is likely to happen for small projects that have been started by the firms themselves and have not been able to attract a large base of developers. The comparison between our surveys and the ones taken on individual developers ([8],[12],[13]), show that firms classified as leading authors contribute a lower percentage of LOCs with respect to similarly classified developers[9] . It could be thought that this is not due to the presence of fewer firms devoting large programming efforts but to the lower concentration of their contributions. However this is not the case, given that more than 88% of the agents contribute no more than 30% of the LOCs of the projects in which they are involved. Another metric, the number of contributions included in the projects' official releases, shows a poor performance by firms. Excluding the outlier values, it appears that firms manage to get fewer contributions into the official versions of projects than individual developers do.

Proceeding in our analysis we refer to the data collected by Hertel et al. ([10]) on 141 developers of the Linux kernel. Two of the metrics collected by the authors are comparable to ours, namely the percentage of LOCs contributed and the number of contributions incorporated into official releases. However, as we explained above, methodological reasons allow to use only the latter.

[7] Our sample includes total staff at around 2,388. The ratio between programmers and total staff is in the range 60-80%, according to a telephone survey on a sub-sample carried out during the follow-up.

[8] In these three studies the percentage of LOCs contributed on average by each developer are respectively 0.1, 0.33 and 0.26.

[9] Differences in the definitions of leading author need to be taken into account.

Table 3 shows the distribution of contributions in the two groups. In comparison to individual developers, there is a higher percentage of firms in the three lowest classes (92.1% vs. 86.4%). However no significant difference emerges in the percentage of agents that do not succeed in placing contributions. The two highest classes display an interesting pattern. About 5% of the firms have a number of contributions ranging from 7 to 10, while only 1.6% of the programmers performed likewise. Nevertheless, 12% of the developers had more than 10 contributions accepted versus 4% by the firms. This sheds light on the presence of an elite of individual programmers performing very well in Open Source developing activities. Firms do not seem to include of such a group. On average firms place fewer contributions into official versions than individual developers do (4.1 vs. 8.7, Mann Whitney test, p value = 0.005).

	Linux Kernel developers (Hertel et al. 2003)		*Firms (our survey, 2003)*	
No. of contributions	*%*	*Cum. %*	*%*	*Cum. %*
0	73,6	73,6	72,3	72,3
1-3	8,8	82,4	14,9	87,2
4-6	4	86,4	4,9	92,1
7-10	1,6	88	4,0	96,0
>10	12	100	4,0	100,0

Table 3. Distributions of the accepted contributions of individual developers and firms.

3. Conclusions

These findings raise an interesting issue, namely that the level of contribution to Open Source projects seems to depend on social and technological motivations over purely economic ones.

There are firms that exploit the low cost, broad availability and good quality of Open Source software to build up a sustainable business model without contributing in the same proportion. More generally, these findings shed light on an interesting evolutionary property of the Open Source communities: robustness. The behaviour of contributing to a common pool resource does not have to be equally shared among contributors in order to be self-sustaining. Open Source communities permit some members to take more than they give, provided they do not violate minimal membership rules. By exploiting existing code more than they contribute, they still enlarge the bases of the Open Source users, indirectly enhancing the motivations of active producers. The literature on public good provision and free riding has probably overestimated the potentially damaging role of a few non-contributors, by assuming that their behaviour would inevitably self-propagate. This is not necessarily true. Nevertheless, because these firms are accepted in the Open Source community as legitimate partners, it is almost certain that, in general, they comply with the rules of membership and legal obligations for Open Source licensing schemes. That is to say, for example, that they do not hijack the code, but rather adapt and redistribute it under an appropriate license scheme. It is clear that these firms take more than they give. It seems that the new organisational mode of software production is strong enough to withstand a sharing of contributing behaviour that includes input effort no higher than the minimum entry level.

References

[1] Bonaccorsi A. and Rossi C. (2003) "Why Open Source software can succeed," Research Policy, vol. 32, No. 7, 2003, pp. 1243-1258.

[2] S. O'Mahony, "Guarding the commons: how do community managed software projects protect their work", Research Policy, vol. 32, No. 7, 2003, pp. 1179-1198.

[3] K. Healy and Schssman A., "The ecology of Open Source software development", MIT Working paper.

[4] F. Hunt and P. Johnson "On the Pareto distribution of Sourceforge projects, Proc. The Open Source Software Development Workshop, Newcastle, UK, 2002, pp. 122-129.

[5] V. Freeh, G. Madey, and R. Tynan, "Understanding OSS as a self-organizing process", Proc. 2nd Workshop on Open Source Software Engineering, IEEE CS Press, Orlando, FL, May 2002

[6] P. Krugman The self organizing economy, Backwell, New York, N.Y., 1996.

[7] J. Lerner, J. Tirole, "Some simple economics of the Open Source", The Journal of Industrial Economics, vol. 2, pp. 197-234.

[8] R. A. Ghosh, R. Glott, B. Krieger, and G. Robles, "Survey of developers, "Free/Libre and Open Source Software: Survey and Study, FLOSS Final Report, International Institute of Infonomics,,

[9] S. Koch, G. Schneider, "Effort, co-operation and co-ordination in an Open Source software project: GNOME", Information System Journal, vol. 12, 2002, pp. 27-42.

[10] G. Hertel, S. Niedner, and S. Hermann, "Motivation of software developers in the Open Source projects: an Internet-based survey of contributors to the Linux kernel" Research Policy, vol. 32, No. 7, 2003, pp. 1159-1177.

[11] J. Feller and B. Fitzgerald, Understanding Open Source Software Development, Addison Wesley, Boston, M.A., 2002.

[12] G. Robles, H. Scheider, I. Tretkowski, and N. Weber, "Who is doing it? A research on Libre Software developers", Fachgebiet für Informatik und Gesellschaft TU-Berlin

[13] J. Bates, C. Di Bona, K. Lakhani, and B. Wolf, "The Boston Consulting Group hacker survey

Improving comprehension and cooperation through code structure

Andrea Capiluppi[1]
Department of Computing and Control Systems
Politecnico di Torino, Italy
Andrea.Capiluppi@polito.it

Abstract

Defining a relationship between a software system's architecture and the process' efforts is one of the most fascinating questions of software engineering. Apparently, when a system's architecture is complex, the process to improve and evolve it will be more difficult. We try to tackle this question from a different point of view: given an Open Source system, in all the phases of its evolution, we focus on both the aspects of software developers, and the obtained software product. More we observe one of the possible architectures of this system, based on the tree structure derived from source components.

First conclusions show that some patterns of tree evolution are recognizable: some branches may appear more promising than other, and are extensively evolved, while other remain in the same status for all the life cycle. More, when the tree structure reaches some status, the process of joining as a core developer seems to forestall.

1. Introduction

Open Source software usually comes in different flavors: in some cases, software systems are claimed to be Open source as long as they release their source code under an open license. In other cases, software systems not only give unrestricted access to the source code, but details on the development process are available too (in forms of how many developers are currently writing code, the CVS access logs, and so on).

Albeit the first case has an appeal in commercial environments, to gain momentum and possibly communities feedback, open process-based OSS systems are

In this paper we use metrics-based data derived from a pilot Open Source software (OSS in the following) system, in order to understand how its internal structure evolves.

We are interested basically to observe source code structure and its changes, to learn from a long-lived OSS system what types of structural changes are more frequently brought to the source code, and to seek for patterns in those changes. For accomplishing these goals, we observe levels of source folders: with this we mean any directory in the code repository which contains source files.

2. Related work

Empirical studies on software development gained momentum after pioneer work of Lehman and his collaborators on OS/360 system [Belady & Lehman, 1976]. Lehman and Belady 's first work [Belady and Lehman, 1976] on software evolution observed 20 releases of the proprietary OS/360 operating systems. From that work, and subsequent studies of other proprietary commercial software [Lehman, 1980], [Lehman and Belady, 1985] [Lehman et al, 1987], [Lehman et al, 1988], the SPE software classification, a set of laws of *E*-type software evolution, now eight, were formulated and refined. Their findings in the seventies and eighties were revisited and supplemented in the recent FEAST projects [Lehman et al., 1998].

OSS systems have an edge over commercial ones when it comes to availability of data: many studies were done since initial efforts on the Apache web-server and Mozilla browser [Mockus et al.].

New studies are both based on individual projects [German], [Koch], [Aoki et al], [Barahona et al.], [Stamelos et al.], [Godfrey & Tu], and on several systems [Capiluppi 2003]

Interesting new approaches to study the evolution phenomenon are given through both quantitative [Antoniades et al], and qualitative [Ramil & Smith] studies on software evolution.

3. Rationale

When investigating code structure of various OSS systems, one may be faced with different patterns of modifications: if we consider code structure from the perspective of code tree (one example is depicted in Figure 1), it's possible to visualize basic components (source files, source folders) as composing a tree, the root of the tree being represented by the parent folder. When analyzing software evolution in a tree perspective, one distinguishes two dimensions:

1. *vertical growth,* that is, creating a sub-branch in an existing branch (upper part of Figure 1)
2. *horizontal growth*, that is, adding a new branch over an existing, common, branch (lower part of

1A. Capiluppi is currently visiting the Department of Maths and Computing in Open University, Milton Keynes, UK

Figure 1)

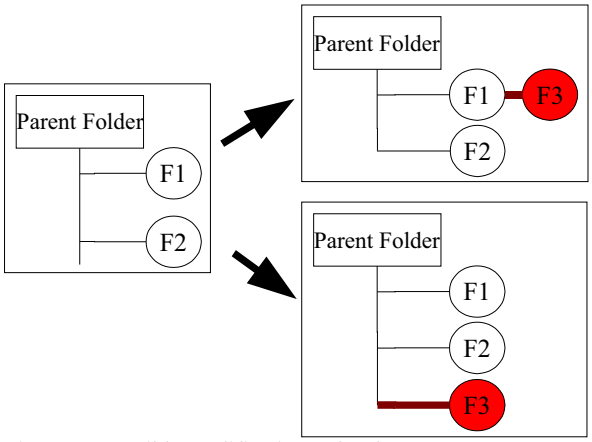


Figure 1 - Possible modifications of code structure

Our study tries to address the evolution of a particular OSS system, based on its structure and developers evolution. We are likely to investigate if there's any link between what happens from a process' perspective (how any developers join in), and the perspective of structural changes.

4. Definition of attributes

4.1. Source code

In this study we measured source code size in three different forms (LOCs, SLOCs and Kbs of code).

4.2. Code Structure

4.2.1. Code components

We deal with source code size in three different forms:

1. *source files*, as to say, all files that match some well-known source code mapping (e.g., *"*.c"*)
2. *source folders*, as to say, macro-components containing at least one source file.
3. *source levels*, as to say each level in the code structure where topologically folders may be placed (folders level)

In some sense, these components form together a code structure which may be interpreted as an architectural view of the system. Based on an open source tool for displaying graphs [Graphviz], we show the evolution of source tree, when intended as composed of files, folders and levels.

In Figure 5, Figure 6 and Figure 7, three diverse stages of evolution in the ARLA system are displayed. The first of these, namely Figure 5, we extracted the tree structure by parsing source components, and automatically feeded the GraphViz tool, in order to display the graph of the first release of this project. Each ellipse represents a source folder. Each connection defines a relationship parent-child through components: labels on connectors represent the amount of files contained in the lowermost folder (the connector arla-0.0 -> appl displays 6, because the 'appl' folder contains 6 files). In this graph, we see that the tree structure is quite defined, further evolution doesn't revolutionary the basic structure (Figure 6 and Figure 7). Since this system involves different platforms in the first instance (linux, bsd, solaris, sun etc.), we suspect that evolution may appear basically on porting new architectures, as well as pushing new functionalities. Figure 6 shows the tree structure when observing the break point observed in Figure 2. A new subtree is added to the basic tree (the red semicircle), as well as new ports for new architectures are added (red rectangles). In Figure 7 we describe the last available release at the moment: a major enhancement is depicted in leftmost red rectangle, while minor enhancements are visible in other parts of the graph. That could mean an increased difficulty in growing the system, as well as a stagnation in both feedback and new functionalities requests (as authors don't grow, in Figure 4)

4.3. Developers

What we are interested in, is OSS software development when it's started by a single person and/or a core of developers, with others getting involved later on. Some studies have published distributions of new developers as the system evolves, even though no common statistical distributions have been found yet [Barahona et al, 2002].

In order to maintain a similar notation to [Mockus, 2002], we should distinguish among developers those who contribute with regularity from those who simply gave code once. We therefore posed as *authors* those developers who submitted code more than once, and in more than a single module/file. We call as *contributors* those developers who contributed code only once, or more than once to the same module/file. Since ARLA ChangeLog provides a very regular and parseable example of keeping tracks of change, we could distinguish which change was done by what developer in which module/file.

5. Pilot Project: The ARLA System

The ARLA project made available its first public release in February 1998, and its most recent release is labeled 0.35.12 (February 2003). 35 major releases were developed, 62 total releases are made available through their web sites, which then included 27 minor releases. ARLA's main purpose is to achieve similar functionality as the IBM AFS file system. It is likely that ARLA has currently achieved even more functionalities than AFS. Its application domain is distributed file systems management, a domain in which a lot of knowledge is available and openly shared. In this respect, this system is similar to flagships OSS successes (Linux, Apache, etc.). In

ARLA's evolution there have been two basic ways of enhancing and evolving the system: adding common features for the system (e.g. supporting of specific network protocols), and adding ports to different architectures.

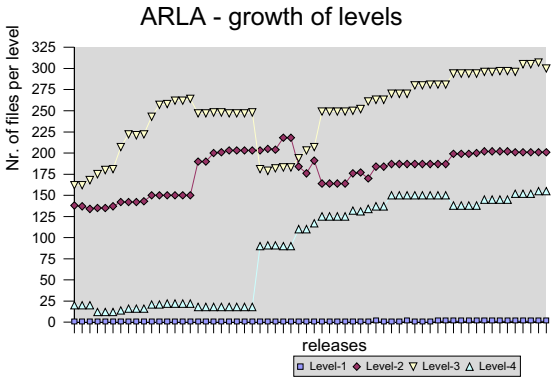


Figure 2 - ARLA levels structure

Observing its folders makeup, as measured by the number of files per folder level (Figure 2), we observe that the majority of the files have been located at Levels 2 and 3. Level 4 experimented a sudden midlife increase at around release 25. Several new folders were added on Level-4, other moved from other parts of the system, which also significantly affect Level-3.

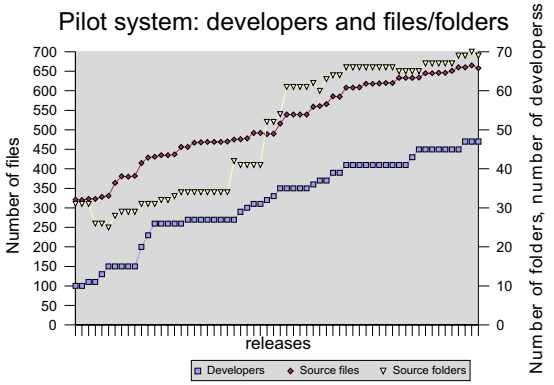
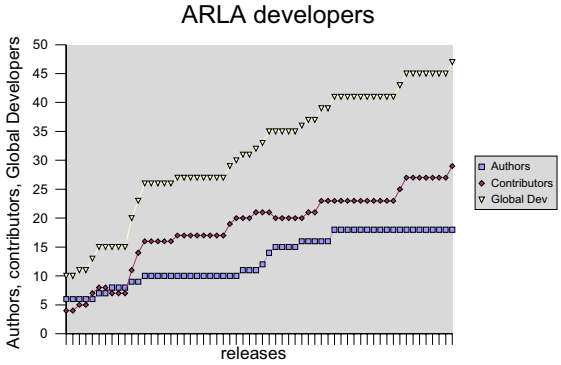


Figure 3 - Developers, Files and Folders growth

In Figure 3, we observe the makeup of evolution of ARLA as depicted by the total number of source files, and touched files over releases. The growth trend can be interpreted as two segments of decaying growth rate with a midlife growth regeneration point. The trend presents similarities with those observed in commercial systems [Ramil & Smith 2002]. The plot of files per level as in Figure 2 is particularly useful for identifying this regeneration points and Figure 2 suggest that the mid-life regeneration in growth rate was facilitated by a restructuring of the software

In Figure 3 it's displayed the trend of both the structural components (files and folders) growth, as well as the developers growth. We observe that the pattern of growth for developers resembles the folders growth, while files grow much faster. When investigating the dualism authors-contributors, we see that contributors grow faster than authors, i.e. it's easier to become a contributor than author (Figure 4).



6. Conclusions

We have deeply analyzed an OSS system from the point of view of code structure. Several subsequent releases were parsed in the perspective of both source code (LOCs, SLOCs, etc.), and source components (source files and folders). Based on this second view, we derived an integrated, graphical view of the system's structure, and we drew graphs for some interesting points we observed in components growth.

Interestingly enough, the system had a very well defined structure at the very beginning of its evolution (first available release), and it was able to evolve that structure with new functionalities. Part of this evolution is due to easily porting the existing system over a new architecture, which resolves in a similar patter over the tree structure (rightmost parts of Figure 6 and Figure 7). Part of the evolution, though, is due to new functionalities, which are either resolved in new subtrees (semicircle in Figure 6), or left as potential seeds over which connect new components (the leftmost rectangle in Figure 7 is derived from a longly lived folder, while other end-like folders are similar to their initial status).

7. References

[Aoki et al. 2002] Aoki, A., K. Hayashi, K. Kishida, K. Nakakoji, Y. Nishinaka, B. Reeves, A. Takashima, and Y. Yamamoto, 2002, "A Case Study of the Evolution of Jun: An Object-Oriented Open-Source".

[Antoniades 2003] Antoniades P., Samoladas I., Stamelos I., Bleris G.L. "Dynamical simulation models of the Open Source Development process". To appear in Free/Open Source Software Development, ed. Stefan Koch, Idea Group, Inc.

[Barahona et al 2001] Jesús M. González-Barahona, Miguel A. Ortuño Pérez, Pedro de las Heras Quirós, José

Centeno González, Vicente Matellán Olivera" Counting potatoes: The size of Debian 2.2", http://people.debian.org/~jgb/debian-counting/counting-potatoes-0.2/

[Basili et al. 1996] Basili, V. R. et al (1996), "Understanding and Predicting the Process of Software Maintenance Releases". In Proceedings of the18th International Conference on Software Engineering, Berlin, March 25-29, IEEE CS Press, Los Alamitos, CA.

[Belady et al. 1976] Belady L.A, Lehman M.M, "A Model of Large Program Development", IBM Systems J., vol. 15, no. 1, pp. 225-252, 1976.

[Capiluppi 2003] Capiluppi A., "Studying the evolution of OSS projects", on the Proceedings of the 7th International Conference on Software Maintenance, September 2003

[Capiluppi et al. 2002] Capiluppi A., Lago P., Morisio M., 2002, "Characteristics of Open Source Projects", on the Proceedings of the 7th European Conference on Software Maintenance and Reengineering, March 2003

[Chapin et al 2001] Chapin N., Hale J.E., Khan K.M., Ramil J.F. & Tan W.G. (2001), "Types of Software Evolution and Software Maintenance", Journal of Software Maintenance and Evolution: Res. and Practice, 13(1), January-February: 1 – 30

[German 2002] German D., 2002, "Using software trails to rebuild the evolution of software", 2002 ELISA Workshop, Amsterdam, The Netherlands

[Godfrey et al 2000] Godfrey, M., and Q. Tu, 2000, "Evolution in Open Source Software: A Case Study". In Proceedings of 2000 International Conference on Software Maintenance.

[Graphviz] Graphviz - open source graph drawing software http://www.research.att.com/sw/tools/graphviz/

[Kemerer et al. 1999] Kemerer, C.F., and S. Slaughter. "An Empirical Approach to Studying Software Evolution". In IEEE Transactions on Software Engineering, 1999. 25(4): 493-509.

[Koch 2000] Koch S., Schneider G., "Results from Software Engineering Research into Open Source Development Projects Using Public Data", in zum Tätigkeitsfeld Informationsverarbeitung und Informationswirtschaft, Hans R. Hansen und Wolfgang H. Janko (Hrsg.), Nr. 22, Wirtschaftsuniversität Wien, 2000.

[Lehman 1969] Lehman M.M. (1969); The Programming Process, IBM Res. Rep. RC 2722, Dec. 1969: 46 pp. Also as Chapter 3 in [Lehman & Belady 1985]

[Lehman 1974] Lehman M.M. (1974); Programs, Cities, Students, Limits to Growth?, Inaugural Lecture, in Imperial College of Science and Technology Inaugural Lecture Series, v. 9, 1970, 1974: 211 229. Also in Programming Methodology, Gries D (ed.), Springer Verlag, 1978: 42 62. Reprinted as Chapter 7 in [Lehman & Belady 1985]

[Lehman et al., 1985] Lehman M. M., Belady L. A., 1985, "Program Evolution: Processes of Software Change". Academic Press.

[Lehman et al., 1997] Lehman M.M., J.F. Ramil, P.D. Wernick, D.E. Perry, and W.M. Turski, "Metrics and Laws of Software Evolution The Nineties View,º Proc. Fourth Int'l Software Metrics Symp., Metrics '97, Albuquerque, N.M., 1997.

[Lehman, 1980] Lehman M.M, "Programs, Life Cycles, and Laws of Software Evolution", Proc. Special Issue Software Eng., IEEE, vol. 68, no. 9, pp. 1,060±1,076, 1980.

[Mockus et al. 2002] Mockus, A., Fielding, R.T., Herbsleb, J.D., 2002, "Two Case Studies of Open Source Development: Apache and Mozilla". In ACM Transactions on Software Engineering and Methodology Vol.11, No. 3, 2002, 309-346.

[Nakakoji et al. 2002] Nakakoji K., Yamamoto Y., Nishinaka Y., Kishida K.,Ye Y., 2002, "Evolution Patterns of Open-Source Software Systems and Communities". In Proceedings of International Workshop on Principles of Software Evolution (IWPSE 2002).

[Nikora & Munson 2003] Nikora A.P. & Munson J.C., "Understanding the Nature of Software Evolution", Proc. ICSM 2003, 22 – 26 Sept., Amsterdam, pp. 83 - 93

[Ramil & Smith 2003] Ramil J.F. and Smith N., "Qualitative Simulation of Models of Software Evolution", Journal of Software Process: Improvement and Practice, vol. 7, 2002, pp. 95 – 112

[Shankland 2000] Shankland S. , 2000, "Linux kernel release falls behind schedule", http://news.com.com/2100-1001-240061.html?legacy=cnet&tag=st.ne.1002.thed.1003-200-1808165

[Stamelos 2002] Stamelos, I., Angelis, L., Oikonomou, A., Bleris, G.L., "Code Quality Analysis in Open-Source Software Development", Information Systems Journal, 2nd Special Issue on OS Software, 12(1), January 2002, pp. 43-60.

[XSCC] A tool for extraction source lines of code, http://members.tripod.com/vgoenka/unixscripts/xscc.html

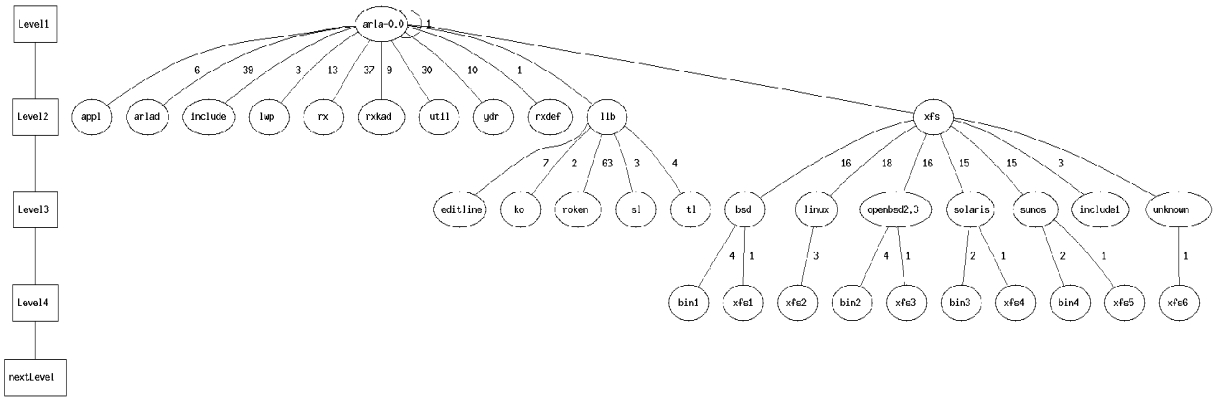


Figure 5 - Arla structure: first stage

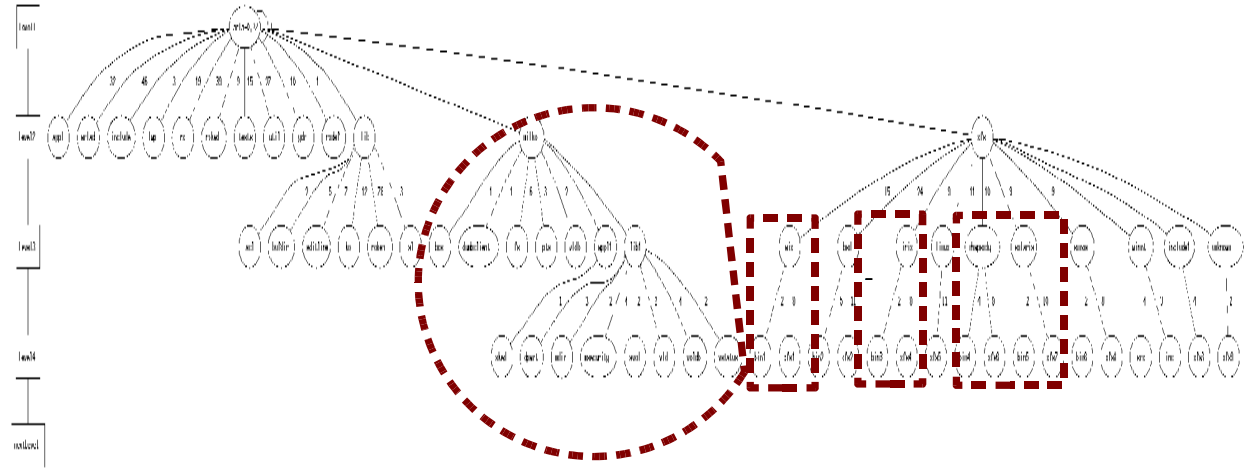


Figure 6 - Arla structure: intermediate stage (in red new branches and folders are displayed)

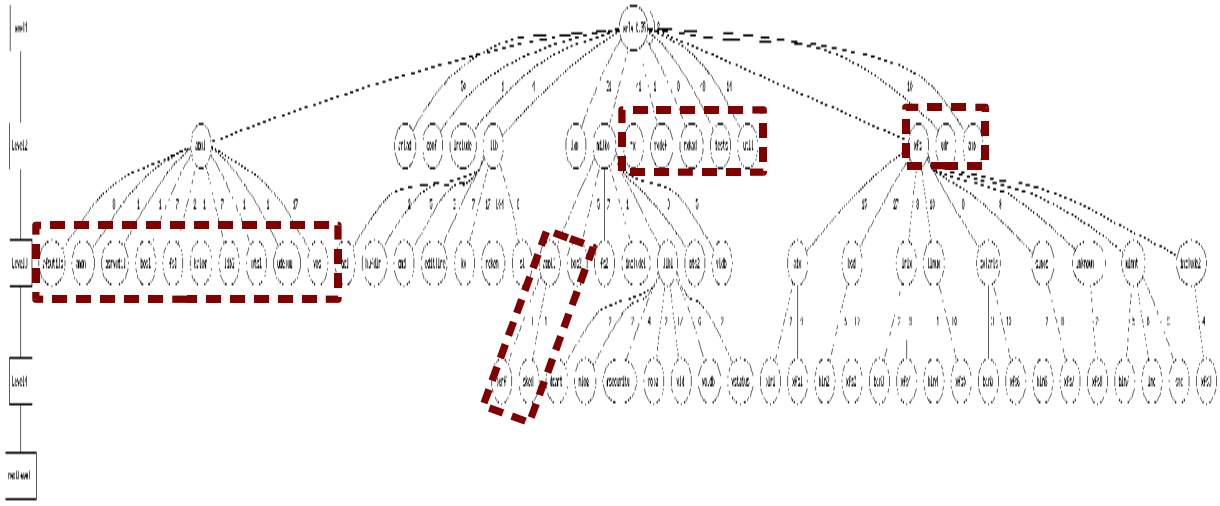


Figure 7 - ARLA structure: final stage

Towards A Portfolio of FLOSS Project Success Measures

Kevin Crowston, Hala Annabi, James Howison and Chengetai Masango
Syracuse University School of Information Studies
{crowston, hpannabi, jhowison, cmasango}@syr.edu

Abstract

Project success is one of the most widely used dependent variables in information systems research. However, conventional measures of project success are difficult to apply to Free/Libre Open Source Software projects. In this paper, we present an analysis of four measures of success applied to SourceForge projects: number of members of the extended development community, project activity, bug fixing time and number of downloads. We argue that these four measures provide different insights into the collaboration and control mechanisms of the projects.

1. Introduction

We are interested in identifying factors that predict distributed team performance, specifically, the performance of Free/Libre Open Source Software development teams on FLOSS development. In this paper, we take a first step in this direction by developing measures for the success of FLOSS projects.

It is important to develop measures of success for FLOSS projects for at least two reasons. First, having such measures should be useful for FLOSS project managers in assessing their projects. In some cases, FLOSS projects are sponsored by third parties, so measures are useful for sponsors to understand the return on their investment. Second, FLOSS is an increasingly visible and copied mode of systems development. Millions of users depend on FLOSS systems such as Linux (and the Internet, which is heavily dependent on FLOSS tools), but as Scacchi [1] notes, "little is known about how people in these communities coordinate software development across different settings, or about what software processes, work practices, and organizational contexts are necessary to their success". A recent EU/NSF workshop on priorities for FLOSS research identified the need both for learning "from open source modes of organization and production that could perhaps be applied to other areas" and for "a concerted effort on open source in itself, for itself" [2]. But to be able to learn from teams that are working well, we need to have a definition of "working well".

2. Measuring project success

Crowston, Annabi and Howison [3] examined the process of FLOSS development to suggest measures indicative of success for these projects. They noted that conventional measures of project success [e.g., 4, 5] focus primarily on use and the use environment of an information system, but for FLOSS, the use environment is generally very difficult to observe, while by contrast the development environment is quite visible. Furthermore, conventional measures define success from the perspective of corporate users who purchase the software, while FLOSS projects depend on the continued motivation of volunteer developers. Crowston, Annabi and Howison [3] therefore suggested developing a portfolio of success measures that includes measures of the development process. While they suggested a number of possible measures, they did not actually present data for any projects.

In this paper, we take the first steps towards implementing a portfolio of success measures. Specifically, we analyze four possible success measures, namely number of members of the extended development community, project activity, bug-fixing performance and number of downloads. These measures were adopted from the list presented by Crowston, Annabi and Howison [3] because they span the development process, including inputs (number of developers), process (project activity and bug fixing) and output (number of downloads).

Our analysis aims at assessing the construct validity of these measures. Each has good face validity, in the sense that a project that attracts developers, maintains a high level of activity, fixes bugs and that many users download does seem like it deserves to be described as a success. However, we are also interested in assessing how these measures relate to one another: do they measure the same construct or are they measuring different aspects of a multidimensional success construct? And most importantly, what insight do they provide into the nature of the development processes in the different projects?

In the following sections of the paper, we will discuss the procedure we used to obtain data with which to operationalize the measures, followed by the details of the analysis approach. We then present the results of this analysis and discuss the implications of these results. We conclude with some suggestions for future research.

3. Data

In this section we discuss the sample of projects selected and the process we followed to gather data about these projects.

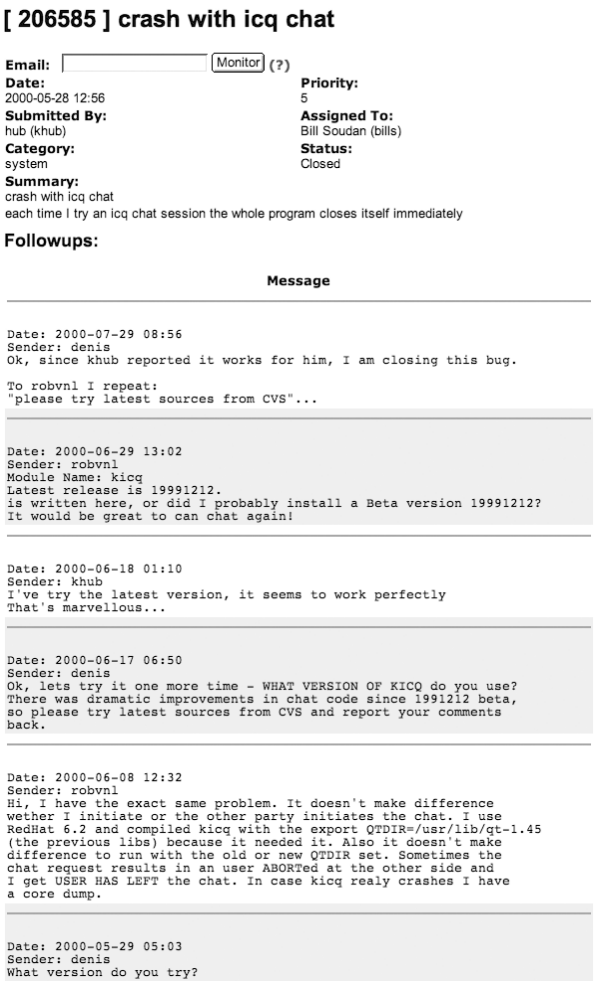
3.1. Project selection

To create a sample of FLOSS projects, we selected from projects hosted by SourceForge, a free Web-based system that provides a range of tools to facilitate FLOSS development (http://sourceforge.net/). At the time we started our study, SourceForge supported more than 50,000 FLOSS projects on a wide diversity of topics (as of 21 March 2004, the number was 78,003). Clearly not all of these projects were suitable for our study: many are inactive, previous studies have suggested that many are in fact individual projects [6], and some do not make bug reports available. Therefore, we restricted our study to projects that listed more than 7 developers and had more than 100 bugs in the project bug tracker at the time of selection in April 2002. Being listed as a developer grants write access to the project's code base, so projects with multiple developers are ones that might be expected to experience significant coordination issues. Having bug reports was a necessary prerequisite for the planned analysis, as well as indicative of a certain level of development effort. Quite surprisingly, we identified only 140 projects that met both criteria. Unfortunately, space does not permit a full listing of the projects, but the sample includes the projects *curl, fink, gaim, gimp-print, htdig, jedit, lesstif, netatalk, phpmyadmin, openrpg, squirrelmail* and *tcl*. Those familiar with FLOSS may recognize some of these projects, which span a wide range of topics and programming languages.

3.2. Data Collection

Two of the measures we included (activity level and number of downloads) are tracked by SourceForge and thus available directly from the project description page. As a measure of activity, we used the project activity rank. To gather these data, we developed a spider to download and parse the project pages. Data were collected in April 2003.

To study the performance of bug fixing, we collected data from the SourceForge bug tracking system, which



[206585] crash with icq chat

Email: Monitor (?)
Date: 2000-05-28 12:56
Priority: 5
Submitted By: hub (khub)
Assigned To: Bill Soudan (bills)
Category: system
Status: Closed
Summary: crash with icq chat

each time I try an icq chat session the whole program closes itself immediately

Followups:

Message

Date: 2000-07-29 08:56
Sender: denis
Ok, since khub reported it works for him, I am closing this bug.

To robvnl I repeat:
"please try latest sources from CVS"...

Date: 2000-06-29 13:02
Sender: robvnl
Module Name: kicq
Latest release is 19991212.
is written here, or did I probably install a Beta version 19991212?
It would be great to can chat again!

Date: 2000-06-18 01:10
Sender: khub
I've try the latest version, it seems to work perfectly
That's marvellous...

Date: 2000-06-17 06:50
Sender: denis
Ok, lets try it one more time - WHAT VERSION OF KICQ do you use?
There was dramatic improvements in chat code since 1991212 beta,
so please try latest sources from CVS and report your comments
back.

Date: 2000-06-08 12:32
Sender: robvnl
Hi, I have the exact same problem. It doesn't make difference
wether I initiate or the other party initiates the chat. I use
RedHat 6.2 and compiled kicq with the export QTDIR=/usr/lib/qt-1.45
(the previous libs) because it needed it. Also it doesn't make
difference to run with the old or new QTDIR set. Sometimes the
chat request results in an user ABORTed at the other side and
I get USER HAS LEFT the chat. In case kicq realy crashes I have
a core dump.

Date: 2000-05-29 05:03
Sender: denis
What version do you try?

Figure 1. Example bug report and followup messages (adapted from http://sourceforge.net/tracker/index.php?func= detail&aid=206585&group_id=332&atid=100332**)**

enables users to report, and developers to discuss bugs. As shown in Figure 1, a bug report includes a description of a bug that can be followed up with a trail of correspondence. Basic data for each bug includes the date and time it was reported, the reporter, priority and, for closed bugs, the date and time it was closed. To collect this data, we developed a spider program that downloaded and parsed all bug report pages for the selected projects. Unfortunately, between selection of projects and data collection, some projects restricted access to bug reports, so we were able to collect data for only 122 projects.

4. Analysis

In this section we discuss how we analyzed the data to develop the four measures introduced above. The

analysis started with basis exploration of the data, which revealed problems with the quality of the data for some of the projects. For example, one team had consolidated bug reports from another bug tracking system into the SourceForge tracker. Unfortunately, these copied-over bugs all appeared in SourceForge to have been opened and closed within minutes, so this project was eliminated from further analysis.

4.1 Development team size

Since the FLOSS development process relies on contributions from active users as well as core developers, we sought a measure that reflected the size of this extended team, rather than just the core developers listed on the project page. As a proxy for the size of the extended development community, we counted the number of individuals who posted a bug or message to the bug tracker. As described above, the poster of bug reports and related messages are identified by a SourceForge ID (though postings can be anonymous), making it possible to count the number of distinct IDs appearing for each project. The counts were log transformed to correct skew.

4.2 Bug fixing time

To assess a project's performance in bug fixing, we examined how long it took the program to fix bugs by calculating the lifespan of each bug from report to close using the timestamps recorded by SourceForge. The most straightforward analysis would be to calculate each project's average bug-fixing time. However, this approach has several problems. First, the time taken to fix bugs is highly skewed, making an average unrepresentative. Second and more problematically, because not all bugs were closed at the time of our study, we do not always know the actual lifespan, but only a lower bound. This type of data is described as censored. Bugs are assumed to be reported at random times, making the censoring time (the point of our data collection) independent of the bug lifespan. As a result, the data exhibits "random type I right-censoring" [7]. Finally, analyzing only the average does not take into account available bug-level data. If there are differences between projects in the types of bugs reported (e.g., in their severity), then these differences could affect the average lifespan for a project.

Analysis of censored lifespan data involves a statistical approach known as survival or event history analysis. The basic idea is to calculate from the life spans the hazard function, which is the instantaneous probability of a bug being fixed at any point during its life. The hazard function can then be used as a dependent variable in a regression. Equivalently, the analysis can be done of the survival rate, which is the percentage of the bugs still open after a given lifespan.

In order to ensure that the hazard function is always positive, it is typical to regress on the log of the hazard function. Various functional forms can be assumed for the hazard function vs. time: for example, in an analysis of human death rates, the hazard function is generally assumed to rise over time as the person ages, while for an analysis of physical events, it might be assumed to be constant. Factors that increase or decrease the hazard rate can be added as additional variables in the regression.

For our initial purpose of developing a project-level measure of bug fixing effectiveness, we simply entered project as a factor in the hazard regression along with the bug priority, allowing us to compute a hazard ratio for each project (ratio because of the use of the log of the hazard). The hazard ratios are the regression weights in the hazard function for the dummy variable for each project, using the first project as the baseline. The analysis was performed using the R-Project statistical system (http://www.r-project.org/), specifically the psm function from the Survival and Design packages.

4.3 Activity level and number of downloads

The two remaining measures, activity level and number of downloads, were taken directly from SourceForge and so did not require further analysis, other than a log transformation of both to correct skew.

5. Results

In this section, we present the results of our analysis, starting with descriptive data about each measure and then considering relationships between measures.

5.1. Descriptive data

We obtained data on a total of 62,110 bug reports, an average of 509 per project. The median number of reports was 279, indicating a skewed distribution of bug report counts. We counted a total of 14,922 unique IDs, of whom 1,280 were involved in more than one project (one was involved in 8 of the projects in our sample). Table 1 presents descriptive data for the four measures analyzed. A plot of the survival rate over time for bugs is shown in Figure 2. The plot shows that bugs with higher priorities are generally fixed more quickly, as expected, but some bugs remain unfixed even after years.

We experimented with different functional forms for fitting the bug hazard rates. Somewhat surprisingly, the form that fitted best was an exponential, that is, one in which the hazard rate is not time varying. The R^2 for the fit was 0.51, which indicates a good fit. To assess the

validity of the computed hazard ratios as a measure of bug-fixing performance, we compared the hazard ratios to median bug lifespan for each project. The high correlation between these values (r=0.88) suggests that the computed hazard ratio does capture the performance of the project team on fixing bugs. Of the two measures, the hazard ratio seems preferable for further analysis because it takes into account the open bugs and possible differences between projects in the mix of bug priorities experienced.

5.2 Relationships among success measures

Table 2 shows the correlations among the four measures proposed. The signs are all in the expected directions, since the community size and number of downloads should increase with success, while the activity rank and hazard ratio decrease (rank 1 is most active). Four of the six correlations are statistically significant (with 122 cases, the critical value for significance at p=0.05 is 0.17) indicating that there is a genuine relationship amongst them. The high correlations between activity rank, community size and number of downloads indicates that they are measuring a common factor, while the hazard ratio seems to be less connected.

	Mean	Median	Stdev
Community size	138.4	87.5	206
Activity rank	620	423	586
Hazard ratio	1.009	0.9438	1.06
Downloads	246,294	36,493	817,874

Table 1. Descriptive data for the four measures.

	CS	AR	HR	D
Community size	1.00	-0.73	-0.27	0.48
Activity rank	-0.73	1.00	0.14	-0.70
Hazard ratio	-0.27	0.14	1.00	0.06
Downloads	0.48	-0.70	0.06	1.00

Table 2. Correlations among measures.

6. Discussion

As mentioned in the introduction, the four measures applied in this paper have face validity as indicators. The preliminary analysis presented above allows us to extend our examination of the validity and utility of the measures.

The high correlation among these measures indicates a degree of convergent validity, since the different measures do correlate, particularly number of developers, activity and number of downloads. However, examining these correlations and correlations with other variables in more detail suggests room for improvement in the measures. In retrospect, our use of the SourceForge activity measure is problematic, since that activity is calculated using a formula that includes the number of bug reports and downloads. The high correlation of activity and downloads can thus be explained—they literally do measure the same construct.

The hazard ratio, on the other hand, is less strongly correlated, suggesting that it provides an independent view of a project's performance. This variable is most highly correlated with the size of the development community, which makes sense: more developers means that there is more help available to close bugs.

Another type of validity is predictive validity, meaning that the measures predict other variables of interest. The high correlations among variables are of little importance if those variables do not help us to understand the projects. We found that our proposed measure of project team size is highly correlated with the number of bug reports (r=0.68). The correlation between the age of a project and the number of bugs is low (r=0.09), suggesting that it is not simply the case that older projects have more time to collect reports and people. In other words, this correlation does seem to reflect a real phenomenon: more bug reports are indicative of a larger extended community. Interestingly, there is a negative correlation between the number of bugs and the time taken to fix them: projects with more developers report more bugs and then fix those bugs somewhat more quickly.

However, preliminary examination of the relationship between these measures and other data raise a second concern. In building a sample of projects to study, we seem to have found projects that all seem to be successful, by and large. As a result, the sample may not have sufficient variance on success, meaning that correlations between these success measures and other variables will be noise. To address this concern, we should collect data on a broader range of projects, including some that seem clearly to be unsuccessful.

7. Conclusions

This paper makes a contribution to the developing body of empirical research on FLOSS by identifying and operationalizing four success measures that might be applied to FLOSS. We have collected data on these measures for a set of SourceForge projects and shown the relations among these measures. We emphasize again that we

do not view any single measure as the final word on success. As the measures draw on different aspects of the development process, they offer different perspectives on the process. Including multiple measures in a portfolio should provide a better assessment of the effectiveness of each project.

Having identified particular effective projects, our future work includes more detailed analysis of the projects. We plan to employ a theoretical sampling strategy to choose a few FLOSS development teams to study in depth. By limiting the number of projects, we will be able to use more labour-intensive data analysis approaches to shed more light on the practices of effect FLOSS teams.

8. References

[1] W. Scacchi, "Software Development Practices in Open Software Development Communities: A Comparative Case Study (Position Paper)," 2002.

[2] R. A. Ghosh, "Free/Libre and Open Source Software: Survey and Study. Report of the FLOSS Workshop on Advancing the Research Agenda on Free / Open Source Software," European Commission, http://www.infonomics.nl/FLOSS /report/workshopreport.htm, 2002.

[3] K. Crowston, H. Annabi, and J. Howison, "Defining Open Source Software project success," in Proceedings of the 24th International Conference on Information Systems (ICIS 2003). Seattle, WA, 2003.

[4] W. H. DeLone and E. R. McLean, "The DeLone and McLean model of information systems success: a ten-year update," J. Manage. Inform. Syst., vol. 19, pp. 9-30, 2003.

[5] W. H. DeLone and E. R. McLean, "Information Systems Success: The Quest for the Dependent Variable," Information Systems Research, vol. 3, pp. 60–95, 1992.

[6] S. Krishnamurthy, "Cave or Community? An Empirical Examination of 100 Mature Open Source Projects," University of Washington, Bothell, Bothell, WA May 2002.

[7] F. E. Harrell, Regression modeling strategies: With applications to linear models, logistic regression, and survival analysis. New York: Springer, 2001.

Toward Collaborative Open-Source Technology Transfer

Jean-Michel Dalle
IMRI-Dauphine, Univ. Paris 6 & Agoranov
jean-michel.dalle@upmc.fr

Guillaume Rousseau
INRIA
guillaume.rousseau@inria.fr

Abstract[1]

We analyze several occurrences of open-source technology transfer where research tools or prototypes developed in academic environments are transferred to private actors to be exploited economically. We enlight common characteristics which lead us to suggest that academic duality is a general consequence of the academic design of research tools and prototypes, and that the associated high transfer costs could be reduced first by implementing dual versioning using a dual licensing scheme, by associating a new academic public license with a traditional technology transfer one, and second through the transparent and shared maintenance of products built according to a dual architecture, as it would be precisely allowed by a dedicated collaborative development platform such as LibreSource.

1. Position of the problem

Software is an increasingly common output of academic research. Computer science theories and ideas are widely tested and implemented as software prototypes. Even more generally, software has become a common research tool in all disciplines, for calculus, simulation and many other tasks associated with academic life. To speak only of the most famous of the latter category, it is now well-know that academics at CERN in Geneva were largely responsible for the invention of the World Wide Web, first as a research tool. But then an important issue arises about how software developed in academic communities is or could be made accessible to a larger public, when of course relevant: and it would indeed be relevant for numerous software technologies that could be usefully exploited in the context of economic products and processes. To put it differently, the importance of the *technology transfer* issue for software research tools and prototypes is increasing rapidly.

Technology transfer issues have been studied for years in economics and other disciplines, and it is not our purpose here to assess the various results that this literature has brought, nor to discuss the hypotheses that have been suggested, and sometimes discarded, about how to improve the transfer of technologies from public-funded research to private actors and markets. It would certainly be fruitful to do so, but we would like to inquire here the issue of software technology transfer according to a different perspective: the reason for this is the rapid surge of a new and fashionable mode of technology transfer for software, namely, *open-source technology transfer*.

Building upon the success of open-source software, for which Linux has become a paradigm, software developed in academia is now increasingly made accessible in an open-source way: not only because what could be called "GPL-publishing", by researchers on their homepages, but also because of more developed attempts of open-sourcing software initiated by higher education and research institutions. Indeed, the initial idea of this study comes from the fact that, in the context of our work as researchers in the economics and management of software and innovation, we suddenly found ourselves coming across more and more numerous examples of academic software developed in open-source mode, many of which relatively recent, some of which older than we would have expected[2].

[1] We would like to acknowledge how much our work has benefited from various discussions with other participants in the 'LibreSource' RNTL grant, whose support is also most gratefully acknowledged: among them, we would notably like to thank Vincent Finet, Laure Muselli and Olivier Rhein. Other discussions with Paul A. David, Rishab A. Ghosh, Gérard Giraudon, Jesus Gonzales y Baharona, Nicolas Jullien and Laurent Kott have also been very helpful in shaping our views. This paper draws significantly from a previous one [3] which was presented at EPIP2 Conference in Maastricht, The Netherlands: most surprisingly, there was a slogan written there, on the wall of the room in which this conference was organized and the paper presented, that was said to date back to the Princes of Oranje, and which read "Je maintiendrai". This sentence in French translates into "I shall maintain": surprised as he was, it certainly helped one of us (JMD) understand that maintenance was key, and that organizing it via a collaborative platform could be crucial for open-source technology transfer.

[2] See [2, 17, 21] for early intuitions about this.

In any case, academia now appears as a *second major source of open-source software* after independent developer communities[3]: a source about which we do not know much yet, and about which we would certainly like to know more to properly assess the consequences of the current wave and perhaps tsunami of open-source academic software research tools and prototypes. This is the basic rationale of this work, which we hope will be followed by others as such studies appear to be rapidly needed by research policy makers, among whose preoccupations stands notably the current disarray of traditional technology transfer offices for which open-source software is still a new and puzzling phenomenon, apparently associated only to non-profit mechanisms – an erroneous conception which we will try to correct below – whereas these institutions have most of the time been created to foster the commercial exploitation of technologies developed in labs, be it by contractual research, licensing, or start-up creation. Open-source cases are on most of their desks now, but they do not know how to handle them according to this new mode of technology transfer[4].

As a further consequence, we will not consider here under which conditions open-source technology transfer could be more efficient than other modes. Considering how fast the open-source technology transfer wave is progressing, we have rather adopted here a more pragmatic approach, and we will try to determine using various case studies the major characteristics of open-source technology transfer, and from then on the conditions under which open-source technology transfer could be made more efficient. As a matter of fact, answers to these questions are relevant for future comparisons with other modes, and could also first of all provide a rationale to try to stop, or not to stop the current wave, or perhaps to try to reorient it in some particular way. In this context, we will start by briefly presenting case studies, and then our main findings for which we will try to provide interpretations, on the basis on which we will finally propose a more general framework for open-source technology transfer[5].

2. Case Studies

We will now briefly present some of the cases studied in the context of the LibreSource project, and which all concern open-source software developed in academia. For now, and since it is an on-going work, we apologize for designating them only with initials: we would notably like to preserve their anonymity until complete conclusions have been reached and disclosed to all relevant parties[6].

- Technology A is a CAD tool for integrated circuit design at a very advanced stage of development. It has now reached wide recognition in its field, and is widely used in the academic community worldwide. However, it is not used by private companies, which prefer proprietary solutions and which have developed in-house plug-in libraries adapted to their own specific circuit elements. However, Technology A is modular and globally interoperable with these proprietary solutions. Technology is essentially a research tool, and is considered to have been a key element to grant the team that developed it with a high academic standard, and to allow it to receive around 1 M€ per year in contractual research from industrial partners, which was notably used to pay for an engineer who maintained Technology A and supplied user support. Technology A's licensing scheme was initially unclear, but it was later released under the GPL. 3 start-ups at least have been created in connexion with technology A, each of which has received VC funding. Two of them market circuits developed using technology A, i.e. research results obtained using A as a research tool, and the last one markets an improved closed and proprietary version of one of A's components, also compatible with proprietary solutions.
- Technology S is a tool for calculus and simulation at a very advanced stage of development. It is not completely modular, although dedicated toolboxes exist and are suited to specific needs. It is a direct competitor of a proprietary software solution that is in a monopoly situation, which benefits from a much greater number of specialized toolboxes, and with which it is not interoperable although S initially forked from the last open-source version of this other technology before it was closed and turned into a commercial product. S is released under a sui generis licence that grants the team which develops technology A with very strict control over contributions, and which does not guarantee explicit recognition for contributors. S is itself challenged by another open-source solution released under the GPL, and which is interoperable with their common proprietary competitor. A consortium of commercial users, mostly

[3] Although general results about open-source software mostly apply: [10] provides a synthetic view. See also [11, 13, 14, 16, 20] about motivations and [6, 7] about competition issues.

[4] These issues are now crucial in France for INRIA, RNTL, CNRS, CEA, ANVAR and for several major universities, to name but a few major actors. We have had various occasions to verify that there were also emerging rapidly in several other countries, as in the UK for the EPSRC or in the US for the NSF.

[5] As should already be clear to the reader, the line of inquiry we have selected implies that our conclusions will for now stand only as conjectures, that is mainly as hypotheses for further and more quantitative research. This team is of course committed to contributing to this task, but a major motivation for this early release – applying to ourselves what other research on open-source development [5] has taught us –, is that we believe that much can to be gained by disclosing results, here on open-source technology transfer, early enough so that they can motivate further inquiries and further research.

[6] A more complete description will be included in the final LibreSource report (forthcoming in 2004).

big companies, has recently been set up, operated by one of the research institutions from which S originates, whose members pay an annual fee in exchange for premium services, and notably to see their suggestions for further development taken into proper account by the academic development team. In this context, technology S's licensing scheme is about to evolve, though the eventual choice is not completely determined yet, possibly toward a dual GPL – LGPL scheme with LGPL rights being granted to members of the consortium. At least one start-up is a member of this consortium and plays the role a specialized integrator for technology S: a similar company exists in Germany but does not seem to be part of the consortium at least for now, contrary to a non-specialized company dedicated to open-source solutions but which is willing to develop a similar business.

- Technology J is a J2EE compliant application server at a very advanced stage of development. It has for the last 3 years become part of a consortium supported by several major companies, and which regroups technology J together with a set of other components dedicated to related functionalities. The consortium is here also, as for S, a contract between users and the research institution from which the technology originates, according to which an annual fee is paid in exchange for premium services and notably inclusion of their expressed needs in upcoming versions. Most of the components regrouped in this consortium along with J are licensed under the GPL, including J, and the consortium plays a coordination role among them to try to foster interoperability and convergence. Numerous individuals, both academics and employees of various companies using J, and also several start-ups, are members of this consortium, together with a major systems integrator, which has partly embraced an open-source strategy, and for which technology J constitutes a key middleware component. The consortium's web site is basically a repository for all these components, and has recently started to host yet another well-known middleware component.
- Technology X is a desktop grid solution at a relatively early stage of development. It has been developed in the context of a successful PhD thesis and was from the beginning released under the GPL, although this licensing scheme does not seem to reflect a proper decision of the university under the auspices of which it has been developed. Technology X is in the process of rapidly achieving a wide recognition and installed academic worldwide. Several potential commercial users have already approached the team that maintains it and plans to continue to develop it with relatively pressing needs, and this team is therefore currently involved in several start-up projects. The business model of these start-ups is not completely determined yet, between a specialized integrator which would answer clients' needs in the context of contracts which could otherwise have been directly passed through the university as research contracts, and a more ambitious component strategy which would be more competitive worldwide with other desktop grid solutions.
- Finally, we would also like to add to this least another well-known example of open-source technology in the process of being transferred to commercial uses, namely the Globus technology for grid computing, denoted here as G for coherence reasons, as it has been studied recently in [9] and to which we would like to directly refer the reader.

3. Preliminary Findings

(i) *Maintenance and governance.* Academic labs and research institutions are generally willing to retain at least *partial control* over software maintenance and development using appropriate licensing schemes (S, G), or by creating consortia that they themselves more or less control (S, J), or simply by continuing to invest in the development and act as maintainers of the technology (all including A & X). The existence of consortia basically tends to balance pure academic leadership and to shift part of the authority to steering committees where both academics and users are represented. Consortia also seem to increase the number of non-academic users, sometimes although on a purely individual basis (J).

(ii) *Licensing.* Very different licensing strategies have been and are being explored, and no general solution has emerged yet. However, viral licensing is now present is all projects we have studied. In this context, and this is clearly related to maintenance and governance issues mentioned above, we have found several examples of legal control over contributions brought by a centralization of rights in the licensing schemes: rights on contributions are to be given to the maintainer's institution, contrary to pure GPL licensing which as a consequence does not prevent forking – although commitment strategies of academic research teams seem to play a counterbalancing role and are generally held to prevent it. The possible existence of software patents held by contributors can also be taken into account so that it is guaranteed that they will not be harmful to the maintainer and to the users of the technology. In addition, we have also found evidence of the desire of some contributors at least to be rewarded for their contributions and notably granted, although we have found no BSD-like clause. How licensing schemes could be designed both to allow for commercial exploitation and to correspond to the strategy of research teams is a major concern for several projects.

(iii) *Commercial partners and start-ups.* Major companies (A, S, J, G) and start-ups (all) are present in most cases. As for major companies, they can be either

users (A, S, J) or systems integrators (J, G), and will deal with academic teams through one-to-one contracting (A and probably G) or through consortium agreements (S, J). As for start-ups, it is unclear whether their prevalence is an artefact due typically to the fact that IT technologies have been especially prone to start-ups creation during the recent years. However, we would like to conjecture here that access to open-source technologies, which are also more visible from outside academic communities and acquire more rapidly a larger, at least academic, installed base, is by construction associated with lower barriers to entry: as a consequence, open-source technologies born in academia are more suitable to be marketed by newly created specialized integrators, i.e. are specially prone to start-up creation. More, labs can here also be more attracted by partners whose bargaining power is relatively limited, meaning that the deals will be more in their advantage and that they will retain more control.

(iv) *Maintenance costs.* In all cases, there is a more or less pressing need of academic players to find a way to finance maintenance costs, since higher education and research institutions experience difficulties in justifying long-term financial commitment of development work. These maintenance costs are sometimes paid for by contractual research attracted by the academic reputation that the team has earned notably due to its achievements in developing a powerful technology and obtaining good research results and academic leadership in its field (A specially and in some extent all others). Consortia partly play a similar role, but extend it by increasing the bargaining power of private actors i.e. by linking a fee paid by commercial entities to more influence on future developments.

(v) *Markets.* Various market conditions characterize each of the technology we have studied: it can be characterized by a de facto standard (S), by a limited number of proprietary solutions (A) probably on the way to de facto standardization (J), or by a rapidly evolving competition (X) where the technology can be dominant (G). In the case of S, a previous technology transfer had been attempted, in traditional closed and proprietary mode, which did not meet success. Actually, these market conditions, and the characteristics of software markets, have most probably played a role in driving most of the technologies to an open-source technology transfer strategy, compared to a direct transfer of the technology in a non open-source mode (A, S, J at least). Due to market dynamics provoked by network effects and externalities, the real chances of a technology, specially when it is not among the early entrants but a latecomer and when it is not supported by considerable investments, to gain a fair share in a direct proprietary competition against other proprietary software producers are relatively low and hazardous. It is particularly so as soon as a de facto standard has already emerged from a former standard race (S, probably soon J), or an equivalent market structure with very high barriers to entry (A). In this respect, a technology born in academia can be a late comer for at least two different reasons: first, because it was developed in a pure academic context for years before the opportunity of commercial exploitation was really explored (J), or because a first and attempt has failed and a new one is launched several years later in completely different market conditions (S); and second, because this technology has often been created as a research tool, i.e. for the research team which develops it and the academic community to have access to an open-source, easy-to-use, and quasi-free research tool, and therefore to avoid having to pay for expensive proprietary solutions whose sources are in most cases unavailable, even on a monetary basis, which renders academic work considerably more difficult, if not unfeasible (A). Whatever the reason – sometimes both could apply – proprietary technology transfer strategies would appear rather quixotic when it would be so, and the technology might experience a better diffusion trajectory and reach a larger installed base and a better global outcome (considering also the reputation reward for the research team), according to an open-source and even a software generics strategy, i.e. in playing the role of a software generic in its particular market. In parallel, supplementary indirect technology transfer strategies can be implemented and include selling components interoperable with the solutions that dominate the market, or exploiting of research results obtained by using the technology as a research tool (A). In this context, more recent technologies (X) are wondering whether they should immediately adopt a software generics strategy or if they have entered the market early enough to have other opportunities, i.e. if the tipping point of the standard race in their market is not too close they could for instance adopt strategies like the one that G has selected i.e. trying to play the role of an open-source de facto standard which most proprietary software producers now implement.

(vi) *Academic installed base.* In most cases, these technologies are widely used in the academic world for teaching and research, and their academic installed base is much bigger than their commercial installed base. This is clearly due to the openness of their sources and to their being accessible for free. There is at least one major limit for this (S notably), which concerns interoperability with dominant proprietary solutions: the basic issue faced by higher education institution when they use an open-source alternative technology for teaching, is that students would have to learn later different procedures if it is not interoperable enough with the solutions which dominate the market, which significantly reduces their employability when it is so.

(vii) *Modularity and design.* Most technologies have some degree of modularity [1, 15], but still relatively imperfect. They are often said to be modular, but we have not been able to really obtain evidence of the extent in which modularity had concretely been achieved in their design, while in some cases it is even not completely clear whether modularity is implemented in the current version of the code or if plans exist to improve it in future versions. Other puzzling design characteristics include for instance the non-interoperability of S for instance, which harms its competitive power both against the existing proprietary standard and against its open-source competitor, even in the academic institutions in this last case. Clearly, imperfectly modular design has to do with the fact that modularity was less of a rule when the older technologies were designed. But we would also like to conjecture that design by academic teams also reflects the context and objectives according to which these technologies were initially conceived, namely, to serve as research tools or as research prototypes, without a proper taking into account of other and notably commercial considerations which could have pleaded for a more modular design and also for other kinds of specifications such as interoperability. *Academic design*, as we suggest to denote this phenomenon, is indeed a good candidate to explain some of the difficulties in technology transfer attempts, as the technology is not well adapted ex ante to commercial users' needs, and the necessary investments to modify its design are thus all the more difficult to finance that initial design decisions are path-dependent. What we are basically suggesting is that technologies created in academia, often as research tools and sometimes as research prototypes, are simply initially designed in an academic way without taking into account other considerations, and that this design can render technology transfer more difficult i.e. more costly for any commercial entity which would be willing to attempt it.

Another way to put it is to say that academic design significantly increases *transfer costs* and therefore the investments private actors have to make, including sustained contractual relations with the initial designers of the technology so as to benefit as much as possible from their skills and from the tacit and uncodified knowledge they possess, to create marketable products and processes. This phenomenon actually has a name in economics, although it does not usually apply to academic but to military technologies: it is called duality and characterizes the fact that military specifications imply characteristics – or design decisions in a more modern terminology – which do not correspond to civilian uses, hence creating a high transfer cost for so-called dual technologies . Here, transfer costs between academic and commercial use are probably be more limited but still exist and stand as a logical explanation for weak commercial use.

Academic duality is a consequence of academic design: most technologies born in labs are created as research tools without any commercial strategy and are designed according to academic standards, and this applies also in a large extent to others research prototypes as the main motivation of scientists is to design them so as to earn reputation from their peers, therefore in line with specifications chosen independently by researchers according to the preferences of the academic community. A further consequence now is that contract research between academic and commercial partners can then be interpreted, at least in some cases, as reflecting an *opportunity cost*: the opportunity cost for the lab of adopting this or that design characteristic instead of the 'academic one'. Interested commercial partners can pay for these opportunity costs on a direct contractual basis, but this rationale is even more clearly reflected by their willingness to engage financially into consortia and to receive in exchange extra bargaining power on design decisions. However, they can most probably not afford these costs when it comes to modifying older design decisions[7].

4. Reducing and organizing academic duality

Academic design and academic duality tends to characterize software, and probably most scientific artefacts[8], both research tools and prototypes, which creates structural technology transfer costs. Clearly then, ignoring academic duality in the design of technology transfer mechanisms would probably lead to repeatedly unsuccessful strategies. In this respect, we would like to suggest here a modest and tentative proposal for open-source technology transfers: an open-source proposal indeed, which we hope will attract criticism and further contributions. Central in this proposal is the notion of organized and reduced duality, and it aims at structurally reducing technology transfer costs. The basic rationale is the following:

4.1. Reducing academic duality for research tools and prototypes

It could first be possible to optimize the *early design* of prototypes, while completely respecting academic motivations. As soon as some of the characteristics of spontaneous or emerging academic design are not specified by the academic context, they should be

[7] At least not in a consortium where others would also benefit i.e. if they would not be able to appropriate returns sufficiently: a traditional public good dilemma, which exclusive rights can solve.

[8] This corresponds indeed completely to other personal experiences that both of us have acquired in various technology transfer offices and science-based incubators.

specified so as to reduce the future cost of technology transfer if such an opportunity would occur. This is especially important when choices made implicitly by researchers are particularly relevant for commercial exploitation, such as interoperability and standardization issues (see above).

We believe that technology transfer offices within higher education and research institutions should have a policy about how to contribute in this respect: in any case, the earlier the better, to the limit that academic technologies are initially developed voluntarily by researchers in the context of their research projects without basically reporting to their host institutions. However, as soon as funding is sought by and allowed to research projects that involve prototyping or development of research tool, it could be easier to ask typically for an explicit design[9].

It would be in the interest of researchers as it could allow them to attract more interesting and rewarding research contracts in the future. As a matter of fact, former research contracts and contacts with private actors could play a significant role in providing researchers with information that they do not possess about relevant characteristics for commercial design, and could sometimes even result in early research contracts for which industrial partners would pay to cover the opportunity costs which correspond to design changes compared to spontaneous academic design, without influencing otherwise academic strategies.

Such a method for reducing academic duality is crucial for software, because design decisions such as interoperability are of special importance, and because software technologies need less capital investment to be developed and are often necessary and expected by peers to validate research work, which implies that software prototypes are often developed by researchers in academic contexts. However, it would clearly apply in other disciplines too, as soon as they engage in creating research tools, be they software or not, or in developing various prototypes.

4.2. Organizing academic duality: dual versioning and dual licensing

Complementarily to the former way of reducing academic duality, *dual versioning* could be a further method of organizing it and of reducing technology transfer costs. The basic idea is to allow for the coexistence of both an academic version and a commercial version, but to organize this coexistence in an efficient way: namely, reconciling versioning both in the economic sense (i.e. segmenting between markets with different products) and in the software engineering sense (allowing for different versions properly speaking). A way to do this is to open-source technology transfer and to use a double licensing scheme similar to those that are now been implemented by various open-source vendors[10].

The academic version of technology T should be licensed under a viral license, which almost completely prevents any commercial use since it implies that any piece of software incorporating T should be under the same license, i.e. it would necessarily be open and could not be integrated with closed proprietary components: this licensing scheme therefore creates a credible threat, as soon as it is enforceable, which forces any private entity willing to sell closed software incorporating T to ask for another license which would typically include royalties and similar clauses[11]. There would therefore coexist an academic version which could typically be used as a research tool by any interested researcher in the world, and commercial versions while private actors would develop by integrating T with complementary proprietary module which would typically make T more efficient for commercial users, a typical such module of course having for instance to do with efficient user interface.

A corollary is then that to successfully implement this scheme, rights have necessarily to be centralized i.e. given to an institution which is able to a) re-license them, notably to interested commercial partners b) represent a really credible threat for counterfeiters – remember that T is widely accessible in its open-source academic version – : the latter is actually done by the FSF for numerous free software projects as it has been spontaneously granted rights by many contributors, but the FSF could on the contrary not involve in the former i.e. re-license any of these projects because it does not possess all the rights. As a consequence, the viral license will in this context, contrary to other dual licensing schemes, not be GPL-like but rather QPL-like (a license created and used for technology Qt) or GTPL-like (a license created and used

[9] A very interesting policy in this respect has been implemented by INRIA, which has a dedicated crew of developers who are assigned to academic projects on a fixed-term basis but who also remain under the supervision and depend upon the authority of the technology transfer office.

[10] About dual licensing, see e.g. [19, 22]. In shaping our views, we have also benefited from several discussions during the eScience workshop held in Oxford on May 2nd 2003, notably with Paul A. David, Rishab A. Ghosh and Tony Hey, after which [12] has suggested a different dual licensing scheme for publicly funded research, which would combine the GPL and a commercial licence, but which we fear would however unfortunately not be appropriate to academic technology transfers, as it would not technically speaking comply with the necessary centralization of rights and would not correspond either to the motivations of academic scientists for maintenance and for citations.

[11] If relevant, exclusive rights could be granted, notably to start-ups, to allow them to raise private funding and to invest more in the development of an efficient commercial version, provided of course a regular assessment is made of the extent of the exploitation they make of the technology with a return clause if exploitation falls below a given threshold, as is usually the case now for technology transfer.

for grid technology Globus) as both of these licenses have implemented clauses that guarantee the centralization of rights, even when rights owners would have filed patents.

However, as soon as there is a crucial and pressing need to adopt a non-standard license, it might be useful to account also for several other features of open-source technology transfers as they have been presented above. The specifications of a new *academic public license* (APL) could then include: (i) A BSD-like clause that would guarantees that contributors would be explicitly credited, and would therefore account for the motivations of scientists in a similar way as the GPL corresponds to the motivations of independent developers: in the academic world, these motivations specially have indeed to do with citations, and committing to credit contributions would attract more academic contributors; and (ii) a clause that would clearly specify applicable law and courts: all projects to which the academic public license would apply would then point to only one court, which would help this court to develop appropriate skills in such new, specialized and delicate matters.

In this last respect, we would like to suggest that French law could have, in some extent, an interesting and distinctive feature here, as it does not rely on *copyright* but on *'droits d'auteur'*: on 'author rights' (*authoright*?). Among those rights that authors possess are "moral rights", which are specifically instituted by law as completely inalienable, meaning that they cannot be sold or abandoned, whatever way. As a consequence, granting all of their 'other' rights to a third party which would centralize them to re-license them according to a dual licensing scheme could then be counterbalanced for developers according to French Law by their retaining of their moral rights, and therefore by their being able to forbid exploitation under certain circumstances which would be strictly contrary to their motivations. This idea would certainly need further legal inquiries, but if it was so it would allow to re-implement simply in a dual licensing context the exact kind of 'credible commitment' mechanism [24] that we had suggested crucial to explain the success and sustainability of GPL'd 'Libre' software [7].

The design of such an academic license would also have the supplementary benefit of being in line with the astute suggestion [9] that modularity and standardization should be developed for technology transfer contracting. Indeed, the rights of the academic public license itself could be given to a relevant international institution, which would have the duty to maintain it in an open-source way so as to adapt it to new and unexpected events and situations as they would inevitably occur, and which would make this institution a good candidate to suggest generic clauses for the other commercial part of the licensing scheme we are describing here. And it could also be granted all the rights of some projects at least, but in this case several academic and semi-academic institutions would also have direct incentives to play a centralizing role for this or that project. By the way, a commitment from these institutions to use a fair share of whatever revenues they would get from open-source technology transfers to finance new and further developments, and also new research projects, would certainly play a major role in attracting more projects and could also be instrumental in implementing a credible commitment mechanism between academic communities, their representative institutions, and private actors. These representative institutions could then easily be foundations, though higher education institutions could also directly play this role: on the contrary, it is highly probable that commercial entities or even consortia including commercial partners would not be able to guarantee an appropriate governance structure and a credible enough commitment to such an organized versioning – the risk being that both versions could converge to a single one to the detriment of the academic version.

4.3. Organizing academic duality with a dedicated collaborative platform

Still, this is most probably not enough, as we have seen that maintenance and governance were specially relevant issues for open-source software technology transfer. We would like to suggest that the use of a dedicated and well-adapted collaborative development platform could considerably help here, in providing a joint framework in which all kinds of consortia could work and be implemented, while at the same time basically reducing maintenance costs.

First, different partners could maintain different modules separately. To put it differently, software could be designed to fit as best as possible with a *dual architecture* which could in itself reduce and organize academic duality: some modules would be academically maintained, while others would be commercially maintained. Organizing duality according to such a division of innovative labor, and *shared maintenance*, would indeed correspond well to two different incentive and motivation systems: one associated with reputation and peer-review [8], which would focus more on algorithmic aspects and lower layer issues, and the other one with profit and the size of the market with a special emphasis on graphic user interfaces and similar functionalities [23]. Indeed, contributions to the academic components could even play a role similar to citations for academics and these modules would in a sense truly become *software journals*.

Such an architecture would obviously reduce technology transfer costs, and would fit also well with dual licensing, but organizing correctly such an improved

division of innovative labor needs an appropriate environment: all the more so when there are numerous partners, and when, as is always the case, perfect dual architectures do not exist, and there are interdependencies between modules which imply that teams still have interactions with each other's work, that shared maintenance extends within modules, therefore generating transfer costs and potential conflicts, which only an appropriate governance structure which would combine authority and leadership with *transparency* could help to solve efficiently. Indeed, transparency, i.e. an easy monitoring of decisions by all relevant parties, is all the more necessary here as the centralization of rights reduces the risk of forking, and hence puts less pressure on the quality of decisions taken by authorities.

A simple way to implement all of this is to host development on a well-adapted collaborative development platform, which would also make decisions both visible and challengeable. Such a platform should be more distributed that CVS not only to simply make development more efficient, but also to accommodate concurrent development by several non-profit and for-profit entities with different time horizons and workloads. More, it should be able to implement efficient user right management, including intellectual property right management, via an efficient tracking of author contributions. And it should be able to provide efficient conflict resolution procedures, notably for the file system.

LibreSource is precisely being designed and developed as a distributed collaborative development platform to manage large collaborative academic and industrial software development projects involving academic and industrial resources. LibreSource possesses all the standard tools of a collaborative development platform, such as a project and user administration module, a Web interface, a bug tracking system, a forum, a wikki, etc. LibreSource also allows a high level of integration within network infrastructures (full compatibility with main firewalls), user environments (client side applications run through JavaWebStart), hosting services (Windows and Linux are supported, and LibreSource is fully based on J2EE-compliant technologies). In addition, LibreSource incorporates functionalities of flexible synchronization "DataFlow" using synchronization networks, a workflow engine "Bonita" allowing the implementation of complex specialised processes, and high-level synchronization and conflict resolution procedures [18].

5. Conclusion

We have tried to suggest here that, if academic duality is largely inevitable in technology transfer situations, due to academic design, therefore contributing to increase technology transfer costs, it could be reduced and organized. Combined together, we believe that the suggestions presented above could integrate into a new framework which would significantly reduce and organize academic duality, and which would fit well with the various observations we have made about open-source technology transfer situations: about maintenance; licensing schemes; contracting with commercial partners, notably to support maintenance costs; market conditions; academic installed bases; and, last but not least, modularity and design issues for research tools and research prototypes. Needless to say, this framework does not solve all issues, notably those which have to do with its practical implementation in various market conditions, and thus with the need for different commercial partners and for suitable contractual arrangements: all of which will have to be practically dealt with in every specific situation by technology transfer office professionals. Finally, we would like to stress that, as they have been presented here, these ideas are mainly specific to software: however, we believe that similar suggestions could be made in other technology transfer situations, as for instance for biotech databases and for educational resources [4].

6. References

[1] Baldwin C.Y., Clark K.B. (1997), Managing in an Age of Modularity, Harvard Business Review, Sept.-Oct., pp. 84-93.

[2] Dalle J.-M. (2002), Open code : the sources of open-source innovation, presentation at the NSF Workshop on « Advancing the Agenda on Open-Source » held in Arlington, VA, January 28th, and to the DRUID Summer Conference on "Industrial Dynamics of the New and Old Economy – who is embracing whom?", Copenhagen-Elsinore, June 6-8.

[3] Dalle J.-M. (2003), Open-Source Technology Transfer, presented at EPIP2 Conference, Maastricht, The Netherlands, November 24-25th

[4] Dalle J.-M. (2004), Vers un modèle collaboratif en matière d'éducation, forthcoming in a book edited by Fondation pour un Internet Nouvelle Génération (FING), (In French : Englih translation available from the author).

[5] Dalle J.-M, David P.A. (2003), The Allocation of Software Development Resources in 'Open Source' Production Mode, SIEPR Policy paper No. 02-027, accessible at http://siepr.stanford.edu/papers/pdf/02-27.html.

[6] Dalle J.-M., Jullien N. (2000), NT vs. Linux, or some explorations into the economics of Free Software, in Ballot G., Weisbuch G. (eds), Application of simulation to social sciences, Hermès, Paris, pp. 399-416.

[7] Dalle J.-M., Jullien N. (2003), 'Libre' Software: turning fads into institutions, Research Policy 32:1-11.

[8] Dasgupta P., David P.A. (1994), Towards a new economics of science, Research Policy 23: 487-521.

[9] David P.A., Spence M. (2003), Towards institutional infrastructures for e-Science: the scope of the challenge, Final Report of the Oxford Internet Institute project on 'The Institutional Infrastructure of e-Science: The Scope of the Issues'.

[10] Feller J., Fitzgerald B. (2002), Understanding Open Source Software Development, Addison Wesley, Boston, MA, USA.

[11] FLOSS (2002), Final Report, Part IV: Survey of Developers, International Institute of Infonomics, University of Maastricht & Berlecon Research GmbH, accessible at http://www.infonomics.nl/FLOSS/report/Final4.htm.

[12] Ghosh R.A. (2003), Copyleft and dual licensing for publicly funded software development, Draft version (1.0), MERIT – Institute of Infonomics, University of Maastricht.

[13] von Krogh G., Spaeth S. & Lakhani K.R. (2003), Community, Joining, and Specialization in Open Source Software Innovation, presented at HBS-MIT Sloan Free/Open Source Software Conference, June.

[14] Lakhani K., von Hippel E. (2000), How Open Source software works : 'free' user-to-user assistance, MIT Sloan School of Management WP #4117.

[15] Langlois R.N. (2002), Modularity in Technology and Organization, Journal of Economic Behavior and Organization 49:19-37.

[16] Lerner J., Tirole J. (2002), Some simple economics of open source, Journal of Industrial Economics 50:197-234.

[17] Marzouki M., Greiner A. (1999), Du rôle des financements publics de recherche dans le développement du Libre. Étude de cas : la chaîne de CAO Alliance, presentation to the 1rst 'Autour du Libre' Conference, Brest, January 25-27.

[18] Molli P., Oster G., Skaf-Molli H., Imine A. (2003), Using the Transformation Approach to Build a Safe and Generic Data Synchronizer, mimeo.

[19] Muselli L. (2003), Les stratégies de licences libres, presented at conference 'Autour du Libre', Paris, May.

[20] Raymond E.S. (1999), The Cathedral and the Bazaar, O'Reilly.

[21] Rousseau G. (2002), Valorisation des ressources humaines et logicielles en relation avec les logiciels libres, presented at the 2nd NME-NEL Workshop, INRIA, Rocquencourt.

[22] Välimäki M. (2003), Dual Licensing in Open Source Software Industry, Systèmes d´Information et Management, 8.1: 63-75, 2003.

[23] Varian H.R. (1993), Economic Incentives in Software Design, mimeo.

[24] Williamson O.E. (1996), The mechanisms of governance, Oxford UP.

Community structure of modules in the Apache project

Jesús M. González-Barahona, Luis López, Gregorio Robles
Grupo de Sistemas y Comunicaciones – Universidad Rey Juan Carlos
{jgb,llopez,grex}@gsyc.escet.urjc.es

Abstract

The relationships among modules in a software project of a certain size can give us much information about its internal organization and a way to control and monitor development activities and evolution of large libre software projects. In this paper, we show how information available in CVS repositories can be used to study the structure of the modules in a project when they are related by the people working in them, and how techniques taken from the social networks fields can be used to highlight the characteristics of that structure. As a case example, we also show some results of applying this methodology to the Apache project in several points in time. Among other facts, it is shown how the project evolves and is self-structuring, with developer communities of modules corresponding to semantically related families of modules.

1. Introduction

Large libre software[1] projects are usually organized as a set of modules. Each one of them can correspond to a given program, library, or any other unit identified by the project as distinct. It is common that developers work in several modules, according to their interests, skills, and constraints, and that those modules to which they contribute change over time. The relationships among modules due to the people working in them constitutes a sort of social structure of the project. In some sense, those human relationships are the glue that maintain the whole project together, and the chains that contribute to spread information and uses from one part of the project to others. In this paper, we explore how those developer connections contribute to the making of the community structure of the project, how it can be identified and visualized and how it evolves over time.

The study and characterization of complex systems is a very fruitful research area nowadays with may interesting open problems. Special attention has been paid recently to complex networks, where graph and network analysis plays and important role and is gaining great popularity due to its intrinsic power to reduce a particular system to its simple components and relationships. Thanks to this, network characterization is widely used in many scientific and technological disciplines as neurobiology [1], computer networks [2][3], linguistics [4], etc.

Among complex networks, social networks appear in a quite natural way as a method for analyzing the structure and interactions of people and groups of people within complex organizations [5][6][7][8][9]. To understand the structure of those networks, we are interested in determining how the different nodes interact and form groups that, in turn, interact with each other giving rise to higher order groups. The set of groups obtained, as well as their relationships, is which we call the community structure of the network.

All the information we need for such an analysis is available in the CVS repository of the project. Using it, we construct the network of modules at a given time, considering a link when there is a common set of developers that have contributed to both modules. Later, we apply on it some techniques from the social networks field.

There have been some other works analyzing social networks in the libre software world. [10] hypothesizes that the organization of libre software projects can be modeled as self-organizing social networks and shows that this seems to be true at least when studying SourceForge projects. [11] proposes also a sort of network analysis for libre software projects, but taking into account this time a technical connection between modules as code dependency is.

How libre software projects organize themselves in subprojects is an issue that has been already discussed in literature. For instance, in [12] it is argued that projects that require a number of core developers that is larger than a given amount (10 to 15 persons) create in effect several related projects if no other means of code control are introduced. In the conclusions of this paper we will discuss the validity of this hypothesis.

In the rest of this paper we detail the methodology we are using to build the network of modules, and how we identify communities in that network. Later, we show some results of applying this methodology to the Apache

[1] Through this position paper we mean by libre software any program which is either open source software (according to the Open Source Initiative definition) or free software (according to the Free Software Foundation definition).

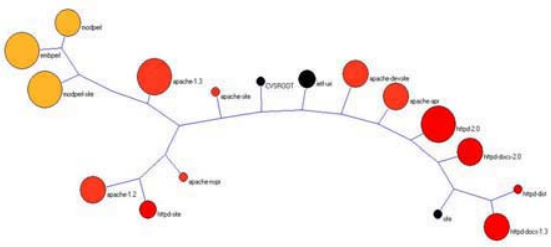
project, and present some conclussions and future lines of work.

2. Methodology

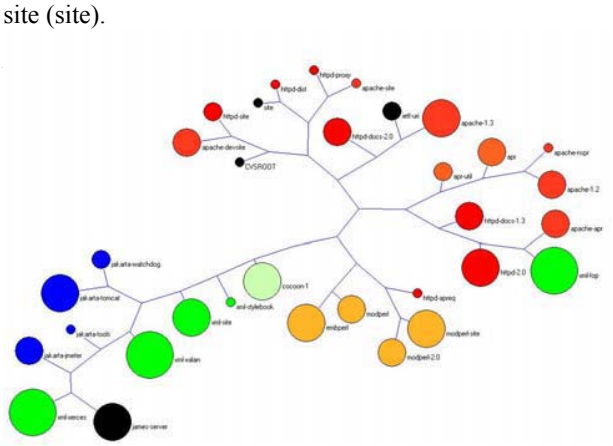
The network which we use to identify communities is built as follows. From the CVS repository we extract, for each module, the list of people committing changes to it, and the number of of those changes (commits) [13]. With this list we compute, for each pair of modules, the set of committers who have collaborated in both modules and the number of commits they have contributed to them. Using these data, we can build a network by considering modules as vertices, and establishing a link among any two vertices if the corresponding modules have common committers. We use wheighted edges, with the wheight being proportional (although not linearly) to the number of commits that common authors have contributed[2].

To obtain the community structure of the resulting network, we perform community analysis based on the Girvan-Newman (GN) algorithm [14] which has been proved to be one of the most accurate[3] (see Figure A-1), and has already been successfully used to obtain the informal collaboration network of organizations (showing that it is self-similar [17]). The GN algorithm produces as output a minimal binary expanding tree, in which we have used colored[4] vertices (modules) to show visually how the extracted communities follow a certain pattern. The size of a vertex is proportional to the size of the module in LOC (lines of code), so that bigger modules can be identified visually.

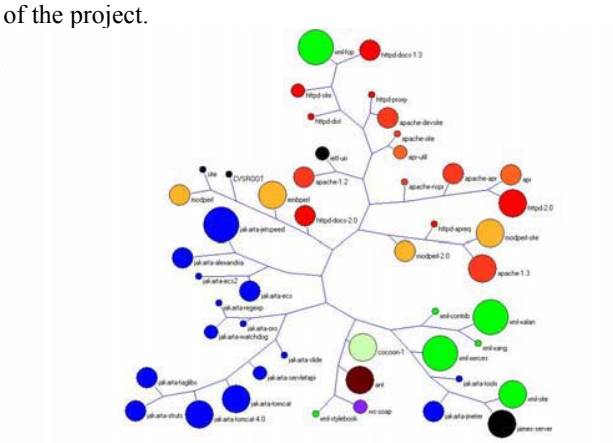
3. A case study: the Apache project



We have applied the described methodology to the Apache project CVS repository. To color the nodes, we have followed in general the naming conventions of the project, which correspond to families of modules. That way, we have used red for those modules related to the HTTP server (apache/httpd), orange for those related to modperl, green for XML related modules, dark blue for Jakarta, light blue for Avalon, yellow for TCL-related modules and black for generic modules like CVSROOT (administrative files of the CVS repository) or the web site (site).



Thanks to the coloring, it can be observed how the project self-structures itself. For instance, in January, 1999 (Figure 1), modules related to modperl are seen only in the upper-left branch, which means that those modules have already some differences in authorship with the rest of the project.



Later, in January 2000 (Figure 2), the community of modperl modules is also clearly seen as a branch (in the center of the figure). Meanwhile, a branch for Jakarta and XML-related modules can be identified (left bottom). Nine months later (Figure 3) Jakarta has differentiated itself (two close branches at the bottom of the figure) from XML-related modules (branch at the bottom right).

[2] nber of aggregated LOC (lines of code) committed by the common developes instead of the number of commits. However, in the cases we have studied the resulting networks are very similar.

[3] Other different alternatives proposed in the literature for obtaining the community structure of a social network can be seen in [15] [16].

[4] There exists a colored on-line version of this paper at http://libresoft.dat.escet.urjc.es/html/downloads/woss-icse-2004.pdf

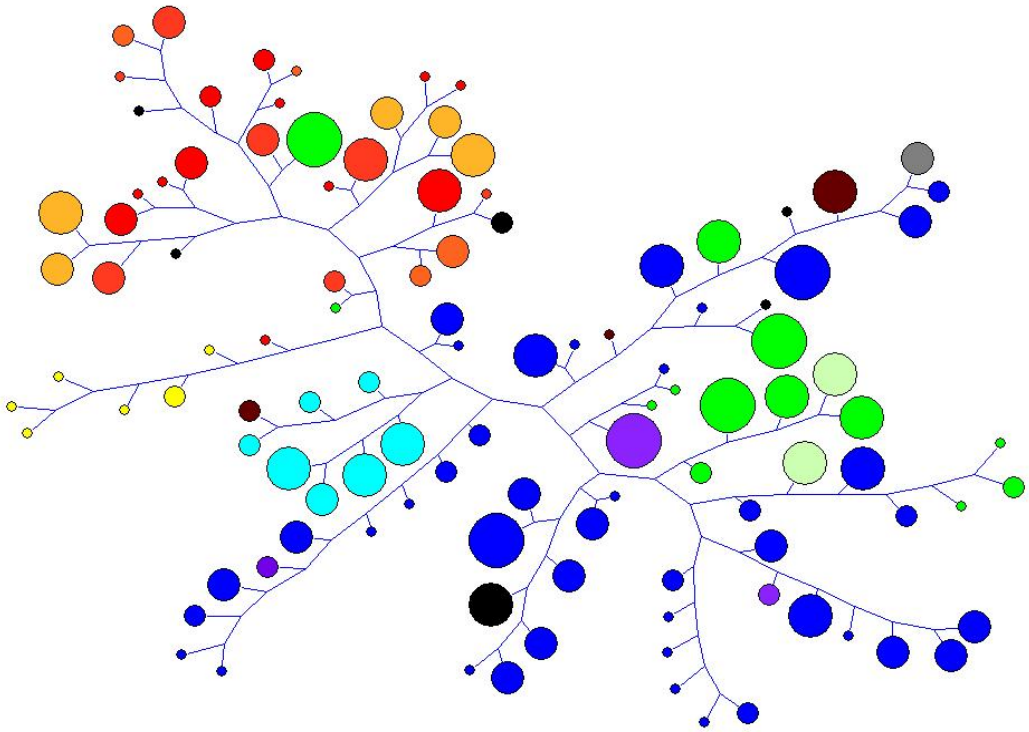


Illustration 4 Community structure of the Apache project. January 1st, 2002

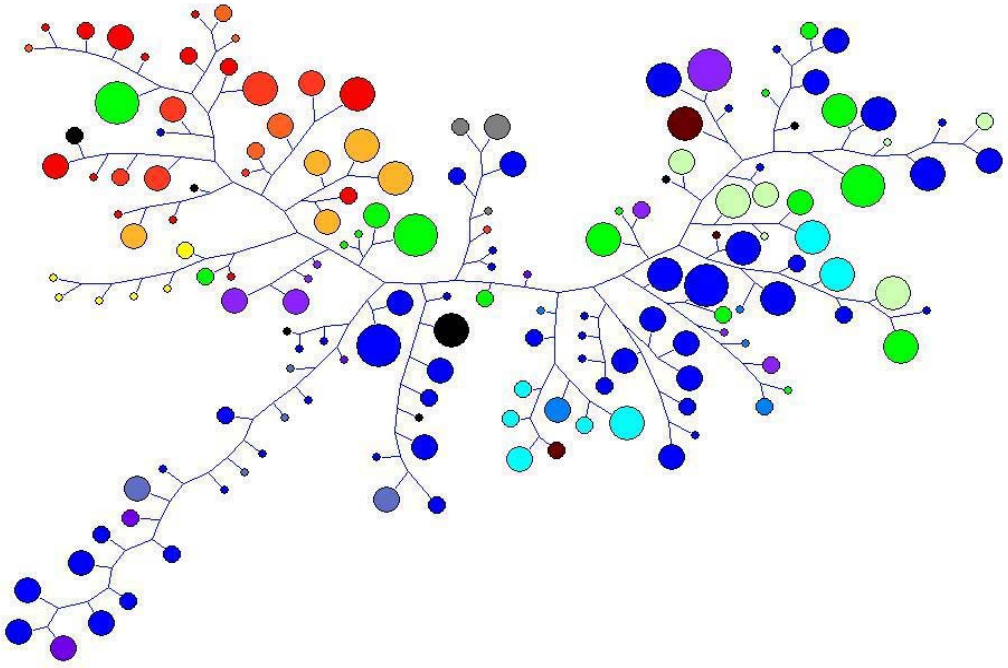


Illustration 5 Community structure of the Apache project. February 1st, 2004

On the other hand, modperl modules are now spread through several branches, mixed with the apache/httpd modules (top part of the figure).

The figures in early 2002 and 2004 (Figure 4 and Figure 5) show how modperl is still somewhat spread through the project, while there are clear communities (branches) for the apache/httpd (red, not too split in large branches), Avalon (light blue, clearly differentiated as a community), Jakarta (dark blue, present in several different communities, intermixed with green, XML-related modules) and Tcl related modules (yellow community).

4. Conclusions and further work

In the Apache project, the structure seen after several years of development is rather similar to that in early 1999, which seems to be the epoch in which it stabilized. It is interesting to see how, even when the project grows a lot during that time, the conservation of the structure means that the pairs of nodes related by the developers working on them remain more or less the same. In addition, it is clear that developers tend to work in the projects within its thematic community (apache/httpd, Jakarta, TCL, Avalon, etc.), but even within these communities there are clear differentiations (see for instance the many large branches of the Jakarta-XML modules).

One of the more impacting cases that come into view through the evolution of the structure of the Apache project is how apache/httpd-related modules (red) and Jakarta-related modules (blue) represent separated communities in the computed binary trees, in all dates during the life of the project. Of course, technological aspects are a key fact for understanding this (Jakarta is mainly based in Java, while apache/httpd uses basically C).

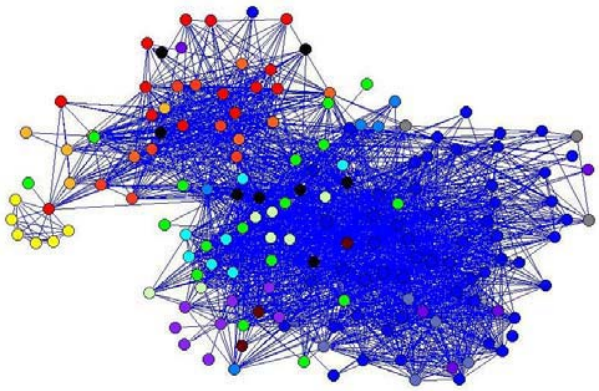


Illustration 6 "Classical" network analysis. Feb 1st 2004

Regarding our methodology, we believe this technique can be used to obtain and compare the community structure of modules in large libre software projects. From this structure, it can be inferred whether there are or not strong communities within the projects, the evolution of these communities, and their interrelationships. These implications can be used to estimate information flow within the project, collaboration patterns, and other important characteristics of the project.

In comparison to previous works, such as [10] this view is a step forward at least in the aspect of offering a way to visually identify an affiliation network. Such a classical network analysis would offer a graph as in Figure 6, where the structure can be hardly infered.

Inspite of the findings in [12] this analysis does not show an effective splitting of modules when the size of the main contributing group exceeds a certain number of persons. In fact, two other behaviours can be deduced: the emergence of new modules around consolidated ones and the specialization of developers in time (which can be can be derivated by the fact that a branch splits in two).

This paper offers also a first dynamic vision of the evolution of the structure, although future work on this issue is pending, specially regarding relationships between the different graphs in time.

A. The Girvan-Newman algorithm

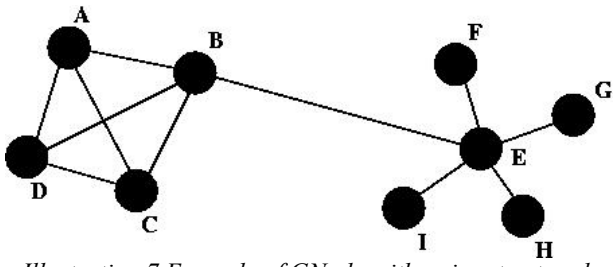


Illustration 7 Example of GN algorithm: input network

When we apply to the network in Figure 7 the GN algorithm, we get as output the tree in Figure 8, which represents its community structure. The GN algorithm proceeds by splitting the network recursively until single nodes are left. The information about the community structure of the original network can be deduced from the topology of the binary tree that represents this splitting procedure. Basically, each branch of this tree can be seen as a community of nodes of the original network.

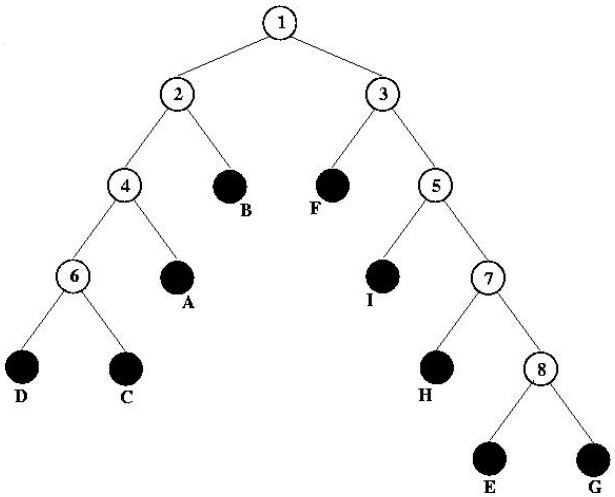


Illustration 9 Example of GN algorithm: output network

The GN algorithm is based on a parameter called the betweenness of edges, which measures the number of shortest paths connecting pairs of nodes which go through that edge [5]. Edges connecting high clustered communities must have higher betweenness. So, the GN algorithm proceeds by calculating the betweenness of all edges and eliminating the edge with the highest betweenness. These steps are repeated until the network is split into two connected components. After that, the process is recursively executed on each of the two separate components. To represent the splitting process, we use a binary tree which is built in the following way: each time a split is carried out, we add a virtual node in the tree (marked with numbers in Figure A-2) which is connected to the two novel components. When these components are single vertices of the original network, we add them as terminal nodes of the tree (marked with letters in Figure A-2).

10. References

[1] D.J. Watts and S.H. Strogatz, Collective dynamics of small-world networks, Nature 393, 440-442, 1998 .

[2] R. Albert, A.-L. Barabasi, H. Jeong, and G. Bianconi, Power-law distribution of the World Wide Web, Science 287 2115a (2000).

[3] R.F. Cancho and R. Sole, The small world of human languaje, Proc. R. Soc. 268, 2261-2265, 2001.

[4] R. Kunar, P. Raghavan, S. Rajagopalan, and A. Tomkins, The Web and social networks, IEEE Computer 35(11) 32-36, 2002.

[5] M.E.J. Newman, Scientific collaboration networks: I. Network construction and fundamental results, Phys. Rev. E 64, 016131 (2001).

[6] M.E.J. Newman, Scientific collaboration networks: II. Shortest paths, weighted networks, and centrality, Phys. Rev. E 64, 016132, 2001.

[7] R. Guimera, A. Daz-Aguilera, F. Vega-Redondo, A. Cabrales, and A. Arenas, Optimal network topologies for local search with congestion, Phys. Rev. Let. 89, 248701, 2002.

[8] L. Lopez and M.A.F. Sanjuan, Relation between structure and size in social networks, Phys. Rev. E. 65, 036107, 2002.

[9] L. Lopez, J.F. Mendes, and M.A.F. Sanjuan, Hierarchical social networks and information flow, Physica A, 316, 591-604, 2002.

[10] Greg Madey, V. Freeh, and R. Tynan, The Open Source Software Development Phenomenon: An Analysis Based on Social Network Theory, 2002, http://www.nd.edu/~oss/Papers/amcis_oss.pdf.

[11] Rishab Aiyer Ghosh, Clustering and dependencies in free/open source software development: Methodology and tools, Abril 2003, http://www.firstmonday.dk/issues/issue8_4/ghosh/index.html.

[12] Audris Mockus, Roy T. Fielding, and James D. Herbsleb, Two Case Studies of Open Source Software Development: Apache and Mozilla, http://www.research.avayalabs.com/techreport/ALR-2002-003-paper.pdf.

[13] Gregorio Robles, Jesús González-Barahona, José Centeno-González, Vicente Matellán-Olivera, and Luis Rodero-Merino. http://opensource.ucc.ie/icse2003/3rd-WS-on-OSS-Engineering.pdf.

[14] M. Girvan and M. E. J. Newman, Community structure in social and biological networks, Proc. Natl. Acad. Sci. USA 99, 7821-7826, 2002.

[15] M.E.J. Newman, Detecting community structure in networks., Eur. Phys. J. B, in press. http://www.santafe.edu/~mark/pubs.html.

[16] M.E.J. Newman and M. Girvan, Finding and evaluating community structure in networks., Phys. Rev. E, in press. http://www.santafe.edu/~mark/pubs.html.

[17] R. Guimera, L. Danon, A. Daz-Aguilera, F. Giralt, and A. Arenas, Self-similar community structure in organizations, http://arxiv.org/pdf/cond-mat/0211498.

Collaboration, Leadership, Control, and Conflict Negotiation in the Netbeans.org Community

Chris Jensen and Walt Scacchi
Institute for Software Research
School of Information and Computer Science
University of California, Irvine
Irvine, CA, USA 92697-3425
{cjensen, wscacchi}@ics.uci.edu

Abstract

Large open source software development communities are quickly learning that, to be successful, they must integrate efforts not only among the organizations investing developers within the community and unaffiliated volunteer contributors, but also negotiate relationships with external groups hoping to sway the social and technical direction of the community and its products. Leadership and control sharing across organizations and individuals in and between communities are common sources of conflict. Such conflict often leads to breakdowns in collaboration. This paper seeks to explore the negotiation of these conflicts, collaborative efforts, and leadership and control structures in the Netbeans.org community.

Keywords

Collaboration, Conflict Negotiation, Leadership, Process, Open Source Software Development, Netbeans.org

1. Introduction

Is open source software development (OSSD)best characterized as being strictly cooperative, or as cooperative and in conflict at the same time [Easterbrook 1993]? Conflict clearly arises during sustained software development efforts [e.g., Sawyer 2001]. But previous studies of conflict associated with Internet-based communities has focused attention to that found in specific OSSD projects operating as virtual organizations [Elliott and Scacchi 2003], as non-profit foundations [O'Mahony 2004], or in online discussion communities [Smith 1999]. None of these studies specifically help us understand the kinds of conflict, cooperation, and collaboration that arises or is needed to coordinate large-scale OSSD processes and effort in large project communities where corporate sponsorship may be a central facet of OSSD.

NetBeans.org is one of the largest OSSD communities around these days [Jensen and Scacchi 2003]. Netbeans.org is a Java-focused OSSD community backed by Sun Microsystems devoted to creating both an integrated development environment (IDE) for developing large Java-based applications, as well as a platform for development of other software products. Originally started as a student project in 1996, the Netbeans.org project was acquired and subsequently released as an open source community project by Sun, whose Netbeans.org team includes many of the community's core developers. While the issues presented here stem from observations in the Netbeans.org community, they are by no means limited to this community, nor have their challenges been insurmountable.

Our discussion is two tiered. First, we explore relationships arising in NetBeans.org on an intra-community level. Then, we look at relationships between communities like Netbeans.org and other communities and organizations.

2. Intra-Community Issues

At least three kinds of issues arise *within* OSSD communities. These are collaboration, leadership and control, and conflict.

2.1. Collaboration

According to the Netbeans.org community Web site, interested individuals may participate in the community by joining in discussions on mailing lists, filing bug and enhancement reports,

contributing Web content, source code, newsletter articles, and language translations. These activities can be done in isolation, without coordinating with other community members, and then offered up for consideration and inclusion. As we'll see, reducing the need for collaboration is a common practice in the community that gives rise to positive and negative effects. We discuss collaboration in terms of policies that support process structures that prevent conflict, looking at task completion guidelines and community architecture.

2.1.1. Policies and Guidelines

The NetBeans.org community has detailed procedural guidelines[1] for most common development tasks, from submitting bug fixes to user interface design and creating a new release. These guidelines come in two flavors: development task and design style guidelines. In general, these policies are practiced and followed without question. Ironically, the procedures for policy revision have not been specified.

Precedent states that revisions are brought up on the community or module discussion mailing lists, where they are debated and either ratified or rejected by consensus. Developers are expected to take notice of the decision and act accordingly, while the requisite guideline documents are updated to reflect the changes. In addition, as some communities resort to "public flogging" for failure to follow stated procedures, requests for revision are rare and usually well known among concerned parties, so no such flogging is done within Netbeans.org.

Overall, these policies allow individual developers to work independently within a process structure that enables collaboration by encouraging or reinforcing developers to work in ways that are expected by their fellow community members, as well as congruent with the community process.

2.1.2. Separation of Concerns: an Architectural Strategy for Collaborative Success

Software products are increasingly developing a modular, plug-in application program interface (API) architectural style in order to facilitate development of add-on components that extend system functionality. This strategy has been essential in an open source arena that carries freedom of extensibility as a basic privilege or, in some cases, the right of free speech or freedom of expression through contributed source code. But this separation of concerns strategy for code management also provides a degree of separation of concerns in developer management, and therefore, collaboration.

In concept, a module team can take the plug-in API specification and develop a modular extension for the system using any development process in complete isolation from the rest of the community. This ability is very attractive to third-party contributors in the Netbeans.org community who may be uninterested in becoming involved with the technical and socio-political issues of the community, or who are unwilling or unable to contribute their source code back to the community. Thus, this separation of concerns in the Netbeans.org design architecture engenders separation of concerns in the process architecture. Of course, this is limited by the extent that each module in the Netbeans.org community is dependent on other modules.

Last, volunteer community members have periodically observed difficulties collaborating with volunteer community members. For example, at one point a lack of responsiveness of the (primarily Sun employed) user interface team[2], whose influence spans the entire community, could be observed. This coordination breakdown led to the monumental failure of usability efforts for a period when usability was arguably the most-cited reason users chose competing tools over Netbeans.org. Thus, a collaboration failure gave rise to product failure. Only by overcoming collaboration issues was Netbeans.org able to deliver a satisfactory usability experience[3].

2.2. Leadership and Control

Ignoring internal Sun (and third party) enterprise structure, there are five observable layers of the Netbeans.org community hierarchy. Members may take on multiple roles some of which span several of these layers. At the bottom layer are users, followed by source contributors, module-level managers, project level release managers (i.e. IDE *or* platform), and finally, community level managers (i.e. IDE *and* platform) at the top-most layer. Interestingly, the "management" positions are simply limited to coordinating roles; they carry no other technical or

1 http://www.netbeans.org.org/community/guidelines/

2 http://www.netbeans.org.org/servlets/ReadMsg?msgId=531512&listName=nbdiscuss

3 http://www.javalobby.org/thread.jspa?forumID=61&threadID=9550#top

managerial authority. The release manager, for example, has no authority to determine what will be included in and excluded from the release[4]. Nor does s/he have the authority to assign people to complete the tasks required to release the product. The same is true of module and community managers. Instead, their role is to announce the tasks that need to be done and wait for volunteers to accept responsibility.

Accountability and expectations of responsibility are based solely on precedent and volunteerism rather than explicit assignment, leading to confusion of the role of parties contributing to development. Leadership is not asserted until a community member champions a cause and while volunteerism is expected, this expectation is not always obvious. The lack of a clear authority structure is both a cause of freedom and chaos in open source development. Though often seen as one of its strengths in comparison to closed source efforts, it can lead to process failure if no one steps forward to perform critical activities or if misidentified expectations cause dissent.

The difficulties in collaboration across organizations within the community occasionally brought up in the community mailing lists stem from the lack of a shared understanding leadership in the community. This manifests itself in two ways: a lack of transparency in the decision making process and decision making without community consent. While not new phenomenon, they are especially poignant in a movement whose basic tenets include freedom and knowledge sharing.

2.2.1. Transparency in the Decision Making Process

In communities with a corporately backed core development effort, there are often decisions made that create a community-wide impact that are made company meetings. However, these decisions may not be explicitly communicated to the rest of the community. Likewise private communication between parties that is not made available on the community Web space or to the forwarded to other members is also hidden. This lack of transparency in decision-making process makes it difficult for other community members to understand and comply with the changes taking place if they are not questioned or rejected. This effect surfaced in the Netbeans.org community recently following a discussion of modifying the release process [cf. Erenkrantz 2003][5].

Given the magnitude of contributions from the primary benefactor, other developers were unsure of the responsibility and authority Sun assumed within the development process. The lack of a clearly stated policy outlining these bounds led to a flurry of excitement when Sun members announced major changes to the licensing scheme used by the community without any warning. It has also caused occasional collaboration breakdown throughout the community due to expectations of who would carry out which development tasks. The otherwise implicit nature of Sun's contributions in relation to other organizations and individuals has been revealed primarily through precedent rather than assertion.

2.2.2. Consent in the Decision Making Process

Without an authority structure, all decisions in development are done through consensus, except among those lacking transparency. In the case of the licensing scheme change, some developers felt that Sun was within its rights as the major contributor and the most exposed to legal threat [6] while others saw it as an attack on the "democratic protection mechanisms" of the community that ensure fairness between participating parties[7]. A lack of consideration and transparency in the decision making process tend to alienate those who are not consulted and erode the sense of community.

2.3. Conflict Resolution

Conflicts in the Netbeans.org community are resolved via community discussion mailing lists. The process usually begins when one member announces dissatisfaction with an issue in development. Those who also feel concern with the particular issue then write responses to the charges raised. At some point, the conversation dissipates- usually when emotions are set aside and clarifications have been made that provide an understanding of the issue at hand. If the problem persists, the community governance board is tasked with the responsibility of resolving the matter.

4 http://www.netbeans.org/community/guidelines/process.html

5 http://www.netbeans.org/servlets/BrowseList?listName=nbdiscuss&by=thread&from=19116&to=19116&first=1&count=41

6 http://www.netbeans.org/servlets/ReadMsg?msgId=534707&listName=nbdiscuss

7 http://www.netbeans.org/servlets/ReadMsg?msgId=534520&listName=nbdiscuss

The governance board is composed of three individuals and has the role of ensuring the fairness throughout the community by solving persistent disputes. Two of the members are elected by the community, and one is appointed by Sun Microsystems. The board is, historically, a largely superficial entity whose authority and scope are questionable and untested. While it has been suggested that the board intercede on a few rare occasions, the disputes have dissolved before the board has acted. Nevertheless, board elections are dutifully held every six months[8].

Board members are typically prominent members in the community. Their status carries somewhat more weight in community policy discussions, however, even when one member has suggested a decision, as no three board members have ever voted in resolution on any issue, and thus, it is unclear what effect would result. Their role, then, is more of a mediator: to drive community members to resolve the issue amongst themselves. To this end, they have been effective.

3. Inter-Community Issues

At least three kinds of issues arise *across* OSSD communities. These are communication and collaboration, leadership and control, and conflict resolution.

3.1. Communication and Collaboration

In addition to their IDE, Netbeans.org also releases a general application development platform on which the IDE is based. Other organizations, such as BioBeans and RefactorIT communities build tools on top of or extending the NetBeans platform or IDE. How do these organizations interact with Netbeans.org, and how does Netbeans.org interact with other IDE and platform producing organizations? For some organizations, this collaboration may occur in terms of bug reports and feature requests submitted to the Netbeans.org issue-tracking repository. Additionally, they may also submit patches or participate in discussions on community mailing list or participate in the Netbeans.org "Move the Needle" branding initiative. Beyond this, Netbeans.org participates in the Sun sponsored *Java.net* meta-community, which hosts hundreds of Java-based OSSD projects developed by tens of thousands of individuals and organizations.

A fellow member of the Java.net community, the Java Tools Community, considered by some to be a working group[9] for the Java Community Process, is an attempt to bring tool developers together to form standards for tool interoperability. Thus Netbeans.org, through its relationship with Sun, is a collaborating community in the development of, and through compliance with, these standards, and looks to increasing collaboration with other tool developing organizations.

3.2. Leadership and Control

OSSD generally embrace the notion of choice between software products to build or use. At the same time, developers in any community seek success for their community, which translates to market share.

In some cases, communities developing alternative tools do so in peaceful coexistence, even collaboratively. In other cases, there is a greater sense of competition between rivals. Netbeans and its chief competitor Eclipse (backed largely by IBM) fall into the latter category. Eclipse has enjoyed some favor from users due to performance and usability issues of Netbeans, as well as IBM's significant marketing and development resource contributions. Yet, they have a willingness to consider collaborative efforts to satisfy demands for a single, unified IDE for the Java language that would serve as a platform for building Java development tools and a formidable competitor to Microsoft's .NET. Ultimately, the union was defeated, largely due to technical and organizational differences between Sun and IBM[10], including the inability or unqillingness to determine how to integrate the architectures and code bases for their respective user interface development frameworks (Swing for NetBeans and XXX for Eclipse).

3.3. Conflict Resolution

Conflicts between collaborating communities are resolved in similar fashion to their means of communication- through discussion between Sun and Eclipse representatives, comments on the Netbeans.org mailing lists, or other prominent technical forums (e.g. Slashdot and developer blogs). Unfortunately, many of these discussions

8 http://www.netbeans.org.org/about/os/who-board.html

9 http://www.internetnews.com/dev-news/article.php/3295991

10 http://www.adtmag.com/article.asp?id=8634, and http://www.eweek.com/article2/0,1759,1460110,00.asp

occur after the collaborating developer has moved away from using Netbeans.org (often, in favor of Eclipse). Nevertheless, the feedback they provide gives both parties an opportunity to increase understanding and assists the Netbeans.org community by guiding their technical direction.

4. Discussion

Generally, volunteer Netbeans.org developers expect Sun to provide leadership but not control. People outside the community (e.g. users, former users, and potential users) often voice their concerns in off-community forums (e.g., Slashdot, blogs, etc) rather than NetBeans.org community message boards, due to accountability or visibility barriers (creating an account, logging in accounts), small as they may seem to be. In addition, such message forums may not be a part of such an individual's daily work habits- they're more likely to visit a site like Slashdot.org than the Netbeans.org forum because they are not interested enough in staying abreast of Netbeans developments or participating in the community. Nonetheless, people working in, or interested in joining or studying OSSd projects, must address how best to communicate and collaborate their development processes and effort, how to facilitate or ignore project leadership and control, and how to work you way through conflicts that may or may not be resolvable by community participants.

Overall, we have observed three kinds of coordination and collaborating issues arise within OSSD project communities, and three similar kinds of issues arise across OSSD communities. Previous studies of conflict in either OSSD projects have examined either smaller projects, or in virtual communities that do not per se develop software as their focus. As corporate interest and sponsorship of OSSD stimulates the formation of large projects, or else the consolidation of many smaller OSSD projects into some sort of for-profit or not-for-profit corporate enterprise for large-scale OSSD, then we will need to better understand issues of collaboration, conflict, and control in OSSD.

5. Acknowledgments

The research described in this report is supported by grants from the National Science Foundation #ITR-0083075 and #ITR-0205679 and #ITR-0205724. No endorsement implied. Contributors to work described in this paper include Mark Ackerman at the University of Michigan Ann Arbor; Les Gasser at the University of Illinois, Urbana-Champaign; John Noll at Santa Clara University; and Margaret Elliott at the UCI Institute for Software Research.

6. References

Easterbrook, S. (ed.), *CSCW: Cooperation or Conflict*, Springer-Verlag, New York, 1993.

Elliott, M. The Virtual Organizational Culture of a Free Software Development Community, *Proc. 3rd Workshop on Open Source Software Engineering, 25th. Intern. Conf. Software Engineering*, Portland, OR, May 2003.

Elliott, M. and Scacchi, W., Free Software Developers as an Occupational Community: Resolving Conflicts and Fostering Collaboration, *Proc. ACM Intern. Conf. Supporting Group Work*, 21-30, Sanibel Island, FL, November 2003.

Erenkrantz, J., Release Management within Open Source Projects, *Proc. 3rd Workshop on Open Source Software Engineering, 25th Intern. Conf. Software Engineering*, Portland, OR, May 2003.

Jensen, C. and Scacchi, W., Automating the Discovery and Modeling of Open Source Software Processes, *Proc. 3rd Workshop on Open Source Software Engineering*, 25th. Intern. Conf. Software Engineering, Portland, OR, May 2003.

O'Mahony, S., Non-profit Foundations and their Role in Community-Firm Software Collaboration, to appear in *Making Sense of the Bazaar: Perspectives on Open Source and Free Softwar*e, J. Feller, B. Fitzgerald, S. Hissam, & K. Lakhani (Eds.), O'Reilly & Associates, Sebastopol, CA, 2004.

Sawyer, S., Effects of intra-group conflict on packaged software development team performance, *Information Systems J.,* 11, 155-178, 2001.

Scacchi, W., Free/Open Source Software Development Practices in the Computer Game Community, *IEEE Software*, 21(1), 59-67, January/February 2004.

Smith, A.D., Problems of Conflict Management in Virtual Communities. In M.A. Smith and P. Kollock (eds.), *Communities in Cyberspace,* Routledge, Ney York, 134-163, 1999.

WebDAV based Open Source Collaborative Development Environment

Sunghun Kim, Kai Pan, E. James Whitehead, Jr.
Dept. of Computer Science
University of California, Santa Cruz
Santa Cruz, CA 95064 USA
{hunkim, pankai, ejw}@cs.ucsc.edu

Abstract

Open source projects are characterized by their distributed developers, openness, and use of a community-based development process. Successful open source projects such as Apache, Linux, and PHP have a web-based Collaborative Development Environment (CDE) that provides source code repository access, knowledge management, discussion lists, bug tracking, and user support. Distributed software development is also increasingly common for commercial software projects, and benefits from using web-based CDEs. GForge, derived from the SourceForge package, is an open web-based CDE that is seeing widespread open source and commercial use. To support remote authoring of web pages and file contents, as well as remote source code access, GForge uses several network protocols, including SSH/SFTP, CVS pserver, and FTP. These protocols inevitably introduce security risks and add administrative work. We modified GForge to eliminate use of FTP, SSH/SFTP, and CVS protocols, using instead a WebDAV-based integration. The new CDE, called Davenport, uses only the HTTP/WebDAV protocol, and HTTP/WebDAV replaces the other protocols without sacrificing functionality. In this paper we describe the architecture and features of Davenport, and show how WebDAV replaces existing network protocols. We discuss the future development directions of CDEs including integration of client side IDEs and using a CDE as a software test bed.

1. Introduction

Most successful open source projects are community-based, consisting of groups such as lead developers, committers, bug reporters, and users [1]. The community members work in a distributed environment, and have few, if any, face-to-face meetings. One key challenge of distributed development is communication among developers; one study reports that distributed work takes twice as long as collocated work due to communication issues [2].

To facilitate communication among the distributed developers in open source projects, various software development and collaborative tools are required. Software Configuration Management (SCM) tools such as CVS [3] are necessary to control source code, allow changes to be made in isolation from other developers, and raise awareness of source code changes. To share ideas and store project knowledge, a discussion forum or mailing list is useful. A bug or issue tracking system is necessary to keep track of defects or new issues. Metrics such as download and page visit counters are helpful to estimate the number of users and popularity of the project. To advertise the project and to support users, a project homepage is essential.

Many open source project have a web-based, portal-style collaborative development environment (CDE) which integrates project tools. These web-based CDEs are a key part of active open source projects. For example, Apache [4] and PHP [5] projects have web-based CDEs which include CVS, a CVS viewer, a discussion forum, a mailing list, a bug tracking system, and a project homepage. The success of SourceForge [6], a web-based CDE hosting site, also highlights the importance of CDEs.

Commercial software development projects sometimes involve globally distributed teams, and can benefit from web-based CDEs. Several SourceForge-like CDEs have been developed to support distributed commercial projects, such as Source Cast [7], SourceForge Enterprise edition [8, 9], and GForge [10].

GForge provides a range of typical CDE services, including a project homepage, discussion forum, mailing list, user access control, bug tracking, a file release system, and CVS [10]. GForge is widely used in open source communities and software development companies including the Helix Community, Korean Linux document project, Philips, Thompson Financial, and NASA Goddard Space Flight Center [10].

To support CVS, homepage authoring, and the file release system, GForge makes use of several network protocols such as LDAP, FTP, SSH/SFTP, CVS pserver as well as HTTP. It raises security risks and administrative burdens.

The use of multiple protocols means that GForge must have multiple protocol servers running simultaneously. Each server is a separate code base, with its potential for security exploits. Hence, the more servers used by a CDE, the higher the chance for security holes. User login via SSH is especially risky due to the ability to have direct file system access, and the risk of users elevating their privilege illegally after system login [11].

Running several protocol handlers and servers requires additional administrative effort. An administrator must check if all servers are working well and ensure the services are running normally. If there are two or more data repositories that must be synchronized, e.g. DBMS and LDAP, it increases administrative work significantly. When these repositories are out of sync or fail to work, the administrator has to manually synchronize them. Additionally, the administrator must backup all repositories, and making consistent backups is a challenge across multiple repositories.

Collecting metrics requires more effort in a multi-server environment. For example the log files for SSH/SFTP, FTP, CVS, and HTTP are in different formats and in different locations. As a result, gathering metrics from all protocol handlers or servers requires a special log parser and analyzer for each of the log formats.

It would be beneficial to have all remote accesses to the CDE performed over one protocol with one administration interface, while eliminating the system login privileges to access the CDE server. Luckily such a solution exists today: the WebDAV protocol suite [12].

WebDAV is an extension to HTTP that enables distributed authoring and versioning [12]. WebDAV can replace FTP, SSH/SFTP, and CVS protocols, allowing CDEs to support their normal remote authoring and source code access functions using only HTTP/WebDAV.

WebDAV provides a superior webpage authoring mechanism as compared to direct file editing using SSH and a text editor, and eliminates the need for shell access to the CDE. Several well-known web content editors, including GoLive and DreamWeaver, already support WebDAV, and various WebDAV clients are available. File transfer using HTTP/WebDAV is trivial, and all major operating systems have native WebDAV drag and drop file upload/download capability. Subversion, a WebDAV based SCM, can be used as the SCM repository without adding a new server by configuring the Apache web server [13]. The unified and congregated HTTP log for homepage access, web authoring, file transfer, SCM usage, and CDE usage enables a CDE to provide abundant and detailed data to calculate metrics.

In this paper, we describe the architecture and features of Davenport, our WebDAV/Subversion-enabled version of GForge. We show how WebDAV can replace other protocols used in GForge without sacrificing existing functionality.

2. Rationale of WebDAV Unification

In this section we discuss the rationale of using WebDAV on top of GForge.

2.1. GForge

GForge uses a database to store most project information, including users, forums, bug tracking, and document data. The web interface is written in PHP [5], and the PHP functions interact with the database, Apache web server, and LDAP server. The simplified GForge architecture is depicted in figure 1.

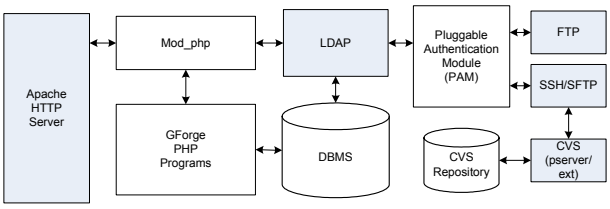


Figure 1. GForge Architecture. Protocol handlers are shown in gray.

To support homepage authoring and the file release system, GForge supports FTP and SSH/SFTP protocols. These protocols use a programmable authentication module with LDAP so that information authentication is performed by checking LDAP database instead of the /etc/passwd file for valid user id and password.

The CVS repository can be accessed through SSH tunneling using the same user id and password as for straight SSH, and CVS pserver is used to allow anonymous CVS access. In the backend, there are several UNIX crontab jobs synchronizing the database and LDAP, creating new CVS modules, creating group directories, gathering statistics data, etc.

As we have shown above, GForge uses several protocols and servers to provide its services. If there is a protocol that can handle all the capabilities provided by HTTP, SSH/SFTP, FTP, and CVS in GForge, it would reduce administrative work and security risks. WebDAV is such a protocol. We introduce WebDAV and compare it with other protocols in the next section.

2.2. WebDAV

WebDAV is a suite of protocol extensions to HTTP/1.1 [14] that supports collaborative authoring, namespace management of resources, setting and retrieving of metadata properties, access control, and resource versioning on remote web servers [12, 14]. The suite includes the DeltaV (versioning) [15], DASL (searching) [16], ACL (access control) [17], and WebDAV core (authoring) [12] protocols.

The WebDAV core protocol (RFC2518) enables users to create, modify, and delete resources or collections on a WebDAV server. WebDAV's concurrency control mechanism, whole resource locking, prevents overwrite conflicts during authoring. DeltaV is a versioning and configuration management protocol, and Subversion [13] uses WebDAV/DeltaV as a transport protocol. For a CDE like GForge, WebDAV can replace SSH/SFTP and FTP,

since it supports remote resource authoring functionality. CVS also can be replaced by Subversion, a version control system based on the WebDAV/DeltaV protocol [13]. The overview comparison between WebDAV and other protocols is shown in table 1.

Table 1. Protocol Feature Comparison

Feature	FTP	SSH	CVS	WebDAV
remote web authoring		✓		✓
remote file upload	✓			✓
versioning			✓	✓
Concurrency control (versioning)			✓	✓
concurrency control (web authoring)				✓
Secure connection		✓		✓

2.3. WebDAV vs SSH/SFTP/FTP

For homepage authoring and file transfer, GForge uses SSH/SFTP and FTP protocols. This practice is susceptible to security attacks. For example, several years ago, Apache Software Foundation and SourceForge servers were compromised due to a weakness in the SSH daemon [11]. Once an intruder achieves shell access, it is possible to illegally elevate privileges using the weaknesses of operating systems.

Let's consider a scenario, in which developers want to edit some project HTML files on a GForge server. With the SSH support in GForge, developers login to the server, and use a text editor on the server to edit the HTML files. In this case, it is hard to use specialized HTML editors such as GoLive or DreamWeaver. An alternative way is to download HTML files onto each developer's local disk, edit them using a HTML editor, and then transfer the changed HTML files to the server using SFTP or FTP. This process requires inconvenient extra work, and it can also introduce the lost-update problem when several developers transfer their data simultaneously to the server.

When using WebDAV instead of SSH/SFTP and FTP, these problems go away. WebDAV has a well-defined concurrency control mechanism: lock and unlock. There are already many WebDAV-compliant HTML editors such as DreamWeaver, GoLive and Adobe Photoshop. Windows, Linux, and Mac OS support WebDAV at the filesystem level, so developers can even mount a server-side Davenport homepage area as a local disk.

Overall, WebDAV provides superior homepage authoring, and it can replaces SSH/SFTP and FTP in GForge, so that the administrative tasks and security risks are reduced.

2.4. Subversion vs CVS

The Subversion project was started to improve the functionality and fix the shortcomings of CVS. Use of Subversion confers many benefits as compared to CVS, including directory versioning, better concurrency control, and property support [13]. Subversion also addresses a long standing CVS issue: resource renaming in CVS causes loss of change histories. That is, when we change a resource's name or move a resource from one directory to another, CVS loses the version history of the resource, while Subversion keeps all history of renamed or moved resources. Subversion also supports data import and export functionalities like those of a DBMS.

Subversion eliminates the need for a specific protocol for SCM, such as CVS pserver or SSH tunneling. Subversion uses the WebDAV/DeltaV protocol, so it is possible to reuse the existing HTTP server in Davenport.

3. Davenport

In this section, we introduce the architecture of our WebDAV integration, Davenport. The Davenport architecture is shown in Figure 2. The additional features are also described.

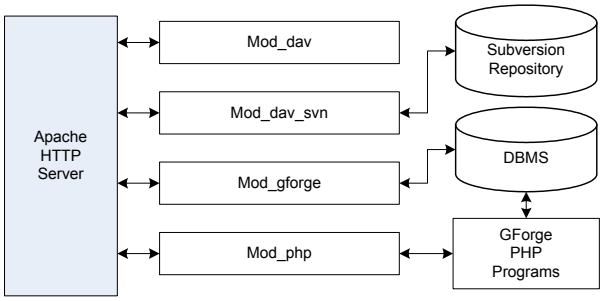


Figure 2. Davenport Architecture. The protocol handler, (Apache HTTP server), is shown in gray.

3.1. Architecture

We added the WebDAV integration using mod_dav [18] so that developers can author the project homepage using the WebDAV protocol rather than using SSH/SFTP or FTP. We also added a Subversion integration using mod_dav_svn [13].

For homepage authoring or source code check-in, a suitable permission is required. For example, only the member of a group should be able to change the source code of the group. To support HTTP authentication for WebDAV and Subversion, we wrote an Apache authentication module, mod_gforge, which retrieves user id/password information from a web browser, and compares it with the id/password record in the database to verify the user authentication.

Mod_dav, mod_dav_svn, and mod_gforge enable Davenport to replace SSH/SFTP, FTP, CVS and LDAP. Davenport uses the Apache HTTPD server for the web service, WebDAV, and Subversion.

Though adding new features and a new module, mod_gforge, may introduce new security risks, we believe this approach introduces much less risk compared to using multiple servers. Using WebDAV/Subversion integration improves GForge in several ways. First it reduces the number of different protocols supported; Davenport only needs to open ports for HTTP(S), which reduces administrative work and security risks. Second, Davenport reuses HTTP and its related technology such as basic authentication and SSL transport encryption. Third, the unified and aggregated HTTP log enables Davenport to provide detailed metrics. Finally, WebDAV and Subversion provide a superior authoring and version control environment.

3.2. Additional Features

In addition to the WebDAV/Subversion integration, we also added several new features, including SCM metrics, Subversion import/export, and Wiki discussions.

Gathering metrics about the project status and process is one of the major functionalities of SCM. For every project in Davenport, Subversion serves as a repository that records the status of all the components and SCM activities of the developers. Davenport collects project metrics, including commit numbers by each developer, the sum of lines changed by each developer, the commit number of each file, and the lines changed for each file, for a given period. With the help of these SCM metrics, displayed as graphs, the project members can supervise, analyze, and manage the development process, view the project metrics over time, and gather insight into each developer's work and status. The developers can also be aware of his/her peer developers' activities, as well as the whole status of the project.

One of the most frequent requests on SourceForge [6] is to import an existing CVS repository into SourceForge and to delete a specific module in CVS. Subversion handles this latter problem, and Davenport provides a web-based interface for Subversion's repository import function. A project administrator can upload a Subversion dump file and Davenport will automatically load the file into the Davenport Subversion repository.

Wiki is useful for discussion and scratch areas. Unlike mailing lists or discussion forums, Wiki provides simple and easy collaborative editing functionality through a web interface without using a special client. A good use of Wiki is sharing ideas or authoring frequently changing documents, such as a FAQ or TODO list. Davenport provides a separate Wiki space for each project using the virtual Wiki functionality in MoniWiki [19].

3.3. Administrative Improvement

Davenport only uses the HTTP/WebDAV protocol, and the administrator needs to monitor an httpd server. There is no need to configure other servers such as LDAP or CVS. If there is need to restart servers, the administrator just needs to stop and start the httpd server. There is no need to synchronize DBMS with any other data repositories.

4. Current Experience

We are hosting 15 class and lab projects with 40 registered users using Davenport at University of California, Santa Cruz [20]. For class projects, students are required to write design and requirement documents, as well as UI prototypes, test cases, and their project source code. Usually a project group consists of three or four students. They use the Wiki to write project documentation, author their project homepage using WebDAV, and use document management to upload and store their documents. They use Subversion daily to store source code and keep all changes. An instructor also creates a group for the whole class, and uses it to share information about lectures and projects using Wiki pages, discussion forums and news. There is no need or requirement to support SSH/SFTP, FTP, or CVS access, and without those protocol supports, Davenport satisfies all needs for developers as a CDE.

5. Future Directions for Davenport

IDE integration is a high-priority planned activity of Davenport. So far, Davenport only provides Web interfaces for users to access most of its functionalities. This is not very convenient for developers who spend their time working in client-side IDEs like Eclipse. In our future work, we wish to build a seamless integration between Eclipse and Davenport. Here are several steps we will take.

First, a WebDAV client will be integrated into Eclipse to access the resources in the WebDAV repository in Davenport. Second, a Subversion client will be integrated into Eclipse to access the resources in the Subversion repository in Davenport. Third, we will build SOAP services into Davenport for most of its functionalities, such as mailing lists, news, task managers, bug tracking, Subversion metrics graphs, diaries and notes. Accordingly, SOAP client libraries will be developed for these Davenport SOAP services and these libraries will be integrated into Eclipse.

If the users write requirements documents or design documents with Wiki in Davenport, some mechanism should be built to allow the user to browse both the structure and contents of these documents in Eclipse. In this way, the users can read the requirements documents or design documents conveniently while coding, and document-to-code relationships can be established and saved.

To evaluate software tools and development process, empirical measurement and feedback are very useful. In that sense, CDEs can serve as a software engineering research test bed. If a research group develops a new tool, to improve the software development process, how can we validate this tool? Also if a research group develops a new software development process, how do we know the new process is better than others? The new tool or new process can in some cases be integrated into a CDE, and the CDE can collect metrics on the process or tool. To support this, the CDE has to have a fully pluggable architecture so that authorized groups or developers can add their own tools or process control on top of the CDE. Also, the CDE must provide detailed awareness functionality.

6. Related Work

Tools and environments are basic aids for software development, and have existed since the early days of computer programming [21]. From UNIX shells to IDEs, development environments keep evolving. We discuss web-based CDEs in this section.

SourceForge Enterprise Edition [8] is a commercial web-based CDE package which is a derived from SourceForge.net [6]. Based on the SourceForge package, it adds several features, such as calculating metrics and enhancing interoperability. Still, it uses several dedicated network protocols, such as CVS and SSH/SFTP, which can be replaced by WebDAV.

Source Cast from CollabNet [7] is another commercial web-based CDE. Instead of supporting SSH/SFTP, it uses CVS areas for homepage storage. Each project has a special CVS module called www, and developers can check web resources into the CVS module to author the project homepage. Since CollabNet supports several Subversion developers, it seems likely they will integrate Subversion into Source Cast in the near future.

7. Conclusions

We have described the architecture of GForge and Davenport and how WebDAV can replace other protocols without sacrificing existing functionalities. Such WebDAV-based CDEs are more secure, reduce administrative overload, and can get benefits from many other WebDAV-aware tools.

Since WebDAV supports a superior environment for webpage authoring and version control environment, and reduces security risks and administrative work without sacrificing existing CDE functionality, we believe web-based CDEs should use WebDAV instead of several other protocols. Furthermore, the WebDAV protocol suite including DASL (searching), ACL (access control), is able to provide searching and detailed access control functionality for WebDAV-based CDEs, thus providing a more functional, standard-based platform for future extensibility.

8. Acknowledgement

We would like to thank the GForge developers. Especially we'd like to thank Jennifer Bevan and Mark Slater for their invaluable discussion and review of this paper. This project is supported by the National Science Foundation under Contract Number NSF CAREER CCR-0133991.

9. References

[1] C. Gacek and B. Arief, "The Many Meanings of Open Source," *IEEE Software*, vol. 21, 2004, pp. 34-40.

[2] J. D. Herbsleb and A. Mockus, "An Empirical Study of Speed and Communication in Globally-Distributed Software Development," *IEEE Transactions on Software Engineering*, vol. 29, no. 6, pp. 481-494, 2003.

[3] B. Berliner, "CVS II: Parallelizing Software Development," Proceedings of Winter 1990 USENIX Conference, pp. 341-351, Washington, DC, 1990.

[4] Apache Foundation, "The Apache HTTPD Server Project," http://httpd.apache.org/, 2003.

[5] The PHP Group, "PHP: Hypertext Preprocessor," http://www.php.net/, 2004.

[6] VA Software, "SourceForge.net Homepage," http://sourceforge.net, 2003.

[7] "CollabNet - SourceCast," http://www.collab.net/products/, 2004.

[8] VA Software, "SourceForge Enterprise Edition," http://vasoftware.com/products/, 2004.

[9] L. Augustin, D. Bressler, and G. Smith, "Accelerating Software Development through Collaboration," Proceedings of 24th International Conference on Software Engineering, pp. 559 - 563, Orlando, Florida, 2002.

[10] GForge, "GForge CDE: Collaborative Development Environment," http://gforge.org, 2003.

[11] B. Behlendorf, "Apache Software Foundation Server compromised, rescued," http://archives.neohapsis.com/archives/apache/2001/0012.html, 2001.

[12] Y. Goland, E. J. Whitehead, Jr., A. Faizi, S. Carter, and D. Jensen, "HTTP Extensions for Distributed Authoring -- WEBDAV," *Internet Proposed Standard Request for Comments (RFC) 2518*, 1999.

[13] B. Behlendorf, C. M. Pilato, G. Stein, K. Fogel, K. Hancock, and B. Collins-Sussman, "Subversion Project Homepage," http://subversion.tigris.org/, 2003.

[14] R. Fielding, J. Gettys, J. Mogul, H. F. Nielsen, L. Masinter, P. Leach, and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1," *Internet Draft Standard Request for Comments (RFC) 2616*, 1999.

[15] G. Clemm, J. Amsden, T. Ellison, C. Kaler, and J. Whitehead, "Versioning Extensions to WebDAV," *Internet Proposed Standard Request for Comments (RFC) 3253*, 2002.

[16] A. Babich, J. Davis, R. Henderson, D. Lowry, S. Reddy, and S. Reddy, "DAV Searching and Locating," *Unpublished manuscript, draft-davis-dasl-protocol-00*, 2000.

[17] G. Clemm, A. Hopkins, E. Sedlar, and J. Whitehead, "WebDAV Access Control Protocol," *Internet-Draft, work-in-progress, draft-ietf-webdav-acl-09*, 2002.

[18] G. Stein, "mod_dav: a DAV module for Apache," http://www.webdav.org/mod_dav/, 2003.

[19] "MoniWiki Project Homepage," http://moniwiki.sourceforge.net, 2004.

[20] "DavForge UCSC Homepage," http://dforge.cse.ucsc.edu, 2004.

[21] H. Ossher, W. Harrison, and P. Tarr, "Software Engineering Tools and Environments: A Roadmap," Proceedings of the Conference on The Future of Software Engineering, pp. 263 - 277, Limerick, Ireland, 2000.

Organizational Structure of "User Collaboration Community": Insights from the Case of an Open Source Software Project

Jen-Fang Lee, Tzu-Ying Chan
National Cheng Chi University of Technology & Innovation Management, Taiwan
E-mail: jflee@nccu.edu.tw; g8359503@nccu.edu.tw

Abstract

Current literatures on organization innovation do not investigate the innovation model which is composed of "network between individuals". Therefore, this study proposes the concept of the "User Collaboration Community", tries to understand this new phenomenon by conducting OSS project is the subject of this analysis, borrows the observation variables and propositions adopted by Mintzberg on structures of the innovative organization. This study further infers the conceptual framework and a series of hypothesis on the organizational structure of the user collaboration community. We expect that the proposition of this hypothesis will function as a concrete description and presentation of the innovation model of the "User Collaboration Community".

1. Introduction

Current literatures on organization innovation do not investigate the innovation model which is composed of "network between individuals". How is the organizational structure of the innovation model operated? Can the current organization models and theories explain or describe the organization platform of the "user collaboration community"? Therefore, this research will pinpoint the deficiencies of these literatures and try to answer these questions. This study first conducts a brief review of some important scholar's research frameworks and observation variables for innovative organizations. After a comparison, this study decides to adopt Mintzberg's observation dimensions and propositions on the innovative organization as a basis for studying the formats of the organization platform of the user collaboration community, and also adopt three open source software projects as subjects for analysis.

This structure of this article is as follows: Section 2 is a literature review. Section 3 proposes the conceptual framework of the organizational structure of the user collaboration community. Section 4 is data collection and analysis, and Section 5 is the conclusion.

2. Literature review

The review of related literatures on organization innovation indicates that most literatures focus on innovation behaviors and determination factors for innovation adopted "inside an organization", emphasizing long-term, cumulative, durable innovation within one organization. In other words, it is organization innovation at the firm level, such as self-organization form [20], heavyweight development team [2], spider web organization [21], hypertext organization [19], fractal organization, practice community [1,25], strategic community [22], and global research network [6,11], etc.; In addition, many literatures also discuss innovation models of network relationships formed by organizations and similar organizations in the external environment, focusing on flexibility and combination. Knowledge accumulates in each actor. The innovation is achieved through the combination of different actors. In other words, it is organization innovation at the industry level, such as virtual organization, network organization, boundless organization, etc.[4,7].

In regard to research fields, literatures on organization innovation did not investigate the innovation model composed of "network between individuals". The most famous innovation model of this type is the operation of Linux community. How is the organizational structure of the innovation model operated? Can the current organization models and theories explain or describe the organization platform of the "user collaboration community"?

[26] indicated that individual variables, organization variables, and environment consequence variables all contribute to organization innovation. Of these three variables, the organization variable carries the most weight in explaining the organization innovation. In the organization variable, scholars often adopt "organizational structure" to study organization innovation. [17] thought that the essence of organizational design is the manipulation of a series of parameters that determine the division of labor and the achievement of coordination. He proposed nine major parameters for structural design.

Many scholars have proposed capabilities of organization innovation such as the degree of organization formalization, the degree of organization specialization, the degree of organization centralization, the entrepreneurship, etc. All of these are important variables that may affect organization innovation. From the literature review, it is not easy to come up with a consensus on organization innovation. In addition, [15] indicated that most studies on exploring the predicting factors for organization innovation focus more on the individual predictability of different variables, but less on the joint predictability of various variables. As a result, these studies could not provide an overall picture of organization innovation and inhibit the accumulation of research results on organization innovation. Configuration approaches, on the other hand, provided a solution for resolving the problems mentioned above and offered an alternative method for organization analysis. For those scholars who adopted configuration approaches, an organization is a combination of various factors that occurred simultaneously. Thus, parameters for organization design could not be selected independently [16.17]. Therefore, this study adopts the theory of organization structure proposed by [17], based on configuration approaches, especially his integrated analysis on innovative organization structure, as the main analytical framework of this study.

3. Conceptual Framework

This study is trying to answer the question "Can current organization models or theories explain or describe the organizational structure of user collaboration community?" Therefore, this research tries to utilize Mintzberg's observation and inference on innovative organization to construct the conceptual structure of this research. Even though Mintzberg used business as the analysis unit, this study uses the individual network as the analysis unit, and this study believes that there will be no difference in the operation model for the innovative organization caused by the difference of participants. The induction of the conceptual framework is listed below:

4. Data Collection and Analysis

The operation platform of OSS project organization not only helps clarify the innovation model of "the organization platform of user collaboration communities", it further provides detailed process data regarding the internal operation of the project. For scholars who study organization innovation, this is a tremendously valuable database. It makes up the insufficiency of the research method of subjective recognition or pure description caused by the difficulties of obtaining data.

This research adopts three open source software projects as the subjects of case study. These projects are Linux kernel, Apache, and SFS (School Free Software). Because each of the three projects has a corresponding web site[26,27.28], this research uses documents on the web site, file records, and participation in observation as major manners of data collection. Linux[3,5,8,10,13,14,23,24]and Apache [9,12,18] focus mainly on the analysis of secondary data. However, SFS interviews related community members who participate in the operation of a project. It also uses VBA program language to automatically grab CVS and BUGDB[30] data for analysis.

SFS projects are briefly described as follows: A grammar school teacher in Taiwan, Ying Guan Chen, developed a curriculum management software and allowed everyone to use, revise, and pass around the software for free. So far there are teachers in more than sixty schools participating in the revision of the software, and there are more than two hundred schools, even schools in Hong Kong and mainland China, trying to install the software.

Parameters	Adhocracy structure characteristics
Behavior formalization	Little formalization
Training	Much training
Indoctrination	A few
Specialization of jobs	Much horizontal specialization
Unit grouping	Functional and market
Unit size	Narrow throughout
Liaison devices	Many liaison devices throughout
Planning and control system	Limited action plan
Decentralization	Selective decentralization

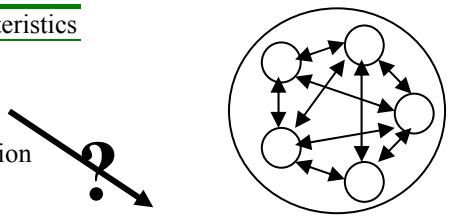


Figure 1. Framework of organizational structure analysis for "user collaboration community"

[17] felt that innovative organization is a highly organic structure with very few formalized behaviors. Members of this organization all receive formal and strict training to perform assigned work.

From table 2, we can tell that the Identifying of the development work of the OSS communities is proposed or reported by the communities. Nobody will assign anybody to perform what task. Whoever feels like performing the task may choose to be responsible for what they specialize or what they prefer. Therefore, there is no need to give orders, and there is no clear division of responsibilities. It is impossible to have a clear picture of division of works or to organize a group. As a result, it is impossible to have any formal or written documents to explain or to operate the activity procedures of members of the organization.

Hypothesis 1 The user collaboration community has the characteristics of a low degree of behavior formalization.

[17] felt that when an organization cannot standardize its behavior, it will assure people who perform the tasks possess the knowledge and skills required for the tasks through intense training beforehand. In the three projects, the phenomenon of "standardization of skill" is not discovered. However, in order to allow beginners to continuously and periodically participate in the operation of the project and also to assure participants' withdrawal from the project will not impact the operation of the project, the OSS project utilized informal training system to assure participants in the project possess a certain degree of knowledge and skills. For example, the design of property right system allows source to be shared, studies, verified, and passed on among the communities. In addition, the matched screening mechanism allows the project to verify the participants' knowledge and skills. It makes the participants' performance visible and transparent. The examination process of the program is similar to the peer evaluation in the scientific field.

Hypothesis 2 The user collaboration community has an informal t raining system.

[17] argued that "indoctrination" is considered to be a substitute of "behavior formalization" and "training". OSS community is composed of a group of members dispersed everywhere in the world. For most of them who are not compensated for their work, it is very important that whether they share a common belief. It relies on volunteers' self-encouragement. It utilizes a complicated but obscure monitoring and encouragement system to reinforce its norms and also allows everyone to contribute. It compensated the participants with peer recognition, but not money. Earning peer recognition is an important motivation of participating in open source. The desire of maintaining and strengthen their own reputation is the key mechanism to assure the progress.

Hypothesis 3 The informal social control system for the platform of the user collaboration community is very active.

[17] felt that there is a high degree of multiple horizontal specialization between members of an innovative organization. It is common to group experts in each field based on their specialized functions, and they then perform in the manner of a project team. Such organizations rely heavily on project teams and task teams and tend to result in numerous teams. However, the size of each team is small.

[17] thought that in order to allow the operation of an innovative organization, the organization has to maintain a variety of members, and each member has to specialize in his own specialty. Their responsibilities are few and the scope is narrow. The core developers in the Linux and the Apache projects devote only in several modules. It shows a high degree of specialization. However, a half of the core developers in the SFS project participate in the addition and the revision of 50% of the modules. The different results are probably caused by the small difference of the technology among each module and the development history of the project is short.

Hypothesis 4 The organization platform of the user collaboration community has the characteristics of a high degree of multiple horizontal specialization

From the three projects, we discover that the OSS project not only separate groups based on the functions, such as groups of core development and testing, when it constructs the project, it also tries to allow the product structure reflect on the project structure. [17]felt that the project team of the innovative organization should be kept at a very small scale due to the coordination. It indirectly leads to a large number of projects. From the three projects, we find that most projects are divided into numerous modules and the number of participants in each module is not large. [18]had the same research result. OSS is a big project. Therefore, it is divided into smaller sub-projects for easier management.

Hypothesis 5 The organization platform for the user collaboration community proceeds in the manner of a project team.

Hypothesis 6 There are many project teams in the organization platform for the user

collaboration community, but the scale of each team is very small.

[17] felt that there are various liaison mechanisms within teams or between teams in an innovative organization to promote mutual adjustments. Power is separated selectively in these teams, and the styles of the power division are selective vertical and horizontal decentralization. Planning and control systems are the grassroots model, and the information flow and the strategy process are very flexible and very informal.

Because there are numerous project teams in the OSS project, it is necessary to have a communication mechanism to coordinate the operation of each team. In the three projects, we discover that in addition to coordinate the operation of the project through abundant collaboration development tools, these projects also have coordination staffs such as project leaders, testing managers, and product managers.

Hypothesis 7 There are many liaison mechanisms in the organization platform of the user collaboration communities

Even though the mechanism of the release of decision-making power is different among each project, they all release the decision-making power to community developers who are highly familiar with each module. From table 3, we can tell that power is also divided among team members. The addition and revision of files and modules are generated jointly in a certain format through the coordination and cooperation of the communities. The principle of the division of decision-making power is totally based on the characteristic of the decision.

Hypothesis 8 The organization platform of the user collaboration communities have the characteristics of selective vertical and horizontal decentralization

Summarizing the development flow of the three projects (table 2), we discover that the plan and the control system of these projects is grassroots model. It is different from the strategy setup manner of the traditional greenhouse model. The web site of the project is an open testing ground. It provides a space that nobody can plan and control. In addition, from the summary of the comparison of the origin and the current condition of the development (table 1), we can tell that these projects were not developed based on the current objective. Follow the growth of user community the direction of the development software functions tends to be beyond the expectation of the original founder. It derives many applications that are not expected originally.

Hypothesis 9 The planning and control system of the organization platform of the user collaboration communities is the grassroots model.

5. Conclusion

This research tries to use Mintzberg's observation and inference on innovative organization to explain the organizational structure of OSS project. This article discovers that even though the analysis unit of OSS project is a community composed of individuals, but not manufacturers, the organization operation model still conforms to Mintzberg's inference on the organization structure of which the mission is to perform innovation.

Acknowledgements

The financial support from National Science Council of Taiwan is gratefully acknowledged (the project NO: NSC 92-2416-H-004-021)

References

[1] Brown, J. & Duguid, P. (1991),"Organizational Learning and Communities of Practice: Toward a Unified View of Working, Learning, and Innovation", *Organization Science*, 2(1): 40-57

[2] Clark, K. & Wheelwright, S. (1999), "Organizing and Leading 'Heavyweight' Development Teams", *California Management Review,* 34(3): 9-28.

[3] Dafermos, George (2001), "Management and Virtual Decentralized Networks: The Linux Project", Durham Business School M. A. Thesis

[4] Davidov, W.H. & Malone, M.S (1992), The Virtual Corporation, Harper Business, New York

[5] Dempsey, B. J., D. Weiss, P. Jones & J. Greenberg (1999). 'A Quantitative Profile of a Community of OpenSource Linux Developers', School of Information and Library Science, University of North Carolina at Chapel Hill, SILS Technical Report, TR-1999-05.

[6] Gassmann, O. & Zedtwitz, M. (1998), "Organization of Industrial R&D on a Global Scale", *R&D Management*, 28(3): 147-161

[7] Hedbeag, B., Dahlgren, G., Hansson. J. & Olive, N. (1997), Virtual Organizations and Beyond: Discover Imaginary Systems, Wiley, London.

[8] Jae Yun Moon, Lee Sproull,(2000), "*Essence of Distributed Work: The Case of the Linux Kernel*",5, No. 11, First Monday, November 2000. www.firstmonday.dk/issues/issue5_11/moon/

[9] Jagielski, Jim(1999),"The Apache Success Story -Exploring an Open Source Development Process", http://www.newarchitectmag.com/archives/1999/10/jagielski/

[10] Koght, Bruce.& Anca Metiu (2001),"*Open Source Software Development and Distributed Innovation*", Oxford Review of Economic Policy, 17(2): 248-264.

[11] Kuemmerle, W. (1997), "Building Effective R&D Capabilities Abroad", *Harvard Business Review*, 75(2): 61-70

[12] Lakhani, Karim & von Hippel (2003), "How open source software works: "free" user-to-user assistance", *Research Policy* ,32:6,923-943

[13] Lee,Gwendolyn K.&Robert E. Cole(2000), "The

Linux Kernel Development as a model of knowledge creation", working paper
[14] MacCormack (2001), "*Product –Development Practices That Work: How Internet Companies Build Software*", MIT Sloan Management Review, Winter
[15] Meyer, A. & Goes, J.(1988), "Organizational Assimilation of Innovations: A Multilevel Contextual Analysis", *Academy of Management Journal,* 31(4): 897-923
[16] Meyer, A., Tsui, A. & Hinings, C. (1993),"Configurational Approaches to Organizational Analysis," *Academy of Management Journal,* 36(6): 1175-1195
[17] Mintzberg, H. (1979), The Structuring of Organizations, Englewood Cliffs, NJ., Prentice Hall
[18] Mockus, A., Fielding, R.T. & Herbsleb, J. (2000), "A case study of open source software development: the Apache server", Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000), Limerick, Ireland. June 4-11, ACM Press. p.263-272
[19] Nonaka & Konno, N. (1993), "Knowledge-Based Organization", *Business Review*, 41(1): 59-73
[20] Quinn, J. B.; Baruch, J. J.; & Zien, K. A. (1997), Innovation Explosion, The Free Press, N.Y.
[21] Quinn, J.B., Philip. A. & Syndey. F. (1996), "Management Professional Intellect: Making the Most of the Best.", *Harvard Business Review,* 74(2): 71-80
[22] Storck, J. & Hill, P. (2000), "Knowledge Diffusion Through Strategic Communities", *Sloan Management Review,*41(2): 63-74
[23] Torvalds, L. & Diamond, D. (2001). Just for Fun
[24] Tuomi, Ilkka (2002), Networks of Innovation - Change and Meaning in the Age of the Internet, OXford University Press
[25] Wenger, E. & Snyder, W. (2000), "Communities of Practice: The Organizational Frontier", *Harvard Business Review*, 78(1): 139–145
[26] Wolf, R.A. (1994), "Organizational Innovation: Review, Critique and Suggested Research Directions", *Journal of Management Studies,* 31(3): 405-430

Web sites

[27] Linux-kernel project http://www.kernel.org
[28] Apache project http://www.apache.org
[29] SFS project http://sfs.wpes.tcc.edu.tw/index.php
[30] SFS bug track system http://cvs.tnc.edu.tw/bugtrack/login_page.php

Table 1 The Origin and Description of Open Source Software Communities

	Linux Kernel	Apache	SFS
Starting Time	1991-	1995-	2000-
Technology Characteristics	Operation system	Web server	Information system (Curriculum management)
Project Origin	Torvalds developed end virtual program to connect newsgroups of Usenet…..	Behlendorf gathered several people to develop revised program for Http web server…	Ying Guan Chen developed a system for books loan service ……
Licensing Agreement	GPL	BSD	GPL
Community Size (The result of innovation?!)	The operation system that has the highest growth rate. Support more than ten different types of CUP. It is good for multiple applications. Better than commercial software in safety, efficiency, and reliability. Evolved more than a hundred versions. There are more than four thousand web sites in the world.	Killer Application(67%) may cross platforms to be operated on about thirty operation systems. (IIS supports only one platform. The market share is 21%)	More than 200 schools installed the system. It even prevails in Hong Kong and mainland China. It also generates community communication stations for other five counties and cities and SFS version for vocational schools. More than 60 schools participate in the revision of the software.

Table 2 The Development Process of Open Source Software

	Linux kernel	Apache	SFS
The recommendation of core members and their responsibilities	Developed from only Torvalds to several hundred developers, assistants. Once a person's reputation is established, the revised program will be passed to this person and this person will gradually become a famous person for studying a specific problem and gradually picks up the tasks of core development teams.	Twenty-five members of the board (AG) are all volunteers without receiving any compensation. They possess the authority of CVS. The core development team includes AG and other active developers.	Started with only Ying Guan Chen to 15 people. They possess CVS authority. Most of them participate in the project through the recommendation of core members.

Identifying of development work	Induce communities' opinions on tasks to be done through discussion groups.	Although AG puts authority limitation in the program to control all revisions on the program, the whole development process is open. Other people may provide feedback on errors or propose directions for improvement.	Communities report errors or propose directions for improvement through news group, mailing list, and Bugzilla.
The assignment and operation of development work	Nobody assigns whom to do what. Anybody who feels like performing the task may handle a certain module.	Anybody may provide patches, develop their favored or specialized modules, or respond other people's questions through mailing list, news group or BUGDB	List problems to be solved through Bugzilla, news group and mailing list Anyone may propose program modules or patches; however, only core members may put source directly in CVS.
Writing programs and testing the program voluntarily	Writing programs and testing the programs voluntarily.	Volunteer developers write programs and test the programs voluntarily on the server to make sure the operation is error free.	Developers voluntarily test the program they write and eliminate the bugs they discover.
Examination	Each of the assistants is responsible for a certain specific dimension of the core program. They screen and examine the patch programs transmitted from other developers, then forward the result to Torvalds	More core developers will examine all changes. Other community members who are not in the core development teams may also participate in debugging. The changes made on the stable releases have to be examined by other developers before the commitment. However, changes made on the development releases will be examined after the commitment.	Nobody will examine the programs in the "program code communication center" before uploading, but core members will examine and test these programs afterwards. If the function of these programs are good, the developers will include these programs as part of the main version. Currently utilize Bugzillla to feedback, control, and manage the progress of debugging.
Releases management	Core programs have parallel release structure that can be classified into hacker version and common user version. These programs are released in an accelerated revision cycle model. Several new versions are released in a month. Revisions are uploaded and released once it is completed.	There is no fixed schedule for the releasing of new versions. Core development team takes turns to be the manager of a certain version. After all approved revised programs are adopted in a new version, and before the version is released, developers are not allowed to make any changes to the depository of the source.	There are stabilized version, development version, and testing version. Once the development version is assured to have no error, it will be released to the stabilized version. The developer personally determines the development condition of a module. In its early days, patches are published periodically for users to download. Currently, the upsfs is adopted and is set to revise every three hours by all schools simultaneously.

Table 3 The Task Division Condition of Open Source Communities

	Linux	Apache [18]	SFS
Revision of programs	_ Even though there are many people, their contribution to the innovation of products is extremely different. *(Several thousand teams develop the program, several hundred thousand people work on debug report. However, not many people really contribute tremendously to the core program. There are about ten core developers in the top level.)* *(2% of the contributors provide 50% of related information.)*	_ Even though there is a large number of participants, the change or the addition of program codes is seldom contributed by non-core developers. The addition and maintenance of new functions are almost operated by the core development team. *(The top 15 program developers contribute 83% of the patch program, 88% of new added program codes, and 91% of deleted program codes.)* _ In addition to core development team, few developers may propose changes regularly. _ Developers strongly need a certain format to coordinate generated products. In addition, nobody can individually write the complete program codes for a certain module. *(42 files have more than 30 changes. More than 10% changes are made by developers for at least 2 of 40 files (at least f of 20 files).* .	_ Core members are in charge of the revision of most programs. *(14% of the developers provide 74% of all modules. The founder, Mr. Chen, contributed 23 modules, only 2% of all modules*)-Program communication center (*Of the 11 developers,one developer participated in the revision of about 71% of the modules. Next is 63% by Ying Guan Chen. Overall speaking, 5 developers participated in the revision of 50% of the modules*) -CVS _ There are frequent revising activities for modules and files. *(More than 14% of the files are revised over 5 times and 56% of the files are changed over 2 times. Analyzing based on number of changes of the modules, about 43% of the modules are changed over 50 times, 98% of the modules are changed more than 10 times, 2% of the modules are changed under 10 times.)* _ Collaboration process *(84% of the modules are developed through the collaboration of more than two people.)*
Report on errors	N/A	_ Other non-core teams, the large number of user communities, are responsible for the important role of system testing. *(Of the top 15 PRs, only 3 are core development teams. There are 213 articles from the top 15 PRs, which is only 5% of all PRs.)* _ Debugging through frequent or more efficient methods. It attains lower poor quality ratio. *(50% of the PRs solve the problem on the same day,75% of the PRs solved the problem in 42 days,90% of the PRs solved the problem in 140 days.)*	_ Error reports are mostly operated by other non-core members. *(The PR contributed by the core members (only 5 people) is only 23% of all the PR.)* _ Communities assume a very important role in finding system errors. *(The types of report bugs are mostly error messages. Problems on structure, database, and program code are the second most frequent problem. Problems on "operation" and "installation" are only 12% of all messages.)* _ Communities assume an important role in suggesting system functions.*(The most common errors reported by communities are "insufficient function". It takes up 34% of all messages.)* _ Bugs may be discovered timely and the problems may be solved through the platform developed by the collaboration and cooperation of communities. *(The longest time to open a bug is 8.03 days, and the average time to solve a problem is 4.28 days.)*

Inside an Open Source Software Community: Empirical Analysis on Individual and Group Level

Wolfgang Maass
=mcminstitute, University of St Gallen, CH-9000 St Gallen
wolfgang.maass@unisg.ch

Abstract

An established Open Source Software community (Apache Cocoon) was explored using an online questionnaire about demographic data and individual and group-related factors. Individual factors encompassed forms of contributions, motivation, expertise and knowledge. Role structures, expectations towards other members, trust and collaboration issues were analysed at group level. More than 60% of the developer community completed this questionnaire. Results provide a valuable basis for deeper understanding of knowledge sharing, collaboration and innovation processes in distributed work groups.

1. Introduction

Open Source software (OSS) communities have recently attracted attention for the impact they have on the global software market. This development especially enlivened research on the organisation and motivational structure of individuals (e.g. [1], [2], [3], [4], [5]).

OSS communities are virtual work groups consisting of members with skills in software development. They work in temporary, culturally diverse, geographically dispersed, electronically communicating work groups ([6]).

In contrast to commercial software vendors, members can freely join OSS communities and are unrestricted in their contributions. Virtual work groups have been investigated within given contexts ([7], [8]) but, compared to Linux or Apache communities, these groups have been rather small. Software development by virtual work groups is a long-standing topic, however if compared to larger OSS communities these studies also mainly focused on smaller projects [9]. In this sense, OSS communities provide unique opportunities for studying virtual work groups and distributed software development.

On one hand, the communication and collaboration of OSS communities are highly transparent. On the other hand, global distribution of community members and lack of job contracts with firms make it difficult to investigate communities on an individual level. Therefore most empirical studies on OSS communities concentrate on secondary logging information such as that provided by mailing lists, IRC chat logs and code repositories.

As a prerequisite for understanding how OSS communities communicate and collaborate, we need better knowledge of their beliefs, goals, attitudes, skill sets, communication and collaborative behaviour. Based on a detailed online questionnaire, the Apache Cocoon community was analysed at individual and group level. More than 60% of the developer community (34 valid responses) responded to this questionnaire.

2. Conceptual Foundations

Commercial software development rests on the ability to allocate technical and human resources in relatively complex organisational settings [10]. Labour is divided into specialist groups, such as technical programming teams, system engineering, integration/testing and project management [11].

Globally distributed software development by virtual teams promises the flexibility, responsiveness, lower costs, and improved resource utilisation necessary to meet ever-changing task requirements in highly turbulent and dynamic global business environments [12].

Open Source Software communities provide a role model for virtual teams that have inherited cultural, organisational and communicational schemes from the scientific community but then enriched and expanded the communication and collaboration processes step-by-step. From a psychological viewpoint, the perception of membership in some common social identity is constitutive for a group to exist [13]. Work groups are groups that are focused on the generation of problem solutions.

Any developer of OSS communities maintains a set of collective, social, norm-oriented and reward motives [2]. However, they also have the liability to deliver

contributions of a given quality for which rewards are granted in terms of gaining reputation within the community. Contributions depend on the expertise and skills that a developer obtains. Physical disconnection is regarded as being a negative factor for team performance [14]. OSS communities organically design organisational structures that intercept this threat and the known limitations of communication infrastructures.

Initial studies explored work group structures of OSS communities ([2], [3], [4], [1]). In a detailed study, Hertel et al. analysed motivational processes of members of the Linux kernel community. This study gave first insights into the demographic structure of parts of a community. Members were differentiated into two groups: developer group and interested reader group. On average, participants worked 18.4 hours per week on Linux and 20% of the developers received a salary for their Linux programming work. 38% were able to carry out Linux-related programming during their regular working hours, whereas the remaining 62% worked on Linux outside their regular work. The Linux kernel community is subdivided into teams ranging from two to 50 developers, with an average of 12. Studies found that the main motivational factors were identification factors, pragmatic motives, norm-oriented motives, social and political motives, hedonistic motives and motivational obstacles related to time losses. All factors were found to correlate positively with willingness to engage in the community, while lack of time is the biggest obstacle. Trust, in a limited meaning, only plays a minor role for motivation.

Established OSS communities such as Linux, Apache or mySQL emerged from small toy projects to fully-fledged software solutions for complex tasks. Accordingly, developers are required to provide mature expertise and development skills. Knowledge of members is codified [15] by information communicated to other members and most of all into documentation and source code. It depends on experience gathered over time and can be transformed into a shared experience that leads groups of people to encode, store, and retrieve relevant information together [16].

At group level, the organisation gives a formal structure to a community. OSS communities typically provide simple and clear-cut organisational structures.

Structural conditions for OSS organisations include a general culture, delegative and participative leadership principles, modular project structure, parallel release policy, motivating credit policy, and clear rules and norms [2]. OSS communities typically grow around an individual or a small team ([17], [18]).

The analysis of OSS communities in respect of their work group organisation is still an open field. Recent studies on open source software communities focus on the evaluation of indirect historical data, as provided by mailing list repositories and Concurrent Versions System (CVS) entries ([1], [19]). The question remains as to whether organisations of OSS communities provide a new model of business organisation [20].

Members in communities communicate and collaborate on the basis of mutual expectations, which in turn generate mutual trust or suspicion [21]. Several factors have been suggested to facilitate the development of trust: (1) shared social norms, (2) repeated interactions, (e.g., [22]), (3) anticipation of future association [23], and (4) physical proximity [24].

In respect of their communication, virtual work groups, and OSS communities in particular, are restricted in the exchanges considered to be relevant for building trust, warmth, attentiveness, and other interpersonal bonds within teams [25].

Software development can be described as a collaborative problem-solving activity where success is dependent upon knowledge acquisition, information sharing and integration, as well as the minimisation of communication breakdowns [3].

3. The Present Study

In order to get a broader empirical basis for research on virtual teams, we searched for an OSS community that possesses important attributes; such as whether it is well-established (at least 3 years), sufficiently large but not to large (50-100 developers), still in its growing phase, and with a vision on its software system that is relevant for business usage. Therefore we selected the Apache Cocoon community, which was established in 1998, includes some 40-60 active developers and has gained growing attention from firms for its architecture on XML publishing (cocoon.apache.org).

The questionnaire targeted (1) demographic data, (2) individual factors (contributions, motivation, expertise), and (3) group factors (organisation, mutual expectations, trust, collaboration, communication channels) as described in the previous paragraph.

4. Method

Data was gathered by an online questionnaire that was developed in cooperation with leading members of the Cocoon community in September 2003. The questionnaire was actively communicated by leaders at Cocoon's first get-together in October 2003, which was the reason why 42 responses (34 valid) were given, representing more than 60% of the developer community.

This questionnaire was designed to receive a broad understanding of the community on which further empirical studies can be designed. The response rate was surprisingly high even though the questionnaire contained 42 questions.

5. Results

5.1. Demographic information

The average age of Cocoon developers is 31.2 years (SD 6.17 years). 73.5% are from Europe, 20.6% from the USA and about 3% from Asia and Australia. Developers have become aware of Cocoon in three phases, showing characteristics of the diffusion function: 26.5% before 1999 (initialisation), 35.3% in 2000 (growth), 38.2% from 2001 to present date (stabilisation). On average, they started to contribute 1.6 years later (0.756, $p < 0.01$)[1].

On average, developers invest 3.1 hours per week in Cocoon (only 16% of the investment of Linux kernel developers [9]). Assuming a total number of developers of 50, this means a total investment of 150 hours per week, about 600 hours per month and 6,600 hours per year, which equals about 3.75 person years. According to the responses, 74% are paid by official jobs (Linux kernel only 38% [9]). 23.5% of Cocoon developers are self-employed, with a significant correlation to conducting business with Cocoon (.502, < 0.01). More than 60% of these people receive between 80-100% of their revenues from business with Cocoon.

5.2. Individual factors

The production fingerprint of an OSS community is given by its distribution of delivery types, such as patches, modules, problem reports, documentation, and ideas.

Members were asked which type of contribution they deliver frequently, periodically, rarely, or never. Only 6% report frequently and 20.6% offer patch contributions periodically, which is consistent with our analysis of CVS activities in Cocoon. The majority indicate that they rarely or never contribute patches. Modules are larger-scale conceptual and functional units consisting of a set of files. Almost 12% indicate that they contribute complete modules on a periodic basis. The correlation analysis shows an expectedly strong, positive relationship between patch and module contribution (0.501, < 0.01). As a rule, modules are provided by members who joined the community early.

The results show that 25% of all developers are active contributors of problem reports. 26.5% of all members write documentation frequently or periodically, which is an astonishingly high percentage. A significant relationship exists between people who write patches and documentation (0.39, < 0.05).

[1] based on Pearson correlations with a two-tailed significance test.

29.4% of members report frequent or periodic contribution of ideas, which significantly correlates with the activity of contribution of modules, documentation, and problem reports. Members who contribute ideas are those who are responsible for modules (0.709, < 0.01) than mere patches (0.364, < 0.01), which supports the view that ideas influence the overall architecture built on the basis of modules controlled and delivered by the same people.

Several significant correlations exist between contribution types, which indicates that active developers contribute anything from patches, modules, problem reports, ideas and even documents. People providing patches, modules and ideas tend to provide documentation to a lesser extent. Documentation is provided more by people who generate problem reports.

The motivation for participation in OSS communities is one of the key issues investigated by recent research [2]. Results from the Cocoon community support these findings showing that the main motivation is to learn new technologies (79.4%). 52.9% report that public utilisation is a main driver for their participation. Data shows a slightly negative correlation between personal branding and public utilisation, indicating a distinction between these two groups.

Teams of knowledge workers are inherently resistant to bureaucratic rationalisation and control [26]. Routine work and organisational issues dominate and are perceived to be a distraction from software engineering work. A hypothesis is that OSS communities are attractive to developers because routine work is omitted and the organisation is reduced to a minimum. This hypothesis can be supported by the result that 44.1% see self-realisation as a main driver for their participation. Further research is necessary for analysis of this hypothesis.

In summary, the analysis of motivational factors supports the view that people participate in the Apache Cocoon community because they want to learn new technologies from and with people with similar interests. Cocoon can be seen as a global family of skilled software developers. Results support the initial hypothesis that Cocoon is a global knowledge-sharing network.

Results show 65.7% of all members have a background in computer science and 24% in electro-engineering or general engineering. The questions regarding educational background show that 70.6% possess or are working on a university degree, while 14.7% possess a degree from an advanced technical college. 14.7% see themselves as self-educated, which represents those with a non-technical or at least a non-IT background.

Years of experience represent an important factor for professional programming. On an individual level it could be shown across various domains that it takes about 10

years to become a domain expert [27]. On average, Cocoon developers have 12.2 years of experience. Given the average age of 31.2, this means that Cocoon developers started programming at the age of 19. This in turn correlates with the beginning of university study.

Age and years of implementation correlate (0.787, < 0.01). Therefore, Cocoon can be separated into a cluster of programmers above 36 years of age and with more than 17 years of experience and a cluster ranging from 19 to about 31 years with 2 to 13 years of programming experience. It can be assumed that these two groups differ in their preferences regarding Cocoon. The correlation between age and the motivation to learn new skills (-0.339, < 0.05) and self-realisation (-0.414, < 0.005) is negative, which indicates that older Cocoon member are less interested in learning new skills or self-realisation than younger members.

5.3. Group factors

The study shows that 37% see themselves as "lurkers" who currently watch activities in the developer community and 71.4% see themselves as developers. The overlapping 8.4% of people watching and contributing can be interpreted as people who are at the edge of actual contribution. 34.3% hold the status of a committer, who dispose of the right to check in changes into CVS and 31.4% the status of a member of the program management committee (PMC). 93.6% of all committers are also members of the PMC.

On average, Cocoon members expect advanced expertise from other members. Members who report that they participate in order to learn new technologies (-0.394, < 0.05) and for self-reputation (-0.394, < 0.05) tend expect lower levels of expertise. Other correlations are rarely visible, so that as a new assumption we can assume that the expected level of expertise is an independent and basic variable.

As already indicated, not all members implement patches or modules but instead provide documentation, ideas, problem reports and others. Therefore we specifically asked for expectations of programming skills. Programming is considered the primary skill enabling an OSS community to deliver its primary product. This hypothesis is supported by the significant correlation between the expected level of general expertise and programming skills (0.874, < 0.01). In general, the expectation of programming skills is almost identical to that of general expertise. Hence, we can conclude that Cocoon members either seem to perceive their ability to program as their most important and dominant skill or any kind of contribution on an advanced level.

In general, 48.6% of all members perceive technical architecture as the most difficult lesson to learn. 20% find it hard to learn the rules of communication and 20% the rules of interaction. This supports the view that the biggest obstacle is also the biggest challenge because on one hand members have the main motivation to improve their technical skills and on the other they have a technical construct that challenges their skills.

28% of all members respond that they find it easiest to learn rules of communication or the "who's who". Together with the indication that people and their roles are also learned quite easily, it can be concluded that members learn the organisation and communication of the Cocoon community fast.

In respect of trust, on average members highly trust one another (average 2.6 on a scale from 1 [blind trust], 2 [very high], 3 [high], 4 [medium], 5 [low] and 6 [no trust]). Trust correlates with delivery of patches (0.415, < 0.05), delivery of modules (0.401, < 0.05), delivery of problem reports (0.380, < 0.05), delivery of documentation (0.411, < 0.05), and delivery of ideas (0.491, < 0.01). In contrast to users, Cocoon committers (0.514, < 0.01), PMC members (0.475, < 0.01), Apache Foundation members (0.364, < 0.05) are positively adjusted regarding trust towards other members. In summary, this presents a positive picture of trust relations between members. Users and lurkers show negative trust relations towards community members, developers tend to be positively biased, and from the status of a committer upward, members demonstrate very high trust in other members. Hence, in the Cocoon community trust seems to be derived from general work relationships rather than from physical interaction, which contrasts Handy's view that trust requires physical interaction (Handy 1995).

On average, Cocoon members claim to collaborate with 2.1 other Cocoon members, with a clear tendency to work with fewer people rather than with more. Committers (0.524, < 0.01), PMC members (0.566, < 0.01) and Apache Foundation members (0.560, < 0.01) significantly work with more members than regular developers or users (0.288). Significant correlations exist to developers who contribute modules (0.521, < 0.01), contribute documents (0.554, < 0.01), and contribute ideas (0.597, < 0.01). When correlated with the main contributions this pattern changes: significant correlations exist to discussions (0.467, < 0.01), votes (0.509, < 0.01), requirements (0.393, < 0.05). Also positive is the correlation with trust in other members (0.456, < 0.01) which indicates that the more people a member works with the higher is his or her trust into them. Furthermore, the more hours a member spends working for Cocoon the more people he interacts with (0.377, < 0.01). Finally it was found that the more revenue a member makes with Cocoon the more people he or she interacts with (0.512, < 0.01).

6. Implications

Work group analysis of OSS communities is an important but yet unexplored field. This study shows that mature OSS communities are able to attract high-profile experts. In contrast to functionally divided organisation of commercial software development projects, Cocoon developers account for almost any type of contribution, such as ideas, problem reports, patches, and modules. The motivation is largely determined by skill improvement and altruism (correlating with [2]). Developers expect other members to be highly skilled and experienced. They trust in other members to a large extent and perform their work in small teams or alone (correlates with "self-efficacy" [2]). This indicates that members perceive themselves as experts for a specific module, which will be valued by other members for its contribution to the overall system.

Various research questions can be derived from the findings of this empirical study. Further research is required on how individual knowledge is transmitted to other members based on limited communication channels and how it is transformed into transactive knowledge. These exchange processes depend on how members appraise each other and how this attitude evolves over time. Trust is a key concept for further analysis of related questions. Comparisons with centralised software development organisations will provide fruitful insights on performance differentials between centralised and virtual group organisations. From an economic point of view this will nurture research on globally distributed innovation and design processes.

7. References

[1] S. Koch and G. Schneider, "Effort, co-operation and co-ordination in an open source software project: GNOME", *Information Systems Journal*, vol. 12, pp. 27-42, 2002.

[2] G. Hertel, S. Niedner, and S. Herrmann, "Motivation of software developers in open source projects: An internet-based survey of contributors to the Linux kernel", *Research Policy*, vol. 32, pp. 1159-1177, 2003.

[3] G. von Krogh, S. Spaeth, and K. R. Lakhani, "Community, joining, and specialization in open source software innovation: a case study", *Research Policy*, vol. 32, pp. 1217-1241, 2003.

[4] L. Y. Moon and L. S. Sproull, "Essence of Distributed Work: The Case of the Linux Kernel", *First Monday*, vol. 5, 2000.

[5] E. von Hippel, "Innovation by user communities: learning from open-source software", *Sloan Management Review*, vol. 42, pp. 82-86, 2001.

[6] A. L. Kristof, H. P. Brown, and K. Sims Jr., "The virtual team: A case study and inductive model", in *Advances in Interdisciplinary Studies of Work Teams: Knowledge Work in Teams*, vol. 2, M. M. Beyerlein, D. A. Johnson, and S. T. Beyerlein, Eds. Greenwich, CT: JAI Press, 1995, pp. 229-253.

[7] M. K. Ahuja, D. F. Galleta, and K. M. Carley, "Individual Centrality and Performance in Virtual R&D Groups: An Empirical Study", *Management Science*, vol. 49, pp. 21-38, 2003.

[8] T. L. Griffith, J. E. Sawyer, and M. A. Neale, "Virtualness and Knowledge in Teams: Managing the Love Triangle of Organizations, Individuals, and Information Technology", *MIS Quarterly*, vol. 27, pp. 265-287, 2003.

[9] D. B. Walz, J. J. Elam, and B. Curtis, "Inside a software design team: knowledge acquisition, sharing, and integration", *Communications of the ACM*, vol. 36, pp. 62-77, 1993.

[10] R. W. Zmud, "Management of Large Software Development Efforts", *MIS Quarterly*, vol. 4, pp. 45-55, 1980.

[11] H. P. Andres, "A comparison of face-to-face and virtual software development teams", *Team Performance Management*, vol. 8, pp. 39-48, 2002.

[12] A. Mowshowitz, "Virtual organization", *Communications of the ACM*, vol. 40, pp. 30-37, 1997.

[13] J. C. Turner, "Towards a cognitive redefinition of the social group", in *Social identity and intergroup relations*, T. H., Ed. Cambridge, England: Cambridge University Press, 1982, pp. 138-161.

[14] C. Handy, "Trust and the virtual organization", *Harvard Business Review*, vol. 73, pp. 40-50, 1995.

[15] B. Kogut and U. Zander, "The Imitations and Transfer of New Technologies", *Organization Science*, vol. 3, pp. 383-397, 1990.

[16] D. Wegner, "Transactive memory: A contemporary analysis of the group mind", in *Theories of group behavior*, G. Mullen and G. Goethals, Eds. New York: Springer, 1986, pp. 185-208.

[17] A. Mockus, R. T. Fielding, and J. D. Herbsleb, "A Case Study of Open Source Software Development: The Apache Server", presented at Proc. of the 22nd International Conference on Software Engineering, Limerick Ireland, 2000.

[18] C. Robottom Reis, "An Overview of the Software Engineering Process and Tools in the Mozilla Project", presented at Workshop on Open Source Software Development, Newcastle upon Tyne, 2002.

[19] G. von Krogh and E. von Hippel, "Special issue on open source software development", *Research Policy*, vol. 32, pp. 1149-1157, 2003.

[20] J. Lerner and J. Tirole, "The Simple Economics of Open Source", National Bureau of Economic Research, Cambridge, Working Reports 7600, 2000.

[21] M. Deutsch, "Trust and Suspicion", *The Journal of Conflict Resolution*, vol. 2, pp. 265-279, 1958.

[22] J. L. Bradach and R. G. Eccles, "Markets versus hierarchies: From ideal types to plural forms", *Annual Review of Sociology*, vol. 15, pp. 97-118, 1989.

[23] W. W. Powell, "Neither market nor hierarchy: Network forms of organization", *Research in Organizational Behavior*, vol. 12, pp. 295-336, 1990.

[24] B. Latane, J. H. Liu, M. Nowak, M. Benovento, and L. Zheng, "Distance matters: Physical space and social impact", *Personality and Social Psychology Bulletin*, vol. 21, pp. 795-805, 1995.

[25] S. L. Jarvenpaa and D. E. Leidner, "Communication and Trust in Global Virtual Teams", *Organization Science*, vol. 10, pp. 791-815, 1999.

[26] M. Reed, "Organizational theorizing: a historically contested terrain", in *Handbook of Organization Studies*, S. R. Clegg, C. Hardy, and W. R. Nord, Eds. London: Sage, 1996.

[27] K. A. Ericsson, R. T. Krampe, and C. Tesch-Römer, "The role of deliberate practice in the acquisition of expert performance", *Psychological Review*, vol. 100, pp. 363-406, 1993.

From Bazaar to Kibbutz: How Freedom Deals with Coherence in the Debian Project

Mattia Monga
Università degli Studi di Milano
Dip. di Informatica e Comunicazione
Via Comelico 39 – 20135 Milan, Italy
mattia.monga@unimi.it

Abstract

The goal of obtaining a coherent distribution of software packages where all programs interact smoothly increases its complexity with the number of applications, the number of architectures involved, and the number of system configurations supported. The Debian project aims at producing a software system with thousands of components running on eleven different hardware architectures, with three different operating system kernels. This paper describes the project and how the work of hundreds of people that never meet one with another can be coordinated to produce reasonably robust and integrated systems.

1. Introduction

Applications do not exist in the desert. In fact, they run in very complex environments where an operating system kernel, some device drivers, system and graphic libraries, common services, etc. coexist in order to provide the software platform on which users can enjoy their applications.

For this reason, from the beginning (see for example the GNU Manifesto [10]) proponents of *free software* aimed at producing complete systems, because no real freedom is possible if developers have to rely on non free components. Historically, a major hurdle on this goal was the unavailability of an operating system kernel. However, in the last fifteen years several kernels (e.g., Linux, FreeBSD, GNU/Hurd, etc.) were made available to the *open source* community, and it was feasible to build entirely open source computing platforms which integrate basic utilities with sophisticated application software.

Discussion on open source software development often focuses on the techniques used to organize an open source project aimed at producing a well defined application. A famous paper by E. Raymond [9] describes a style of development metaphorically called *bazaar*. In a software bazaar anyone could contribute code to the original promoters of the project, who take care of integration in the mainstream code. This approach is contrasted by Raymond to the traditional software engineering process, that in another famous writing [2] F. Brook compared to the approach people used to build *cathedrals*, where an architect leads a small group of skilled and specialized workers, with precise schedules and responsibilities. In fact, as recent studies have shown [6], some of the most popular open source projects (e.g., the Apache web server, the Linux Kernel, the Mozilla browser) are in between the two extremes: the project is carried on and scheduled by a core group of developers, strongly committed to the product, and the openness of the source code enables contributions from individuals who correct some bugs or add some features. These contributions are often scarce and it can be safely assumed that in general open source projects have a core of developers, no larger than 10 to 15 people, who control the code base, and it is responsible for 80% of written code.

In this paper I want to discuss how an open source community can produce an integrated system, composed by aggregating several software packages that typically derive from independent sources. These systems are commonly called *distributions*. To date (February 2004), the Linux Weekly News list of Linux distributions contains 374 items [1]. Indeed, one of the business models proposed in order to make profit from open source software is selling the added value of an assembled distribution [5], and consequently some of the most successful distributors (Red Hat, SUSE, Mandrake, etc.) are commercial firms, driven by tight coupled groups of developers. Instead, in the following sections I will focus on the Debian project, aimed at producing a coherent distribution of free software leveraging only on the work of independent volunteers.

As previously said, the limiting case of open source software development process was metaphorically compared to bazaars, in which contributors put their work in the "magic cauldron" of the community. However, building a coherent distribution requires a great effort of coordination and cooperative work, thus the bazaar metaphor seems completely inappropriate. Yet, in the case of Debian, the cathedral metaphor is also inadequate, since no main architects are present and the work is carried entirely on a voluntary basis. Therefore, I suggest the new metaphor of the *kibbutz*[1], for a cooperative

1 Kibbutzim are Israeli communal form of agricultural settlement. Originally it was predominantly agricultural and practiced a very high level of sharing,

community of volunteers sharing a common goal. The properties that characterize such a community are:

- people join the community on a voluntary basis, and they do not expect to be paid for their work;
- members agree on an ambitious final goal ("making the desert where users live blooming of good and free software");
- members share a civil consciousness and they accept that their work is regulated by explicit rules established by direct democracy.

Though voluntary work is an essential part of Debian, given the great number (thousands) of people involved in the project, I believe that a study of their coordination effort can be valuable in a greater context, since most of the ideas have to do to management of complexity and heterogeneity and I suggest they could be applied also to commercial organizations.

The paper is organized as follows: Section 2 presents the Debian project and its organization, Section 3 describes the development process it adopts and how coordination is achieved, Section 4 explains how Debian systems can be customized and finally Section 5 draws some conclusions.

2. The Structure and Goals of Debian

2.1 The Debian Motivation

The Debian[2] project was started by Ian Murdock on August 16th, 1993 in order to "carefully and conscientiously put together and maintain and support with similar care" a distribution of Linux software by working "openly in the spirit of Linux and GNU" [7]. From the beginning all Debian members were volunteers and they are still not paid by Debian to do their work in the project. However, from November 1994 to November 1995 Debian was sponsored by the Free Software Foundation and Debian motivated the creation of "Software in the Public Interest", a non-profit organization that provides a mechanism by which The Debian Project may accept donations. The money collected pays hardware and actual duty expenses of Debian representatives.

A Linux distribution puts together pieces of software that are in general built by people unrelated with the distributors themselves. The Debian project requires that software included in a Debian system is compliant to the "Debian Free Software Guidelines": basically, software has to be licensed with an open source license [5] that allowed freedom of use, distribution and modification without discriminations and restrictions that can affect unrelated code.

The first release to a greater public of a Debian GNU/Linux system was issued on January 1994 (ver. 0.91). It contained a few hundreds of programs and was put together by a dozen of developers. Today (January 2004) the project count 1268 members distributed worldwide and it manages more than 13,000 binary packages (corresponding to more than 8,000 of source packages) ported to 11 different architectures (i.e., Alpha, arm, hppa, i386, ia64, m68k, mips, mipsel, powerpc, s390, sparc) [4]. At least three complete Debian systems exist: beside the main Linux based one, there is one based on the BSD kernel and another based on the GNU Hurd kernel. The original Debian founder, Ian Murdock, does not work actively in the project since 1996.

2.2. The Debian Structure

Everyone can apply to become a Debian member. In order to be accepted in the project one has to demonstrate the control of the basic skills needed to manage software packages and the understanding of the "Debian Free Software Guidelines" and the "Debian Social Contract"[3].

By joining the group one gives his or her consent to contribute to the project according to the Debian *Constitution [8]*. The Constitution defines a lean organization with a *Project Leader (DL)*, a *Project Secretary (DS),* a *Technical Committee (TC)*, and *Individual Developers*. The DL, DS, and the chairman of the TC has to be three different persons. The work is entirely voluntary: nobody is obliged to do anything and everyone chooses freely to be assigned to a task he or she does find useful or interesting. A new DL is appointed every year by a general election involving all the individual developers that vote with a Condorcet's mechanism. The DL can make urgent decisions and he or she appoints the DS and, together with the TC, renews the members of the TC itself. The TC is composed up to 8 members, with a minimum of 4 people, and it decides technical policies and it composes developers disagreements. Individual developers can override any DL and TC decision by issuing a general resolution with a qualified majority. The DS is appointed by DL and the previous DS every year. The DS is in charge of managing elections and other calls for vote and it adjudicates any disputes about interpretation of the constitution. The properties and financial activities are managed by "Software in the Public Interest, Inc" (SPI), in which every Debian member can be a voting member.

The consequence of this organizational structure is that no single individual can take personal control of the project. Even better, "Any individual Developer may make any technical or nontechnical decision with regard to their own work." [8] However, since coherence of the final product is one of the goals on which members agree, this absolute freedom has to be temperated by coordination achieved by a number of *policies*, that, after discussion on the mailing lists (most of them are public

including collective rearing of children. More recently (by 1998) industries have taken over a significant role in the Kibbutz economy.

2 The official pronounciation of Debian is 'deb'~ee~en'. The name comes from the names of the creator of Debian, Ian Murdock, and his wife, Debra.

3 The Debian Social Contract states that the Debian project will be always free software (according to the definition of the Debian Free Software Guidelines), it collaborates with the free software community and it follows procedures open to the public.

and also non developers can contribute to the discussion -- see http://lists.debian.org), are defined by the TC, but they should meet a high degree of general consensus to not be overridden by general resolutions. I will discuss policies further in Section 3.2.

In order to study the Debian organization it is important to consider a number of actors that interact with the Debian galaxy, without necessarily being members of the project. Yet, they may influence the Debian work.

First of all there are *Upstream Authors*. They contribute to Debian by writing open source software. In theory they could not even know about Debian. In practice they are often in direct communication with Debian developers, because inside Debian a lot of work is done to discover and correct bugs. Thus, it is common that Debian developers (from now on, DDs) forward to upstream authors bugs, patches, suggestions, new features requests, etc.

Secondly, there are *Users*. Satisfaction of users is of course an important force that indirectly drives the project. Moreover, Debian systems provide a sophisticated infrastructure for bug tracking (Debian Bug Tracking System, DBTS). It is the main avenue through which users can report problems and propose enhancements. It is important to understand that it plays a critical role in the pursuing of coherence, since it is used also to report bugs of the distribution itself. For distributions the same Linus' Law [9] of generic software applies: when they are exposed to a great number of observers, with different needs and slightly different operating environments, all bugs are shallow.

A third category that is worth mentioning for its increasing significance is the one composed by people that use Debian systems to build their own *specialized distributions*. The openness and intimate coherence of Debian systems make them ideal candidates to be customized for specific purposes. The most successful customization is probably the Knoppix distribution, aimed at producing a system running entirely from a CD and able to recognize automatically a huge set of different hardware on i386 machines. As I will discuss in Section 4, the open architecture of Debian system is particularly apt to customizations without necessarily going out of sync with the mainstream Debian distribution.

3. The Debian Development Process

3.1. Debian Distributions

A distribution of a Debian system is composed by an installation program and a set of software packages. The installation program is able to set up the system from scratch on a large number of different hardware configurations: this makes the installation a quite complex operation. Software packages can be retrieved from a set of CDs, a local hard disk or the network.

All the Debian development effort is focused on the production of packages. A *package* is the minimal unit that can be installed or removed from a system. Consequently, each DD is responsible for one or more packages, and he or she is said to be the *maintainer* of that package[4]. When a maintainer has put together his or her package, it is uploaded to a public repository from where Debian users worldwide can try to install it on their systems. Since up to now the package was tested only on the DD's machine, its status should considered *alpha-testing* and the repository is called the *unstable distribution*. However, notwithstanding the scaring name, a considerable number of users (and virtually all the DDs) tries packages from the unstable distribution, thus the test is quite significant. If a package lives in the unstable distribution for ten days without any critical bug is notified, it is *automatically* uploaded to another, more stable, repository corresponding to a *beta-testing* status. This repository is known as the *testing* distribution. Approximately yearly, a *Release Manager* is appointed by the DL, and starting from the testing distribution a set of packages is *frozen*. This means that no new packages can be added to the set, included packages evolve only for bug correction, and eventually, when all *release critical bugs* are corrected, a new *stable distribution* is released to the public. The stable distribution is what is normally considered the official Debian distribution and included packaged are updated only for fixing security vulnerabilities.

It is worth noting that DDs normally produce their package on a specific architecture (the most common is i386). However, unless the package control file specifies explicitly that its use is restricted to a single architecture, every package inserted in the unstable distribution is automatically build for all the architectures considered by Debian (eleven, to date) and it can enter in testing only if the build process is successful.

3.2. Coordination

The goal of obtaining a coherent distribution where all programs can interact smoothly is a very complex one. The problem seems without a solution if a distribution is obtained by aggregating thousands of packages produced by hundreds of developers on dozens of different systems configurations. Nevertheless, Debian systems were able to obtain a quite good overall user satisfaction, as testified by several awards won in 2003 (Linux journal readers' choice, Linux enterprise readers' choice, Linux new media award). In fact, the main effort carried on by DDs is directed to ensure that their packages are fully compliant to Debian *policies*.

Policies are key in the Debian approach to software distribution. Freedom of DDs is unlimited as long as they comply to their collectively agreed policies. Policies are often based on international or community standards (e.g., the Filesystem Hierarchy Standard [3]) and they concern all the global issues that affect the coherence of a system: i.e., libraries deployment, environment variables,

4 A few complex applications, i.e., the XFree86 package, are maintained by a team of four or five people

shared services, scripting languages. They sometimes take the form of general principles ("Maintainer scripts must be idempotent"), but more frequently they assert some automatically checkable property of the installed package ("Link targets like foo/../bar are deprecated"). For complex subsystems special sub-policies exist: for example, the Emacs extensible editor has its own policy that reduce possible conflicts among the huge number of emacs-specific packages coming from different sources.

Policy enforcement is pursued at different levels, in order to exploit cross validation to minimize inconsistent packaging:

- during package assembling: most of the policies are associated to a tool (collectively called "debhelpers") that ensures the correct application. For example, documentation can be introduced in a package by using the script *dh_installdocs*; it guarantees that when the package will be installed, the documentation files will be put in */usr/share/doc* and compressed with *gzip*.
- during package testing: several tools exist to check policy compliance before uploading the package to the public repository. The most important one is *lintian*, a script that analyze a package for about thirty categories of policy violations. Moreover, when a packaged is uploaded to a public repository some critical checks are repeated and the package is refused if checks fail.
- during package deployment: every user that detects an incoherence can issue a bug with an automated procedure (*reportbug*). Since policies are public and available on every Debian system, also not harmful violations can be in principle discovered (and bug reports show that they often are) and notified to DDs.

Every package implicitly assumes a working environment providing to it some services. DDs should make explicit these assumptions by defining a set of *dependencies* for each package. The richness of Debian dependency language enables fine tuning of installed systems: if A *depends* on B, B must be installed in order to install A; if A *recommends* B, most users would not run A without B; if A *suggests* B, B may enhance A functionalities, but A can be used in most cases also without B. Moreover, two packages can *conflict*, a package may *replace* another, and a package A can *provide* the functionalities of B. The latter relationship makes useful the existence of *virtual packages* (e.g., a generic mailer application) that can be required by others. In order to foster reuse and avoid duplications, Debian promotes *micro-packaging,* therefore it is common that from a single source package several binary packages are generated. Thanks to these dependency relationships, installing a new application on a running system can be as painless for users as typing a "apt-get install application" command: all the required packages are retrieved from a public repository (possibly on a set of CDs), installed and configured. In most cases even running services can be upgraded in this way, since Debian policies define standard mechanism for stopping and restarting daemons. Moreover, when an application is removed, it is possible to check which libraries were "orphaned" (i.e., they are no more requested by any package) by this removal and remove them too.

4. Customizing and Mantaining a Debian System

One of the added value of open source systems is that they can be customized to better satisfy user needs. However, customization is also risky. A highly customized system can be very difficult to keep in sync with the mainstream open source development. Suppose for example that a user wishes to use a program *java-local* rather than the program *java* provided by the Debian "java" package. If the user overwrote */usr/bin/java* with *java-local*, the package management system will not know about this change, and it will discard the customization on upgrades. For this reason, Debian introduces the concept of *package diversion*, by which users can maintain their *diverted* versions of programs, while enjoying mainstream upgrades. For example, by issuing the command *dpkg-divert --divert /usr/bin/java.debian /usr/bin/java* all future installations of the Debian "java" package will write the file */usr/bin/java* to */usr/bin/java.debian*. Moreover, several alternative equivalent programs can be installed in a system and simple infrastructure can be used to keep a generic name linked to the preferred alternative (e.g., *x-www-browser* may point to *galeon*, even if both *mozilla* and *galeon* are installed. These facilities make Debian systems ideal to be used as a starting base for specialized distributions: successful examples are the Knoppix distribution (running entirely from a CD), and the Familiar distribution (intended to be run on PDAs): while very different, they all share the same packet infrastructure and they keep reusing the daily work of DDs notwithstanding their customizations.

Another problem that sometimes hurdles users in upgrading their customized systems, is that configuration options can be discarded by the new version of applications. Roughly speaking, the configuration of an application is a three steps process. Major options are set system wide when the application is installed; they can affect major issues (for example, how a program is started: if it is an *inetd* daemon or if it is *SUID root*) and they are only rarely modified. Other less important options are more frequently changed by editing configuration files. User options are changed by users themselves and settings are stored in their home directories. In Debian systems, preservation of major options across upgrades is achieved by exploiting the *debconf* database. When a new application is installed for the first time some questions are asked to the user. The answers provided by the user are stored in that database and when a new version of the application is going to be installed only new options are presented to the user. Users' choices are preserved for unchanged options and the installation script is responsible for traducing them in

the possibly new syntax of configuration files. Moreover, every time an upgrade affects a system wide configuration file, a warning is issued, asking which version the user wants to keep and, if the files are human readable text files as it is common in the Unix world, differences can be merged together. Instead, no support is provided at the moment to evolve end-user options. However, these are often just cosmetic ones and therefore much less critical.

Another approach is worth mentioning in this paper is what can be called *aspect oriented package maintenance.* In any system complex enough, there are issues that *cross-cut* the whole system and cannot be easily packaged in an isolated module. The Debian solution to this problem follows an aspect oriented approach: special events of the package life cycle are exposed to other packages and they can, *obliviously* from other packages points of view, introduce actions that will be performed when these events occur. For example, the *localepurge* package aims at not installing all localized files (i.e., files specific for different languages) not explicitly preserved by the user of the system (it is a big waste of space to install non useful Japanese documentation files if nobody reads Japanese!). Other packages know nothing about *localepurge*, but, when it is installed, its execution is needed during their installation. Therefore, the package installation system (clearly a cross-cutting issue) can be customized by specific programs, that may subscribe themselves to be executed when well defined events occur (basically installation and removal of a package)

5. Conclusions

The goal of obtaining a coherent distribution of software packages where all programs interact smoothly increases its complexity with the number of applications, the number of architectures involved, and the number of system configuration supported. The Debian project copes with this complexity with an approach that does not resemble neither the *cathedral* model with a single architect with unlimited power, nor the *bazaar* model where the only coordination force is mutual interaction. Instead, freedom of action is preserved, and a democratically decided coherence is pursued as far as possible by technical means. The Debian coordination effort to manage complexity and heterogeneity should be studied in depth in order to understand which techniques can be applied conveniently also to commercial organizations.

Acknowledgments

I would like to thank all the members of the Debian project for their voluntary work. I am a member of the Debian project since 2000. However, the opinions expressed in this paper, except the ones taken from cited official documents, are my own only responsibility.

6. References

[1] *Linux weekly news*. http://www.lwn.org/distros, 2004.

[2] F. P. Brooks. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley publishing company, USA, anniversary edition, 1995.

[3] Freestandards.org. *Filesystem hierarchy standard*. http://www.pathname.com/fhs/, 1998.

[4] B. Garbee. *Where would you like 100,000 users to go today.* http://www.gag.com/~bdale/talks/2004/adelaide/keynote/bdaleLCA.html, 2004. Linux Conf. Australia, Adelaide.

[5] Open Source Initiative. *Open source licences*. http://www.opensource.org/licenses/, 2004.

[6] A. Mockus, R. T. Fielding, J. Herbsleb, *Two case studies of open source software development: Apache and Mozilla*. ACM Trans. Softw. Eng.

[7] I. Murdock. T*he Debian manifesto*. http://www.debian.org/doc/manuals/project-history/ap-manifesto.en.html, 1993.

[8] Debian Project. *The Debian constitution*. http://www.debian.org/devel/constitution, 1998.

[9] E. S. Raymond. *The cathedral and the bazaar*. http://www.tuxedo.org/esr/writings/cathedral-bazaar/, Nov. 1998.

[10] R. Stallman. *The GNU manifesto*. http://www.gnu.org/gnu/manifesto.html, 1985.

In the network: Distributed control in Gentoo Linux

Thomas Østerlie
Department of computer and information science
Norwegian University of Science and Technology
thomas.osterlie@idi.ntnu.no

Abstract

This position paper reports on the findings of an empirical pilot study of Gentoo Linux. Gentoo Linux is an open source Linux distribution developed by a geographically distributed community of volunteers. The reported findings are based on the analysis of a specific episode using actor network theory. With basis in the analysis, it is argued that control in this specific episode can be interpreted as both distributed and local at the same time. Control here being the power to define a problem and make the decision about the appropriate solution to the problem defined. Control, it is argued, is distributed in that it is the function of reciprocal influence among several human and non-human actors. Furthermore, it is argued that control can be interpreted as not inherent in organizational structures or hierarchies, but locally embedded among actors in the decision making process.

1. Introduction

Geographical distribution is one of the distinct characteristics of open source software development. Open source software development has been connected with teams of geographically distributed developers ever since Raymond's first description of the bazaar [1]. Despite the geographical distribution of developers, Raymond describes control in the bazaar as centralized, headed by the 'benign dictator'. Using open source software development as an example of computer-supported distributed work, Moon and Sproull [2] argue that an enabling condition for the success of the Linux kernel are the "capabilities a single leader brings to a project". They argue that the "clear locus of decision-making, singular vision, and consistent voice" are important in controlling this kind of collaborative effort. This supports Raymond's notion of the 'benign dictator'. Control in these two works is therefore understood as centralized.

Mockus and Herbsleb [3] describe the Apache open source web server community in two contradictory ways. On the one hand there is a formal organizational structure for making decisions about code integration. On the other hand, they report that work is not assigned but that individual developers choose what to do themselves. "The choices are constrained, however, by various motivations that are not fully understood." Understanding control as the power to define problems and their appropriate solutions, and thereby making decisions about what tasks to prioritize, Mockus and Herbsleb's description points to a tension between centralized and distributed control.

Picking up on Mockus and Herbsleb's observation, this paper raises the question whether control always is centralized in open source software development? How can we understand the tension between distributed and centralized control?

The paper is organized as follows. Section 2 presents the empirical findings. The section contains a short presentation of the Gentoo Linux case, details of the method employed, and a detailed presentation of the reported episode itself. Section 3 discuses how control can be interpreted in the reported episode. The conclusion draws implications of the discussion, and formulates directions for future work.

2. The case

This section presents the empirical findings. For context, an overview Gentoo Linux is presented first. Then the methods of data collection and analysis that form the basis for this position paper are described. The reported episode is described afterwards, after which the episode is analysed in terms of the mechanics of framing the problem to be solved and what actors take active part in framing the problem.

Gentoo Linux is an open source Linux distribution developed by a geographically distributed community of volunteers. Aiming for advanced users, the distribution is a mix between Linux from scratch and a regular Linux distribution. Gentoo Linux provides the minimum of support for installing a bare bones Linux system. In this way the user can build an installation from the bottom up, tuning it to his exact needs; be it a workstation

installation, a secure server, or a gaming system. That is why Gentoo Linux is also called a meta distribution.

Portage, Gentoo Linux' software distribution system, is the technology that makes this possible. Portage keeps track of the third party software, also called packages, available for Gentoo Linux at any one time. At the time of writing there are over 6000 packages available. Portage also keeps track of which packages have been installed on the local system. Information about installed packages is stored in a database. For each installed package this database contains information such as the absolute path for every files installed by the package, the compiler flags the package was built with, and the package's license.

When installing new packages, Portage compiles the software on the local system. The user can therefore fine-tune such things as compilation flags and additional software support. This information is stored in a set of configuration files.

2.1. Method

The episode reported in this position paper is part of the empirical evidence collected during a pilot study of Gentoo Linux. Data for this pilot study was collected with a number of methods. Archival data was collected from the Gentoo web site at `http://www.gentoo.org`, and from the Gentoo mailing list archives accessed through the `news.gmane.org` service. The IRC logs that form the basis of the analysis which this position paper is based on, were downloaded from Gentoo's home pages. In addition, the pilot investigation involved participatory observations with a software consultant using Gentoo Linux as development platform, and a semi-structured interview with one of the Gentoo Linux developers. The interview was performed according to the guidelines laid down in [4]. Ethnographic field notes [5] were taken in connection during the participatory observation and later written out as a full field report

The episode reported in this position paper is primarily based on the IRC log of the Gentoo managers' meeting from December 15 2003. Using actor network theory, an analysis was performed on basis of the log supplemented by the interview. Actor network theory is a method for analysing the relationship between the technological and the social [6,7]. Unlike traditional software engineering methods that teaches us to categorizes entities into classes such as roles, instances, technical artefacts, organizational artefacts, just to mention a few, actor network theory attributes symmetry to all entities in the network by promoting them to actors. This reflects the basic assumption that all entities in the network are capable of acting upon each other.

Central to actor network analysis is identifying the actors and associations between them. Thinking of actors as nodes and associations as connections between the nodes, the network appears. The network is composed of heterogeneous nodes—technical and non-technical, human and non-human, etc.—that are associated for a period of time. However, the actor network is reducible neither to an actor alone, nor to a network. In addition, the network is seen as constantly shifting, and not as a representation of the original or final state.

In actor network theory the network is an analytical structure constructed by the analyst. Instead of thinking of the actor network as a representation of things out there, it is a conceptual frame, a perspective to interpret social and technological processes. The episode reported in sections 2.2, 2.3, and 2.4 is related as interpreted through the perspective of actor network theory.

2.2. The episode

The Gentoo managers' meeting is a biweekly meeting for Gentoo developers to coordinate activities. The managers' meeting is arranged over the Internet, using IRC. During the Gentoo manager's meeting December 15 2003 [8], the issue of third party utilities operating on Portage's database and configuration files is discussed. Some of these utilities mangle the configuration files, while other utilities no longer work because the Portage database format has changed. One of these utilities, `qpkg`, a utility for querying Portage's database, has accumulated over 20 unresolved bug reports in Gentoo Linux' bug tracking system. The source of all these problems is identified to be code that is out of synchronization with the rest of the system. This kind of problem has been resolved before by introducing the maintainer role. The maintainer is responsible for keeping specific parts of code in synchronization with the rest of the system. The conclusion is that the code in question is outdated because it has not been assigned a maintainer.

An additional response to the problem is to introduce an abstraction layer, an API, on top of Portage's database and configuration files. All utilities accessing the configuration files and database must do so through this API. Two Gentoo developers are assigned to develop and maintain this API.

There is dissent among the participants at the meeting about priorities. Gentoo Linux' chief architect proposes to base the API on Portage's own code. The two developers in charge of the API, while agreeing that this would be a good idea, argue that there are other factors that are more important to take into account when resolving the problem. Especially the issue of missing maintainers for utilities accessing the Portage database and configuration files. The `qpkg` utility is used as an example of these difficulties. The utility was included in the distribution by a developer who later left the project. `qpkg` implements its own code for accessing Portage's database. Responsibility for the utility was handed over to someone

else when the original developer left Gentoo Linux. This second developer went on leave, and `qpkg` was left unmaintained. The problem, while technical in symptoms, is something more and something else. It is also symptomatic for the problems to be addressed by the API developers, in that `qpkg`, like the other utilities, implement its own code for accessing Portage's database and configuration files directly. Without any guarantee for how long the developers for these utilities will stick around Gentoo Linux, the situation that the API is to address is to keep the way utilities access Portage's database and configuration files synchronized even after the original developers leave.

2.3. Framing the problem

The decision to introduce an API on top of Portage's database and configuration files is an answer to a problem the Gentoo developers want to solve. Thinking in terms of actor networks, the problem can in fact be conceptualized as an actor. However, it is not an actor that exists before the meeting starts. It is actually a constructed actor. The problem is "a list of ... trials ... hooked to a name of a thing and to a substance" [7, p.122]. The way the problem is given substance, its framing, is the topic of this section.

In the transcript from the Gentoo manager meeting December 15 2003, one of the developers participating in the meeting states that there are a "slew of util[itie]s lying about". He associates these with mangled Portage configuration files, in that the utilities "hack, slash and mutilate the ... config[uration] files". Then he associates the Portage database with the "util[itie]s lying about", as "these util[itie]s misreads /var/db [the Portage database, author's comment], so as not to be consistent with [P]ortage". Another problem with the "util[itie]s lying about" is that they have overlapping functionality, and none do their tasks particularly well:

"*we don't need five half-working use flag editors. we need one really good one*"

The problem is framed by the developer associating different actors, framing a problem in such a way that the other developers understand it as their problem, too. Figure 1 illustrates how the different actors are associated in framing the problem.

Having framed the problem as a shared problem, its cause is established. The cause of the problem is that the utilities lying about have not been properly updated, as "a few of the existing tools [the same as the utilities lying about, author's comment] don't work with portage 2-0.50 due to API changes [in Portage, author's comment]". That is also why the `qpkg` utility does not work any longer, since there are "20+ bugs [reports] about `qpkg`" that remain unresolved in the bug tracking system. The technical cause of the problem is outdated code, but this is more a representation of the larger problem:

"*now I have 20+ bugs about qpkg assigned to me, it's a mess, and nobody wants to touch it. Who is responsible to maintain it now?*"

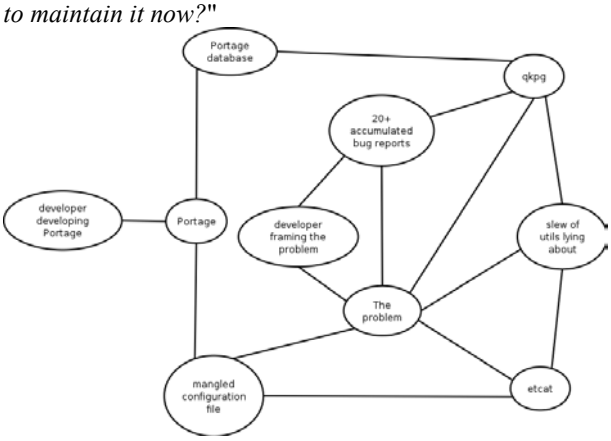


Figure 1 The problem framed

The symptom is that the utilities lying about have not been updated, but this is caused by the fact that there are no one maintaining the "slew of util[itie]s lying about". In this way, the maintainer replaces the problem in the actor network, providing a solution to the situation.

Control is exercised in deciding what activities are to be undertaken, how and when. There are hundreds of unresolved bug reports in Gentoo Linux' bug tracking system. In making the decision about which of these bug reports are to be resolved, decisions about what activities to prioritize are made. Framing the problem can therefore be understood as the power to determine the activities to be undertaken. From this follows that the task of identifying who is in power in the episode above, is the task of identifying who has the power to frame problems.

2.4. Who frames the problem?

At first glance, the problem facing the developers seems to be framed by one of the developers participating in Gentoo manager meeting. As a response to the problem the maintainer role is introduced. The maintainer role, as an actor decoupled from a person, was once constructed to resolve similar situations. In framing the problem at hand in this particular way, the answer to introduce a maintainer becomes a given. Following this line of thinking, one can go as far as saying that the maintainer role participates in shaping the problem. If you have a hammer, all you see are nails. The knowledge among discussion participants that this role exists can be considered constitutive to the problem framing. Looking at the episode this way, the maintainer role is turned from passive to active in framing the problem.

It is highly unlikely that every bug experienced by Gentoo Linux users is reported in the bug tracking system. However, the bugs that are used to frame the problem are those reported in the bug tracking system. Bugs are given priority, severity, status, and assigned to a given person or group of persons for resolution. A bug is resolved when it is fixed or labelled invalid. As long as a bug remains unresolved but assigned to a developer, the bug is a reminder to the assignee. In this sense, bug reports are also active in framing the problem.

Framing the problem is not a function of a single developer or a closed group of developers. Instead, it can be interpreted as the function of a number of actors, both human and non-human. Neither is the power to frame the problem one-sided in that one actor forces other actors to do something they do not want to. Instead, framing is a reciprocal relationship between the Gentoo developers, the maintainer role, and the bug reports.

3. Discussion

This discusses how control can be interpreted in the above episode above. Three aspects of control are discussed. First the implication of the episode in terms of control and organizational hierarchies is discussed. Then we discuss how control can be interpreted as distributed among human and non-human actors. Finally, it is argued that actor network theory makes the interpretation of control as reciprocal among actors likely.

3.1. Relation of control and organizational hierarchy

Gentoo Linux is split into projects and sub-projects. Herds consisting of maintainers are responsible for keeping a set of packages up to date. This is how the Gentoo developers describe their organization in terms of hierarchies and distribution of roles. However, by conceptualizing the way the Gentoo developers talk about the organization during the Gentoo Managers' meeting as an actor network, another view appears. In framing the problem that the API resolves, the maintainer is introduced as an actor in the network. In contrast Gentoo Linux' chief architect does not get through his idea to base the API directly off Portage.

Looking at the organizational hierarchy, the architect is placed farther up than the developer. If control and organizational hierarchies were related, the chief architect would have the power to make his view the prevailing. In the episode above, this does not happen, though. Why not?

Control can be understood as local in the way actors enrol other actors and are enrolled themselves in the immediate actor network. If control was inherent in the hierarchy, the chief-architect should have gotten his view through. That he does not get his view through can be explained by him never enrolling the chief architect role, considered an actor in an actor network analysis, in the immediate actor network.

The implication of the above interpretation is that there need not be an inseparable relation between organizational hierarchy and control. Control can be locally embedded among actors in the immediate network. The actors brought together by the hierarchy have no essential relation to each other, but can instead be understood as dispersed actors temporarily brought together through the hierarchical ordering. By viewing of actors as inherently dispersed, thinking of the organization as an actor network shows that the hierarchical description of organization is just that: a hierarchical description of organization, an abstraction. As such organizational hierarchy need not be inherently connected with control.

3.2. Control is distributed and heterogeneous

In saying that a corrupted configuration file is the same as a missing maintainer, technical (the corrupted configuration file) and organizational (the maintainer) actors are treated as equals. By treating all actors symmetrically this way at the same level of analysis, control can be interpreted as the mutual relationship between heterogeneous actors. Control is not the relationship between action and structures of signification, legitimization and domination [9], but in the direct relationship between actors in the network. A possible implication of this interpretation is that control is no longer purely social, but a function of human and non-human actors, of technological and non-technological actors, of organizational and non-organizational actors. Control becomes orthogonal. It is a function between all actors in the network, regardless of classification schemes. Actors are no longer higher or lower in the organizational hierarchy, technical or non-technical, human or non-human; they are all and the same: actors in the network.

3.3. Control as reciprocal

In saying that control can be understood as local to the immediate network of actors, control becomes both the actors' ability to frame problems, and the ability to limit other actors' framing activities. Control can therefore be understood as more than the traditional control relation within a set of actors

A→B
C→D
D→B
A→E

but as a relationship where actors reciprocally control each other, understood as the relation of

(A, B, C, D, E)

In the latter relationship lies the argument that control is distributed. Control can't be reduced to an actor A's ability to overcome actor B's and thereby exert control over B, as implied in the relationship A→B. It is not one-sided, but distributed. A must not only overcome B's resistance, but the resistance of the other actors in the immediate network. In this sense, in exerting control over B, A exposes itself to the controlling power from the other actors.

4. Conclusion

This paper has argued that traditional notions of control may be inadequate in describing distributed control in Gentoo Linux. Control, it is claimed, need not be limited to the people who seem to be making decisions. Rather, control can be interpreted as distributed among both human and non-human actors. In reported episode, control is distributed among a number of Gentoo developers, the maintainer role, and bug reports. In this sense, control is not distributed in terms of geographical distribution, but distributed as in shared among a handful of human and non-human actors.

While Gentoo Linux is geographically distributed, the interpretation of distributed control is not connected with the geographical distribution. It is, rather, connected with the distribution of elements within an actor network. The key points of distributed control are:

a) that control need not be inherent in the organizational hierarchy, but can be interpreted as embedded in the immediate actor network
b) that control need not be inherent in structures, but can be distributed among actors,
c) that control can't always be reduced to a function of human agency, but may at times be understood as the function of all actors in the network such as tools and organizational roles
d) that control can be a reciprocal relationship between a set of actors

Thinking of distribution this way, similar analysis of distributed control could therefore be equally applicable in geographically co-located software development efforts, too. Distribution is not geographically, but instead understood as distributed among actors.

In arguing that control is distributed in Gentoo Linux, this position paper addresses only the mechanics of control through following the construction of networks through enrolling. The rules of this construction are left untouched. How is it that some actors in the network inscribe stronger behaviour than others? What are the rules for enrolling actors, and what are the rules for excluding actors as valid to be enrolled? These issues need to be addressed in future studies.

The decision to do an API on top of the Portage database and configuration files were only a month and a half old when this pre-study was done. At the time of writing, the API has still to be integrated in a large scale. It is available in Gentoo Linux, but very few utilities actually use the API. A point of future study is to follow up how the implementation of the API and its integration with utilities goes. How is access through the API enforced? How are bugs connected with not using the API handled? What are the effects of introducing the API? Does it lead to lesser problems for utilities integrating with Portage's database and configuration files?

7. References

[1] E.S Raymond, *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*, O'Reilly, Sebastapol, 1999.

[2] J.Y. Moon, and L. Sproull, "Essence of Distributed Work: The Case of the Linux Kernel", First Monday, 5:11, 2000.

[3]A. Mockus, and J.D. Herbsleb, "Why Not Improve Coordination in Distributed Development by Stealing Good Ideas from Open Source?", Proceedings of the 2nd Workshop on Open Source Software Engineering, IEEE, 2002.

[4] S. Kvale, *InterViews: An Introduction to Qualitative Research Interviewing*, SAGE Publications, New York, 1996.

[5] R.M. Emerson, R.I. Fretz, L.L. Shaw, *Writing Ethnographic Fieldnotes*, University of Chicago Press, Chicago, 1995.

[6] M. Callon, "Some elements in a sociology of translation: domestication of the scallops and fishermen of St. Brieuc Bay". *Power, Action and Belief*, Routledge, London, 1986.

[7] B. Latour, "Technology is society made durable", *A Sociology of Monsters. Essays on Power, Technology and Domination*, Routledge, London, 1991.

[8] Gentoo Managers' Meeting Log, `http://www.gentoo.org/proj/en/devrel/manager-meetings/logs/2003/20031215.txt`, last accessed March 1 2004.

[9] A. Giddens, *The Constitution of Society: Outline of the Theory of Structuration*, Polity Press, Cambridge, 1984.

A Conflict Detected Earlier is a Conflict Resolved Easier

Anita Sarma and André van der Hoek
Department of Informatics
University of California, Irvine
Irvine, CA 92697-3425 USA
{asarma, andre}ics.uci.edu

Abstract

Open Source development is highly distributed and parallel in nature. There are no definite boundaries, either for people or from where they work. This high level of parallel, distributed development leads to conflicting changes made concurrently by different developers. Because OSS developers lack the kinds of informal coordination opportunities that collocation offers, OSS developers must rely on mailing lists, discussion groups, and tools such as CM and bug tracking systems to try to manage their parallel efforts such that conflicts do not occur. Unfortunately, these coordination mechanisms are not adequate: it still regularly happens that parallel changes interfere, either via direct overlap or indirect, semantic conflicts. In this paper, we build upon our previous work in raising awareness as a mechanism to support better coordination among developers, and introduce a new integration of our Palantír tool with Eclipse as well as a new visualization of parallel work that we believe is especially useful in Open Source settings.

1. Introduction

Open Source development is highly distributed and by its very nature a multi-person, team effort [1, 2]. One of the key aspects of any Open Source project is that there are no definite boundaries for people – who participates – and for location – from where they participate. Although there is typically a small group of core developers, other developers enter and leave the project on will, and may or may not contribute significant changes. Usually there are no set release dates; the software is released when the core group thinks that it is time. This development style is very different from the traditional corporate development style in which a predefined group of developers follows a carefully-defined process to develop a software system according to a predefined timeline. We make two observations regarding the style of OSS development as compared to traditional software development.

1. It has been shown that collocation helps developers in getting an idea about their co-developers work [3, 4] Coffee hour talks or just casual conversations provide the context that helps developers gauge such things as what problems their co-developers are addressing and the schedules under which they are operating. This context helps developers in informally coordinating their efforts and code changes [5].
2. It has also been shown that parallel development leads to conflicting changes [6]. Some changes can be resolved with the help of automated merge tools [7], but when changes overlap or, worse, have semantic aspects, they have to be manually resolved, often a time consuming and costly effort.

From these two observations we can infer that Open Source development, by its very nature of being distributed and involving parallel work, will lead to the presence of conflicting changes that cannot be caught by the same kinds of informal interactions as present in traditional software development. OSS, in a way, is the epitome of parallel development: anyone who is interested in a project can work on anything that interests them at any point in time.

To compensate, most OSS developers extensively use such tools as e-mail, discussion lists, bug tracking tools, and CM tools to "informally" coordinate their work with ongoing and past activities. This clearly helps, and many problems are avoided. However, it also is somewhat of an undesirable situation as it is labor intensive, requires manual action and interpretation, and typically is based on "old" data – data about events that have already happened. Take CM systems, for instance. These systems manage parallel work by partitioning work into isolated workspaces. Developers make their changes in isolation and periodically resynchronize their work with the contents of the repository. This is done either by checking in the files that they have modified, or by synchronizing the files in a workspace with newer versions that other developers have committed to the repository. Conflicting changes, thus, are discovered only *after they have been introduced*. Some of these conflicts can be merged with the help of automated tools [8], but in the event of over-

lapping changes (either directly, as in changes to the same lines of code, or indirectly, as in changes that are in different parts of the code but semantically interfere) merge tools fail or produce false results and the conflicts have to be manually resolved [7].

The problem is that in current CM systems it is not possible to get an overall picture of the extent of parallel changes in different workspaces and how those changes relate to each other. The isolation enforced by the CM systems limits developers in knowing "what is going on", and thus takes away the information upon which they could decide to have informal "online hallway conversations".

To address this problem, we built Palantír, a CM workspace awareness tool that deliberately breaks the isolation of workspaces. Palantír operates by providing developers with an at all times up to date picture of any relevant changes in any of the parallel workspaces. The availability of this information gives developers a chance to be aware of "problematic" concurrent parallel activities, and coordinate with each other or even adjust ones own activity based on this information. Developers can detect potential conflicts *as they are occurring*.

In our previous work [9], we showed that Palantír can easily be used to instrument existing CM systems with awareness (we integrated Palantír with RCS [10], CVS [11], and Subversion [12]; each integration was less than 1000 lines of code). Moreover, we concentrated on building a series of visualizations that each have different properties of how conflicting changes are presented to the developer [13]. In this paper, we concentrate on the applicability of Palantír in the Open Source domain. In particular, we introduce a new view (which we call the full project view) that presents a project-wide overview of all workspaces and introduce our integration of Palantír with Eclipse. The first helps in giving the core set of developers of an Open Source project an idea of all individual efforts currently underway, the second demonstrates that Palantír can be integrated into a representative development environment and helps those developers using Eclipse with a minimally obtrusive awareness mechanism.

The remainder of the paper is organized as follows. Section 2 discusses our previous work and the high-level approach that Palantír follows. Next, we present the new full project view in Section 3. Section 4 introduces the integration of Palantír and Eclipse. We discuss related work in Section 5, and conclude in Section 6 with an outlook at our future work.

2. Palantír to Date

Palantír is based on the hypothesis that conflicts in parallel development can be considerably reduced, both in magnitude and number, by providing developers with an insight into ongoing changes in parallel workspaces. Instead of solely relying on the coordination mechanism of CM systems (i.e., the process of checking in and checking out artifacts, with or without a necessary merge step), Palantír provides developers with information so they can detect potential conflicts as they start occurring, and can self-coordinate their activities to avoid them or resolve them before they grow too large.

Palantír is not another CM system, but is a workspace awareness tool that builds on top of existing CM facilities and concentrates on the collection, distribution, organization and presentation of relevant workspace information. Palantír does not change the way developers interact with the CM system (i.e., the commands with which they interact with the CM system), but instead silently intercepts activities and shares relevant information regarding those activities.

We briefly discuss the components of Palantír below (a full discussion of Palantír, its architecture, visualizations, and integration with several existing CM systems can be found in [9]).

A *workspace wrapper* collects configuration management activities and translates those to Palantír events. These events are then distributed by the Siena event notification service [14], which we have adopted as our mechanism of transmitting information from workspace to workspace. Since different CM systems have different access mechanisms, a workspace wrapper is designed specifically for a particular CM system and must interface with the existing mechanism of user interaction (for instance, through wrapping existing commands or leveraging the trigger capabilities present in some CM systems).

The *internal state* stores and organizes all events, the relevant set of which are then extracted by the *extractor* before being displayed by one or more *visualization* components. Currently we have implemented four different visualizations (ticker tape, tabular view, explorer view and the fully graphical visualization), two of which are presented in Figure 1. Each visualization shows, in its particular form, specifically which artifacts are being changed by which other developers. In addition, they present a measure of severity of those changes: for each change, Palantír calculates the percentage of the artifact that has changed as a rough measure indicating "how much" potentially interfering activity is occurring in another workspace.

In the *explorer* view, shown in the left hand side of Figure 1, the artifacts in the local workspace are shown organized in the familiar form of an expandable tree. This tree is enhanced with vertical bars indicating the severity of ongoing and committed changes: the longer the bar, the higher the severity of the change. Although not visible in this black and white view, changes are color coded to distinguish changes in a local workspace from changes in other, "remote" workspaces.

The fully graphical visualization, shown on the right hand side of Figure 1, presents a developer with a hierarchical "3-dimensional" stackable view of an artifact and its constituents (in this case version 1.1 of the folder home/word"). Each constituent artifact may itself contain other artifacts and each artifact in the view may be present and being changed in multiple workspaces (as indicated by stacks of artifacts). Color-coding separates different workspaces. For instance, the stack for the artifact "/home/word/edit" indicates that Ellen, Pete, and Mike each have a version of the artifact in their workspace. Pete and Mike each have version 1.0 in their workspace, and their changes are still in progress as indicated by the question mark. Ellen, on the other hand, already has checked in a new version of the artifact (as indicated by the exclamation mark), resulting in her having version 1.1 in her workspace. The severity of ongoing and committed changes is shown by the vertical blue lines: the taller the bar, the higher the severity.

3. Full Project View

Thus far, the visualizations of Palantír have been geared towards individual developers. Each visualization only shows concurrent changes that relate to artifacts that are in the "local" workspace. We realize that only having such self-centered views is a limitation and that, benefits exist to having an overarching view of all parallel activities. In particular, we believe project managers in regular corporate settings as well as core programmers in Open Source settings can benefit from having a view in which all concurrent workspaces are shown and potential conflicts among workspaces are displayed. The needs of project managers are different: they need both a high level and detailed view of parallel activity to better handle task management. Similarly, core developers in open source projects can benefit from knowing who is working on what, what changes may be forthcoming for future integration cycles, and generally steer developers who are working on related projects towards each other.

We are currently building a new visualization for seeing the entire project at a glance. This visualization, an early prototype of which is shown in Figure 2, provides an overview of changes in all the workspaces. In particular, it lays out all the workspaces in the project in a circle. If there is any overlap of work in any pair of workspaces (as defined by changes to the same files), this overlap is indicated with the presence of a line between the two workspaces. The higher the overlap, the thicker the lines appear. In order to avoid cluttering the visualization, the manager can set a threshold for viewing the overlaps. For example, one may want to only see overlaps that are greater than 40%.

Anyone using the full project view can zoom into any of the workspaces to view the changes in that workspace. Specifically, zooming brings up the Palantír explorer visualization discussed in the previous section. Double clicking on a developers name in the visualization fully opens that developers' workspace in a separate window. This allows easy exploration of conflicts, by showing workspaces side by side. Managers can guide a project; core developers can point novices to things they may have missed or are doing wrong. In turn, novices can use the view as a rudimentary kind of "expertise" browser [15], in the sense that they can see who is working on

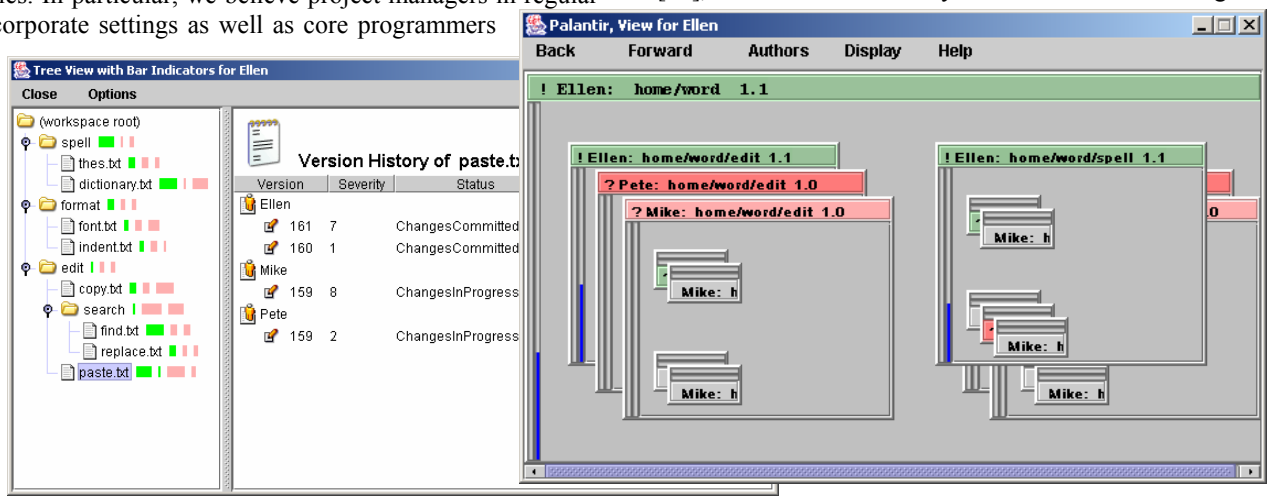


Figure 1. Explorer Visualization and Fully Graphical Visualization.

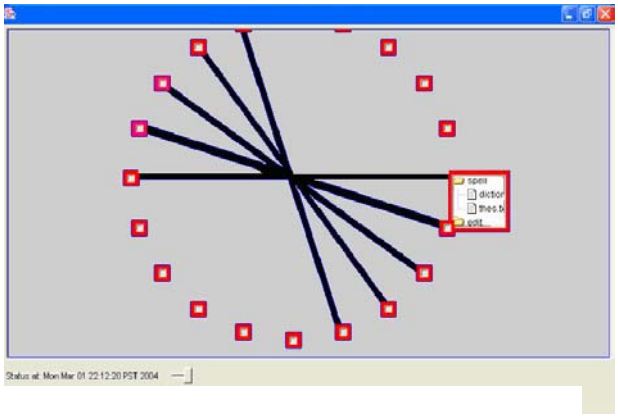


Figure 2. Project View Visualization.

what parts of the code.

A particularly interesting feature of the full project visualization is that it allows the user to go "back in time". By using the "time" slider at the bottom of the window, it is possible to view the level of conflict at regular past intervals. This helps managers and any other party using the visualization to gain an understanding of how a project evolves and the level of parallel conflicts during that evolution.

We are currently still in the building stages of the project management view, but have high hopes that it will be one of the most informative views of Palantír yet – it can be used by anyone, and provides interesting information at both a high and a very detailed level.

4. Eclipse Integration

So far Palantír has been independent of any development environment and runs as a separate application. On the one hand, integration with a particular development environment is not good, as different developers have different preferences and integration with a particular editor limits the use of Palantír. On the other hand, when a developer is deep into programming they might not switch to any other applications until their task is done to avoid any kind of distraction. Thus, in the case of Palantír, the developer might be missing out on important information during that time. To avoid this context switch between the editor and Palantír, we have integrated Palantír with Eclipse, a Java development environment. We are aware that different developers have different preferences and Eclipse may not be the editor of choice of all developers. The purpose of our integration is therefore to demonstrate that it is possible to easily integrate Palantír into a development environment; we hope other integrations can be as simple as this one.

We decided to integrate Palantír with Eclipse specifically because Eclipse is itself an Open Source product geared towards OSS development. Since its initial release Eclipse has become increasingly popular as the Java editor of choice. Additionally, our choice was guided by the fact that Eclipse has been designed such that adding new functionality is relatively easy; it's plug-in facility is very powerful and the number of extension points is great.

Our integration is provided through a new Palantír perspective in Eclipse, similar to the already existing Java perspective. Shown in Figure 3, the perspective lays out all the artifacts present in the local workspace in an expandable tree view. Each artifact is annotated on the left hand side with a small red triangle denoting the cumulative severity of changes made by other developers: the higher the cumulative severity, the larger and redder the triangle. On the right hand side of the artifact name, the numerical value of the cumulative severity is listed in text as well. A similar mirror image of a blue triangle is used to denote the impact of the changes.[1] This view presents only cumulative values, developers should use the other Palantír visualization for in-depth investigations.

Note that the Palantír-Eclipse integration recalculates the severity of changes every time a file is saved: this allows Palantír to share up-to-the-minute details regarding activities in other workspaces.

5. Related Work

To date, CM systems have very limited awareness capabilities. Existing CM tools usually require explicit manual action to break the isolation of workspaces, and even then information that is provided is old – not current and up-to-date directly from the workspace. Other than Palantír, few tools attempt to bridge CM and Computer-Supported Collaborative Work in raising awareness for users of CM systems. Jazz [16] is a collaborative tool built within the Eclipse IDE that shows the status of artifacts in other workspaces (artifact has been checked out and/or committed). Night watch [17] builds on the CVS watch facility, and notifies developers by email when an artifact in which they are interested has been changed by others. State Tree Maps [18] is an awareness widget that shows which artifacts have been modified locally or remotely and which artifacts have been checked-in into the repository. Compared to these tools, Palantír has two advantages: it provides a measure of severity, and it provides the full project view.

[1] We intend to build not just a measure of severity, but also a measure of change impact – one to measure "how large" a change is, the other to measure "how much I should care" about the change.

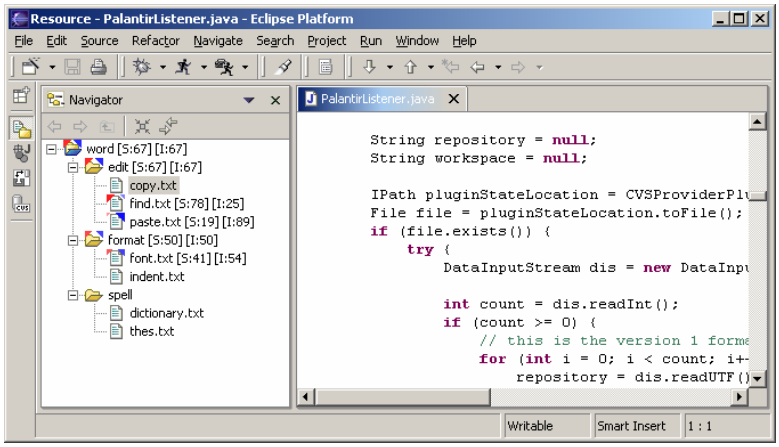


Figure 3. Eclipse Integration.

6. Conclusions

Palantír is based on the hypothesis that enhancing CM systems with awareness allows developers to have an improved insight into potentially conflicting parallel activities. The hope is that developers will use this insight to self-coordinate, in effect making earlier the point at which a conflict is detected and thereby reducing the amount of effort involved in addressing it.

We are currently finishing up the Eclipse integration and full project view. Our next steps in the project are to put Palantír in use, first in a class project in which we will have a large group of students all be part of a single team; and then in actual real-world settings as we release and deploy Palantír to the Open Source community itself.

7. Acknowledgements

We thank Ryan Yasui, Roger Ripley and Ksatria Williams for their contributions to Palantír.

Effort funded by the National Science Foundation under grant numbers CCR-0093489 and IIS-0205724.

8. References

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

[2] Erenkrantz, J.R. and R.N. Taylor. *Supporting Distributed and Decentralized Projects: Drawing Lessons from the Open Source Community*. in *Proceedings of 1st Workshop on Open Source in an Industrial Context*. Oct, 2003. Anaheim, California.

[3] Dourish, P. and V. Bellotti. *Awareness and Coordination in Shared Workspaces*. in *ACM Conference on Computer-Supported Cooperative Work*. 1992.

[4] Gutwin, C. and S. Greenberg. *Workspace Awareness for Groupware*. in *CHI'96 Conference Companion on Human Factors in Computing Systems*. 1996.

[5] Grinter, R.E., *Supporting Articulation Work Using Software Configuration Management Systems*. Computer Supported Cooperative Work, 1996. **5**(4): p. 447-465.

[6] Perry, D.E., H.P. Siy, and L.G. Votta, *Parallel Changes in Large-Scale Software Development: An Observational Case Study*. ACM Transactions on Software Engineering and Methodology, 2001. **10**(3): p. 308-337.

[7] Mens, T., *A State-of-the-Art Survey on Software Merging*. IEEE Transactions on Software Engineering, 2002. **28**(5): p. 449-462.

[8] IBM, *XML Diff and Merge Tool*. 2002.

[9] Sarma, A., Z. Noroozi, and A. van der Hoek. *Palantír: Raising Awareness among Configuration Management Workspaces*. in *Twentyfifth International Conference on software Engineering*. 2003. Portland, Oregon, USA.

[10] Tichy, W.F., *RCS, A System for Version Control*. Software - Practice and Experience, 1985. **15**(7): p. 637-654.

[11] Berliner, B. *CVS II: Parallelizing Software Development*. in *USENIX Winter 1990 Technical Conference*. 1990.

[12] Tigris.org, *Subversion*.

[13] Sarma, A. and A. van der Hoek. *Visualizing Parallel Workspace Activities*. in *IASTED International Conference on Software Engineering and Applications (SEA)*. 2003. Marina Del Rey, California.

[14] Carzaniga, A., D.S. Rosenblum, and A.L. Wolf, *Design and Evaluation of a Wide-Area Event Notification Service*. ACM Transactions on Computer Systems, 2001.

[15] Mockus, A. and J. Herbsleb. *Expertise Browser: A Quantitative Approach to Identifying Expertise*. in *2002 International Conference on Software Engineering*. 2002.

[16] Cheng, L.-T., et al. *Jazzing up Eclipse with Collaborative Tools*. in *18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications / Eclipse Technology Exchange Workshop*. 2003. Anaheim, CA.

[17] O'Reilly, C., P. Morrow, and D. Bustard. *Improving Conflict Detection in Optimistic Concurrency Control Models*. in *Proceedings of the Eleventh International Workshop on Software Configuration Management*. 2003. Portland, Oregon.

[18] Molli, P., H. Skaf-Molli, and C. Bouthier. *State Treemap: an Awareness Widget for Multi-Synchronous Groupware*. in *Seventh International Workshop on Groupware*. 2001.

Version Control Tools: A Collaborative Vehicle for Learning in F/OS

Maha Shaikh
Department of Information Systems
London School of Economics
m.i.shaikh@lse.ac.uk

Tony Cornford
Department of Information Systems
London School of Economics
t.cornford@lse.ac.uk

Abstract

In this paper we explore how version control software participates in learning within free/open source activities (F/OS). We see F/OS in terms of a product, and a community of people engaged in the process of its development, with version control software at the centre of all three activities as they learn and innovate. Learning is analysed through the perspective provided by Bateson's Levels of Learning, a relational model that stresses collaboration and conflict as drivers of learning and showing how conflict resolution may lead to higher and more profound or significant learning.

1. Introduction

It is usually taken for granted that version control software [VCS] is an indispensable part of free/open source [F/OS] development activity. Software patches are accumulated and exchanged via such tools, working systems are distributed, and developers manage some of their communication through them. However, VCS does more than just support development activity at the level of changing chunks of code. Here we ask what role VCS plays in learning in F/OS? F/OS can be seen as a combination of three elements: a *product* - software that is jointly worked on by developers and is available to a wider group of users; a *community* of developers centred around the product; and a *process* that they engage in

F/OS software *products* have certain characteristics that can be learned about or innovated, such as their architecture, criteria of quality, coherence and accessibility. Given Linus' Law, there is a particular need for transparency to allow review. *Community* refers to the norms of the developers, how they enter the community, their motivations for participation, the ideology that binds them together, and the levels of trust required to be self managing. Qualities that make a F/OS *process* appropriate include a high degree of responsiveness (release early, release often), to be inclusive, reliable and coherent (understood). Such distinctions are not absolute, but for the sake of clarity we use them here to identify various opportunities for learning that may be mediated to some degree through VCS.

The structure of the paper is as follows; in the next section we briefly introduce version control software. This is followed by an explanation of Bateson's learning ideas and a quick sketch of our case study and finally onto the analysis. The diagonal line in Table 1 indicates the path of analysis we take.

Table 1. Role for version tools in F/OS learning

Bateson's Levels of Learning	Role of Version Control Tools		
	Product	Community	Process
Learning I			
Learning II			
Learning III			

2. Version control software

Version control softwares are used by software developer communities in both proprietary and free/open source environments. Usually designated as a 'tool' they are conventionally defined as '*a mechanism for managing the multiple versions of the software objects that are created during the software development process* [4] and emphasise control, '*keeping track of the configuration items which are any documents created during a software development process, and which are found necessary to be placed under configuration control like requirements documents, data flow diagrams, design documents, source code, and test results* [7]. In contrast to a simple tool or control perspective, our view is to see VCS as an

actor within a heterogeneous network of interests which it inscribes and translates.

Such software has long been a key part of open source activity with a history going back at least to the origins of UNIX [13]. F/OS communities, by their very nature, require them to manage software revision and release control within a multi-developer, multi-directory, multi-group environment [3, 6]. Tools such as RCS [Revision Control System] and SCCS [Source Code Control System] have long been in use [8], but Concurrent Version System [CVS], an OS product, is now recognized to be the most popular tool in F/OS communities [17].

3. Levels of learning

To understand how learning occurs and is manifested in F/OS we have adopted Bateson's concepts of levels of learning as our framework [2]. Argyris and Schon [1] base their organizational learning work on Bateson's ideas, adapting it to help understand collective or organizational learning. Bateson suggests that "*what can be studied is always a relationship or an infinite regress of relationships. Never a 'thing'*". This takes us away from learning as something that can be isolated but rather, learning occurs through relationships and collaborations [14]. VCS mediate such relationship building and it is central to this paper to see them as actors that are engaged within a learning process, promoting and reflecting learning. For Bateson learning is about some sort of '*change*', a *communication of ideas* and a *mastery of a new approach or solution* [11]. His view of learning is adaptive and ecological; 'learning is adaptation, and evolution is its highest form' [11], and emphasizes learning as a product of collaboration that is inherently social and relational. But any relational process involving collaboration also invites paradoxes and conflict. Such paradox and conflicts, and how they are understood or resolved, are proposed as the key to understanding how an individual or organization learns.

The version control software considered here has often been at the centre of conflict. For example, conflicts over patch submission may be resolved partly by VCS itself. It is, after all, named as a *control* device, and it does, through its process of managing software items, exercise control over development. The choice of VCS itself has also lead to explicit conflict, revealing the dynamic within F/OS communities and their conception of the process they engage in [13].

Bateson's presents learning as occurring at four different levels. At the lowest is Zero learning, which is hardly any learning at all. This involves simple reflex actions, or stimulus-response, which are needed to maintain a local status quo.

Learning I is the ability to learn to respond to diverse stimuli within the same or similar context – learning something. Learning II is a reflexive activity that allows activity to move across contexts and resolve paradox – learning about learning. Change of context and paradox resolution differentiates Learning I from Learning II; an ability to adapt responses and to consciously reflect on learning techniques and habits and improve them – often described as learning to learn. Reflection may be triggered by some crisis, breakdown or paradox that forces consideration of behaviour. This can lead to either reflection on learning habits in order to improve skills, or cause confusion and result in a state of limited reaction.

Learning III is understood as rarer and entails a 'profound reorganization of character' as for example in religious conversions. Learning III is a response to significant challenges to established theory or paradigms. It is an intriguing feature of open source communities that this form of learning lies at its core, and the Linux kernel community, for example, has over the last 10 years undergone a series of profound reorganizations when their learning paradigm has adapted.

4. Linux kernel case study

The F/OS case used here is the Linux kernel community. This community has a long history of concerns with VCS adoption and use [13]. They began with pre-patch releases and then part of the community adopted CVS. This VCS was not espoused by Linus Torvalds who, after some time, chose a closed source tool BitKeeper [BK]. This proved to be the catalyst of much distrust and dissent. It also lead to an attempt at a 'GPL'd clone' called BitBucket [9]. Torvalds backed BitKeeper and uses it to this day. His reasons for doing so, and the problems this caused, are discussed in Shaikh and Cornford [13].

The creator of BitKeeper, Larry McVoy, resented the BitBucket project and threatened to sue. At around this time he made 'amendments' to the BitKeeper license [BKL] making it impossible for any developer who had previously worked on 'competing' VCS to use BK. This threatened to become a serious dispute with the level of distrust rising.

Another issue that arose was the incompatibility between CVS and BK. Those developers who could not use BK, or did not wish too because of a conflict in ideology, were unable to access the main tree of the Linux kernel kept by Torvalds. McVoy took the opportunity to appease kernel developers by creating a gateway between BK and CVS. The situation seemed to get better only to end in further suspicion. BK's conversion of data into CVS did not add up completely to what was held by BK. What was available to the CVS users was not *all* the metadata and nothing McVoy said about the insignificance of the difference altered some kernel developers' resentment or mistrust.

5. Analysis

The brief history given in the section above will help the reader to understand the significance of VCS for F/OS communities. The case suggests immediately how such software mediates aspects of the community's governance structure, signifies access to code and metadata, and inscribes procedures for patch submission. Table 2, provides our framework for analysis of how VCS engage with activities of learning within this community and indicates areas of learning we identify within our study of the VCS debate within LKML. We only explore some of the framework here, taking a path to consider learning I in *product*, learning II in *community*, and learning level III in *process*.

5.1. Learning I in Product

VCS stores source code. In the case of BK, code is held as changesets, which are groups of deltas, accessible to developers to read and reuse. Both BK and CVS provide developers with an opportunity to read and compare code written by others and thus 'learn something'. A requirement for comparison may arise when more than one patch is sent to solve the same bug. In the case of BK the second developer is made aware of the duplication of effort and is then asked to determine which of the two patches should be accepted by looking at both patches side-by-side[5]. At a simple level, this comparison affords developers a window to observe *how* others would attack the same bug, but also allow the review of other code-use in other contexts. VCS also offers developers the chance to read metadata written by developers to explain their lines of code [12]. This, like the openness of source code, grants developers a chance to compare and learn.

5.2. Learning II in Community

A slightly different scale of comparison is offered by the ability to backtrack through deltas to see *how* change has occurred and what direction it took. This can support learning within the community through emulation and adapting of ideas. It is at the point of adaptation of ideas that this level of learning penetrates into the next, higher level of Learning II. It also moves from simple learning about code and coding to learning about aspects of participation within the community.

A community in F/OS clings to and seeks to sustain some ideology, has certain motivations, and relies on a culture built on trust and peer-review. Learning II indicates ability within such a community to adapt behaviour when the context changes and as a result of reflection. This level of learning is characterized by breakdowns, paradox situations or challenges that need to be addressed to regain some form of order.

As the brief history given above shows, VCS is seen to exercise control over the community and (in the case of BK) is structured to meet the needs of the governing body. One of Torvalds' main reasons for embracing BK was to control who had access to 'his' tree. He didn't want all developers to have access or the ability to load their patches without going through him first.

The community conflict over BK vs CVS, the gateway and the metadata, challenged many people to reflect on their current ways of working, the values they espoused, the norms of participation and development contribution. The context had changed and old behaviours or beliefs would not work. Such reflection demanded some new responses – for example, to GPL BK, to work through a gateway, or to engage in more debate and dissent.

5.3. Learning III in Process

In conventional software engineering terms VCS are all about process (control, tools), while in F/OS terms the process is itself a radical reorientation of software engineering. Here we want to analyze the role version control tools play in challenging fundamental assumptions and paradigms, and how learning at level III is manifested.

There is more than one instance of such a challenge and response in the Linux kernel case, for example in discussion of the purity of the GPL, ownership of codes, rules of participation and forking. Rare as this level of learning may be in individuals, it seems to be a recurrent theme in F/OS communities as they debate their existence and fundamental values. Here we focus on Torvalds decision to adopt BK [16] rather than CVS, remembering that in 1995 Torvalds had decided clearly that he would not use CVS for kernel development, though he was happy to use the tool for his work at Transmeta [15]. If this had been a move to a F/OS tools it still would have caused a stir, but the idea of using a closed source tool to create *the* kernel that stands for F/OS for many was an outrage. It divided developers between BK and CVS users and served as a catalyst for serious dissension and even a threat of a fork.

Using a closed source product to create their open source kernel threw the GPL into question and the sanctity of the code they produced. Faced with two equally distressing choices [or double-bind] of either forking the kernel or adopting a closed source tool gave rise to two responses. The community adapted in the form to create a GPL'ed clone of BK [BitBucket] so that they wouldn't have to use BK, and it also reinforced a sense of community and voice sufficient to persuade McVoy to create the BK→CVS gateway [10] which allowed CVS

users indirect access to the source code and (some) metadata held in the BK repository.

6. Conclusion

Our contention is that F/OS activity is based on learning as a response to changing environmental factors and on the basis of experience. We suggest that such learning activity can be identified, for analytical purposes, within three main elements, the code itself, the community that surrounds it, and the process they engage in. Each element has to be learned about, each element poses problems and paradoxes that require some reflection on learning style or modes. At times, and more often perhaps than we thing, learning in F/OS is focused around a questioning of certain fundamental assumptions or core values, as in issues of license discussed here. In all this we see version control software playing a role within the heterogeneous network of interests. It is both a medium of learning (e.g. code visibility), but also a product of learning as it embodies or inscribes certain interests, while attenuating others. We see the community and the 'tool' mutually constructing one another as a learning activity

7. References

[1]C. Argyris and D. Schön, *Organization learning: A theory of action perspective*, Reading, Mass: Addison-Wesley, 1978.

[2]G. Bateson, *Steps to an ecology of mind*, New York: Chandler, 1972.

[3]B. Berliner, "CVS II: Parallelizing Software Development," in *Proceedings of Proceedings of the USENIX Winter 1990 Technical Conference*, Washington D.C., 1990.

[4]G. Clemm, "Replacing Version-Control with Job-Control," in *Proceedings of Proceedings of the 2nd International Workshop on Software configuration management*, Princeton, New Jersey, United States, 1989, pp. 162-169.

[5]V. Henson and J. Garzik, "BitKeeper for Kernel Developers," in *Proceedings of Ottawa Linux Symposium*, Ottawa, Ontario Canada, 2002, pp. 197-212.

[6]J.D. Herbsleb and R.E. Grinter, "Splitting the organization and integrating the code: Conway's law revisited.," in *Proceedings of 21st International Conference on Software Engineering (ICSE 99)*, Los Angeles, 1999.

[7]T. Kilpi, "New Challenges for Version Control and Configuration Management: a Framework and Evaluation," *IEEE Computer*, no. 1st Euromicro Working Conference on Software Maintenance and Reengineering (CSMR '97), 1997, pp. 33-41.

[8]H. Koike and H.-C. Chu, "Integrating Version Control and Module Management Using Three-Dimensional Visualization.," in *Proceedings of HCI*, San Francisco, California, USA, 1997, pp. 853-856.

[9]P. Machek, BitBucket: GPL-ed BitKeeper clone, University of Indiana, 2003. http://www.uwsg.indiana.edu/hypermail/linux/kernel/0302.3/0931.html

[10]L. McVoy, kernel.bkbits.net and BK->CVS gateway, University of Indiana, 2003. http://www.ussg.iu.edu/hypermail/linux/kernel/0305.1/0632.html

[11]W.J. Rothwell, "Bateson's Heterarchy of Learning," *Training and Development Journal*, vol. July, 1983, pp. 24-27.

[12]M. Shaikh and T. Cornford, "Version Control Software for Knowledge Sharing, Innovation and Learning in OS," in *Proceedings of Open Source Software Movements and Communities Workshop hosted by the International Conference on Communities and Technologies*, Amsterdam, The Netherlands, 2003, pp. 39-46.

[13]M. Shaikh and T. Cornford, "Version Management Tools: CVS to BK in the Linux Kernel," in *Proceedings of 25th International Conference on Software Engineering - Taking Stock of the Bazaar: The 3rd Workshop on Open Source Software Engineering*, Portland, Oregon, 2003, pp. 127-132.

[14]S. Star and K. Ruhleder, "Steps toward an Ecology of Infrastructure: Design, Access for Large Information Space," *Information System Research*, vol. 7, no. 1, 1996, pp. 111-134.

[15]L. Torvalds, Re: CVS, Linus, and us, University of Indiana, 1995. http://www.uwsg.indiana.edu/hypermail/linux/kernel/9602/0800.html

[16]L. Torvalds, linux-2.5.4-pre1 - bitkeeper testing, University of Indiana, 2002. http://www.uwsg.indiana.edu/hypermail/linux/kernel/0202.0/0989.html

[17]A. van der Hoek, "Configuration Management and Open Source Projects," in *Proceedings of Proceedings of the 3rd International Workshop on Software Engineering over the Internet*, Limerick, Ireland, 2000.

Table 2. Framework to understand the role played by version control tools in F/OS learning

Bateson's Levels of Learning	Roles for Version Control Software			
	Issues where learning may centre.	*Architecture* *Code quality* *Coding conventions* *Security*	*Membership* *Motivation* *Ideology* *Reputation and trust* *Norms*	*Responsive* *Inclusive* *Reliable* *Coherent*
		Product	**Community**	**Process**
	Learning I *Learning 1 is an ability to understand and repeat behaviour in similar contexts.*	▪ Allow developers to compare and read the code ▪ Allows simple reading of metadata	▪ Informs the community about *how* to 'talk' to each other and communicate ▪ A vehicle for learning netiquette	▪ How to submit a patch or read code and deltas ▪ Defining software activities ▪ Defining software roles
	Learning II *Learning 2 is the ability to adapt behaviour when the context is changed and is a result of reflection and paradox resolution.*	▪ Backtracking through deltas to see *how* change occurred ▪ Comparing patches to improve technique ▪ Insights into how 'elegant code' can be written	▪ Changing norms of communication. ▪ Use of peer review ▪ Conflict resolution activities.	▪ Inscribing the management of the software process ▪ Adding audit trails and accountability
	Learning III *Level 3 is a more profound ability to challenge and change assumptions.*	▪ Visibility of code and architecture allow technical challenges ▪ User demands meet established practices	▪ Inscribing new community norms of governance and hierarchy ▪ Addressing questions of license which challenge openness and ownership of code	▪ Challenges to issues of openness and access to code and metadata ▪ Questioning participation ▪ Demanding new processes (gateway)

OSS Project Success: From Internal Dynamics to External Impact[1]

Katherine J. Stewart
R. H. Smith School of Business
University of Maryland
kstewart@rhsmith.umd.edu

Abstract

Success in open source software (OSS) development has been viewed in many different ways. This paper examines the relationship between success in terms of the development efforts devoted to an OSS project and success in terms of the acceptance of the software by stakeholders beyond the immediate development group. Research has discussed the importance of communication, norms, and other factors related to the dynamics of work within open source software development groups. The model proposed here suggests a positioning of these internal dynamics in a larger nomological net explaining variation in the success of OSS efforts.

1. Introduction

Research has assessed the output of individual OSS projects and examined how dynamics within OSS projects influence the development of the software product (e.g., [21, 25]). In addition, several studies have taken a higher level view and sought explanations for the phenomenon of OSS and an understanding of the wider OSS community (e.g., [17, 18]). Building on this work, this paper suggests positioning what has been learned about OSS development in a wider framework targeted at understanding the impact of OSS on users. A main contention of the paper is that as OSS diffuses across a broader set of users and types of uses, it becomes important to understand how the individual characteristics and internal dynamics of OSS projects influence the eventual impact that the software produced in those projects has on users.

The paper first suggests a multifaceted understanding of success in OSS from both a development perspective (i.e., success in terms of attracting input and producing output at the project level) and a usage perspective (i.e., success in terms of user interest, adoption, and impact related to specific OSS projects). Section three then summarizes the proposed theoretical model and the high level propositions it encompasses. The final section discusses further development of the model, limitations, and implications.

2. Conceptualizing OSS Success

OSS development has been conceptualized as a phenomenon at the community [1], organizational [19] and team/group level [21]. While recognizing the utility of these different levels of analysis, the focus of this paper is limited to discussing OSS success at the project level. Thus dimensions of and antecedents to success discussed below are conceptualized at the level of the team working on a project, the project itself, or the users of a specific project.

Some success indicators applied to commercial software projects – e.g., being on-time, on-budget, and meeting specifications – may not be readily applied in the OSS setting. In this setting, there may be no a priori budget, timeline, or set of specifications [25]. Other evaluation criteria applied to projects in the IS and software engineering literatures are more applicable. These include system quality, information quality, system usage, user satisfaction, and system impact on users and organizations [5].

In addition to conceptualizations of success that may be applied from closely related settings, there are some kinds of success uniquely relevant in OSS development. Because OSS often depends on volunteer labor, the extent to which a project attracts, retains, and motivates developers is an important aspect of success [21]. This kind of success may be indicated by such factors as the number of developers involved, the level of project activity (e.g., bug fixes, patches provided, new features and software releases), or project development status (e.g., alpha testing, beta testing, production, etc) [3, 27].

The model depicted in figure 1 includes two broad types of OSS success, labeled *development success* and *usage success*. Development success is proposed to

[1] This paper is based on research supported by the National Science Foundation award IIS-0347376.

consist of success in attracting input to the project and success in producing project output. The former might be indicated, for example, by the number of contributions made to the code repository while the latter might be indicated by an assessment of the quality of the software release. Usage success is composed of user interest, which might be indicated by a user downloading project files, and also user impact, which might be indicated by performance changes that result from use of the software.

3. Antecedents to OSS Success

Research suggests many potential factors that may influence both development and usage success. For the purpose of this paper these factors are grouped into three categories: those associated with the development team (Team Factors), those associated with the product of the team (Project Factors), and those associated with users of the product (User Factors). The remainder of this paper presents propositions about the main mechanisms by which these sets of factors may influence OSS success both in terms of development and usage.

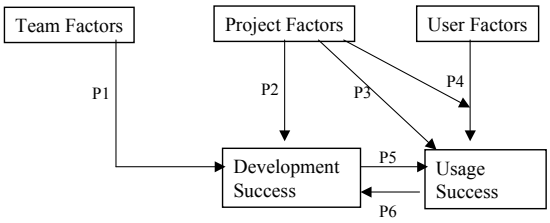


Figure 1: A general framework of OSS success

3.1 Impact of Team Factors

In many ways OSS teams are like the virtual teams studied in the IS literature. For example, they generally use the Internet as their main means of communication, and they may have widely distributed members. Among the team factors argued to be important in prior work, trust, communication, and culture have been suggested to play especially important roles in OSS [18, 28]. Because they have been the focus of prior work, these three team factors are discussed as exemplars below.

Trust. Ljungberg ([18]:215) states that "(Open source) could be seen as a virtual organization, but where trust has replaced law in regulating relations." This view is consistent with much of the management and virtual teams literature, which has argued that trust is closely linked to performance outcomes (e.g., [15]). When affective trust exists, sensitivity to the personal and work-related needs of other group members is high [20], leading to higher levels of responsiveness in helping other team members. Affective trust has been shown to have a positive impact on organizational citizenship behavior among managers and peers [20]. Further, affective trust is persistent, leading to the discounting of events that are contrary to ongoing harmony [14]. In the OSS context, for example, a contributor whose code is not accepted into a new release in one instance may feel rejected and choose to leave the group by ceasing to contribute. However, if affective trust is high, negative events such as this are more likely to be attributed to situational factors and not result in a reduction of commitment to the project.

Communication. Studies of commercial software design efforts have found that high performing virtual teams significantly "outcommunicated" low performing virtual teams [22]. Previous work on communication in virtual teams implies at least two distinct types of communication may be important: social communication, in which team members discuss things other than the project (e.g., their hobbies, families, and weekend plans) and task relevant communication, which is characterized by the extent to which the communication is perceived to be useful (e.g., timely and helpful with regard to the project) [15, 36]. Social communication enhances identification with the team and commitment to the team while task communication is required to complete and coordinate work. Thus social communication may influence retention of developers and therefore impact input while task communication facilitates the coordination and combination of input to produce high quality outputs.

Culture. Of the many antecedents to success suggested in the virtual teams literature, culture has received the widest acknowledgement as a factor especially important in OSS [1, 2, 19]. Ideology, a key component of culture, is defined as "shared, relatively coherently interrelated sets of emotionally charged beliefs, values, and norms that bind some people together and help them make sense of their worlds" [30]. Beliefs refer to understandings of causal relationships, values refer to preferences for some behaviors or outcomes over others, and norms refer to behavioral expectations.

Though culture is by definition shared, it may vary across subgroups [12, 13]. Open source developers have often been said to constitute a community, identifiable by its common culture [1]. At the same time, subgroups are formed around projects within that community, and such subgroups may vary in the extent to which they conform to the overarching community ideology. Indeed, Ljungberg ([18]: 210) suggests that ideology varies widely across OSS developers: "At one end of the spectrum there is great zeal…at the other end there is no big deal about the ideology."

In general, the existence of a shared ideology in a workgroup is thought to facilitate group efforts. Shared values can lead to better mobilization of group members

towards directed ends by connecting group goals to moral imperatives, and personal commitment may be created by increasing the salience of collective identities and values and linking behaviors to these identities and values [26]. Ideological commitment serves as a particularly effective control in distributed contexts where close supervision is not feasible and every type of contingency may not be easily accounted for and codified in contractual terms [23].

The OSS ideology in particular seems designed to facilitate project work. Main components of the OSS ideology as including values related to sharing and helping others, learning through participation, and protecting group member contributions [28]. Thus the extent to which project members subscribe to the common open source ideology may have positive effects on development success by increasing group commitment and acting as a control mechanism.

Summary. The brief discussion above is meant to show that team factors discussed in prior work – trust, communication, culture – may all work through similar mechanisms to influence the success of OSS development efforts. The proposition below is intended to imply that other team factors – for example, leadership style or group member heterogeneity – might be fruitfully examined by considering their impact on these same mechanisms.

Proposition 1: Team factors that enhance developer identification with and commitment to a project attract and retain more developer input and facilitate the successful conversion of that input into output, leading to greater overall development success.

3.2 Impact of Project Factors

Contrary to some popular conceptions of OSS development as drawing from an infinite pool of talent (e.g., [31]), OSS work requires specific skills and there is a limited pool of people with the knowledge and motivation to be able to productively contribute, leading to potential competition among projects to attract developer efforts. For example, [16] cites the Orbiten Free Software Survey [8], which indicates that the 100 most prolific OSS contributors contribute to 1,886 distinct projects, a contributor to project ratio of approximately 1 to 19. Other analyses [6, 17] also show that a relative few individuals make most contributions, while most contributors make only a single contribution. Thus given the voluntary nature of most OSS participation, development success may depend in large part on the extent to which productive developers are motivated to contribute to one project versus another.

With regard to motivation, researchers have suggested that OSS contributors find programming intrinsically motivating, deriving feelings of competence and self-determination from the activity itself or from helping others [3, 11]; that they contribute to satisfy their personal needs for software [11, 24, 34]; and that programmers contribute to enhance their skills and reputations [16], possibly with the expectation of future returns [4, 11, 17]. Future returns might be derived because participation in OSS both allows contributors to increase their human capital (e.g. by honing their skills), and it allows them to advertise their skills thereby enhancing their reputations and their prospects for paid development work [10, 11, 17]. Given that projects compete for members and that participation in a project is largely a function of a developer's motivation, the model suggests that a major influence on development success will be the relationship between project characteristics and developer motivations. Potential developers will judge the opportunities for skill development, potential utility, and possible reputational benefits based on characteristics of the project such as the license used, the problem domain addressed, or the programming language used.

Proposition 2: The greater the alignment between project factors and OSS developer motivations the greater will be the development success of the project.

Project factors may also be important in determining usage success, as characteristics such as license may be readily observed by potential users. The IS literature on technology acceptance and usage provides a foundation for considering antecedents to OSS user interest and impact. This literature has overwhelmingly supported the idea that perceptions of utilitarian value (perceived usefulness) and perceptions of required effort (perceived ease of use) are key determinants of behavioral intention to use and actual use of technology [29, 33].

Research in the marketing literature has demonstrated that perceptions of value, which in this setting would result from usefulness and ease of use, may be developed from individuals' use of extrinsic cues in the formation of product quality and cost perceptions [7]. Using affect-referral, customers simplify their decision making process by basing their judgments on summary information (e.g., brand attitudes) rather than on product attribute information [35, 37]. In the OSS context, although code is available for inspection, users may not have the necessary background knowledge to evaluate the inner workings and features of a software program before they install it, or even if they do have the requisite skill, they may seek to minimize the cognitive effort involved in evaluation by relying on more easily interpreted cues. Project characteristics such as organizational affiliation and license may be viewed as salient extrinsic cues for evaluating the usefulness and ease of use of OSS projects.

Proposition 3: Project characteristics that are perceived to indicate high levels of usefulness and ease of use of the software will have a positive impact on user interest and adoption.

3.3 Impact of User Factors

The least explored areas of OSS success included in the model are user impact and the role of user factors. When contemplating adoption of a new software package, users must consider many issues including integration with current technology infrastructure and future requirements for maintenance. Thus users may vary in their ability to use and benefit from OSS based on factors such as their technical expertise. However, the importance of such factors will depend heavily on the nature of the software being considered. While integration with current technology may be crucial for the organization user contemplating adoption of an OSS software development tool, it may be irrelevant for the individual user considering downloading a new OSS game. Thus the proposition represented in the research model is derived from contingency theory [9, 32], which suggests that outcomes depend on the fit, or lack of fit, among organizational and environmental factors. The impact (positive or negative) that OSS adoption may have on users will depend on the extent to which characteristics of the project "fit" with characteristics of the user.

Proposition 4: The extent to which project characteristics are aligned with user factors will have a positive impact on usage success by both increasing user interest and increasing the likelihood of positive impact from adoption.

3.4 Relationship between Development and Usage Success

Above, it has been assumed that in order for a project to be successful in creating output, it must first attract development input. Similarly, project output may be expected to affect user interest such that more active projects are also more popular, based on the idea that the quantity and quality of output are critical to perceived usefulness. The model also suggests a feedback effect from usage success to development success that is derived from a consideration of the motivations of OSS developers. The greater the user interest in a project, the wider the audience for individual contributions and therefore the more visible the efforts of contributors. Hence there may be greater potential reputation benefits from working on more popular projects [17], and we might expect such projects to attract more activity from developers. Further, an active user base will generate defect reports and support requests [21], providing greater opportunities for developers to hone their skills on a variety of tasks and thereby stimulating more development input.

Proposition 5: OSS development success will increase the perceived usefulness of an OSS product and thereby have a positive effect on usage success.

Proposition 6: OSS usage success will enhance the motivation of developers to contribute to a project and thereby have a positive effect on development success.

4. Conclusion

This paper has presented a model of how developer, project, and user factors may influence the success of OSS projects both in terms of attracting input and generating output (development success) and also in terms of attracting user interest and providing benefits to users (usage success). As depicted in figure 1, the research reviewed above indicates three important categories of influences on OSS success, labeled team factors, project factors, and user factors. Propositions outline the mechanisms by which these factors may influence OSS success. The model has several limitations, for example there may be other categories of factors that are omitted but have important impacts on success (e.g., environmental factors such as the current economic climate in an industry could impact the success of OSS products targeted toward that industry). Such limitations aside, the model is presented in the hope that it may encourage researchers to more deeply examine questions related to the impact that OSS may have on the user community and how the internal dynamics in projects, here encompassed in team factors, may have effects on the ultimate external acceptance and impact of the software.

6. References

[1] Bergquist, M. and J. Ljungberg, "The Power of Gifts: Organizing Social Relationships in Open Source Communities," *Information Systems Journal*, 2001, 11(4): p. 305-320.

[2] Bretthauer, D., "Open Source Software: A History," *Information Technology and Libraries*, 2002, 21(1): p. 3-11.

[3] Crowston, K. and B. Scozzi, "Open Source Software Projects as Virtual Organizations: Competency Rallying for Software Development," *IEE Proceedings Software*, 2002, 149(1): p. 3-17.

[4] Dalle, J.-M. and N. Jullien, ""Libre" Software: Turning Fads into Institutions?," *working paper*, 2001, http://opensource.mit.edu

[5] Delone, W.H. and E.R. Mclean, "Information Systems Success: The Quest for the Dependent Variable," *Information Systems Research*, 1992, 3(1): p. 60-94.

[6] Dempsey, B.J., et al., "Who Is an Open Source Software Developer?," *Communication of the ACM*, 2002, 45(2): p. 67-72.
[7] Dodds, W.B., K.B. Monroe, and D. Grewal, "Effects of Price, Brand, and Store Information on Buyers' Product Evaluation," *Journal of Marketing Research*, 1991, 28: p. 307-319.
[8] Ghosh, R.A. and V.V. Prakash, "The Orbiten Free Software Survey: May 2000," 2000, http://orbiten.org/ofss/01.html
[9] Gresov, C., "Exploring Fit and Misfit with Multiple Contingencies," *Administrative Science Quarterly*, 1989, 34: p. 431-453.
[10] Hann, I.-H., et al. "Economic Incentives for Participating in Open Source Software Projects," in the proceedings of the *23rd International Conference on Information Systems*, 2002, Barcelona, Spain.
[11] Hars, A. and S. Ou, "Working for Free? Motivations for Participating in Open Source Projects," *International Journal of Electronic Commerce*, 2002, 6(3): p. 25-39.
[12] Hofstede, G., *Culture's Consequences*. 1980, Beverly Hills, CA: Sage.
[13] Hofstede, G., et al., "Measuring Organizational Cultures: A Qualitative and Quantitative Study across Twenty Cases," *Administrative Science Quarterly*, 1990, 35(2): p. 286-316.
[14] Holmes, J.G. and J.K. Rempel, "Trust in Close Relationships," in *Close Relationships*, C. Henrick, Editor. 1989, Sage: Newbury Park, CA. p. 187-202.
[15] Jarvenpaa, S. and D. Leidner, "Communication and Trust in Global Virtual Teams," *Organization Science*, 1999, 10(6): p. 791-815.
[16] Krishnamurthy, S., "Cave or Community? An Empirical Examination of 100 Mature Open Source Projects," *working paper*, 2002, http://opensource.mit.edu
[17] Lerner, J. and J. Tirole, "Some Simple Economics of Open Source," *Journal of Industrial Economics*, 2002, 50(2): p. 197-234.
[18] Ljungberg, J., "Open Source Movements as a Model for Organizing," *European Journal of Information Systems*, 2000, 9(4): p. 208-216.
[19] Markus, M.L., B. Manville, and C.E. Agres, "What Makes a Virtual Organization Work?," *Sloan Management Review*, 2000: p. 13-26.
[20] Mcallister, D.J., "Affect- and Cognition-Based Trust as Foundations for Interpersonal Cooperation in Organizations," *Academy of Management Journal*, 1995, 38(1): p. 24-59.
[21] Mockus, A., R.T. Fielding, and J.D. Herbsleb, "Two Case Studies of Open Source Software Development: Apache and Mozilla," *ACM Transactions on Software Engineering and Methodology*, 2002, 11(3): p. 309-346.
[22] Ocker, R.J. and J. Fjermestad. "High Versus Low Performing Virtual Design Teams: A Preliminary Analysis of Communication," in the proceedings of the *33rd Hawaii International Conference on System Sciences*, 2000, Hawaii.
[23] O'leary, M., W. Orlikowski, and J. Yates, "Distributed Work over the Centuries: Trust and Control in the Hudson's Bay Company," in *Distributed Work*, P. Hinds and S. Kiesler, Editors. 2002, MIT Press: Cambridge, MA.
[24] Raymond, E.S., *The Cathedral and the Bazaar: Musing on Linux and Open Source by an Accidental Revolutionary*. 2001, Sabastopol, CA: O'Reilly.
[25] Scacchi, W., "Understanding the Requirements for Developing Open Source Software Systems," *IEE Proceedings on Software*, 2002, 149(1): p. 24-39.
[26] Shamir, B., R.J. House, and M.B. Arthur, "The Motivational Effects of Charismatic Leadership: A Self-Concept Based Theory," *Organization Science*, 1993, 4(4): p. 577-594.
[27] Stewart, K.J. and T. Ammeter. "An Exploratory Study of Factors Influencing the Level of Vitality and Popularity of Open Source Projects," in the proceedings of the *23rd International Conference on Information Systems*, 2002, Barcelona, Spain.
[28] Stewart, K.J. and S. Gosain. "An Exploratory Study of Ideology and Trust in Open Source Development Groups," in the proceedings of the *22nd International Conference on Information Systems*, 2001, New Orleans, LA.
[29] Taylor, S. and P.A. Todd, "Understanding Information Technology Usage: A Test of Competing Models," *Information Systems Research*, 1995, 6(2): p. 144-176.
[30] Trice, H.M. and J.M. Beyer, *The Cultures of Work Organizations*. 1993, Englewood Cliffs, NJ: Prentice Hall.
[31] Valloppillil, V. and J. Cohen, "The Halloween Documents," 2000(4/25/01), http://www.opensource.org/halloween/halloween1.html
[32] Van De Ven, A.H. and R. Drazin, "The Concept of Fit in Contingency Theory," in *Research in Organizational Behavior*, B.M. Staw and L.L. Cummings, Editors. 1985, JAI Press.
[33] Venkatesh, V., "Determinants of Perceived Ease of Use: Integrating Control, Intrinsic Motivation, and Emotion into the Technology Acceptance Model," *Information Systems Research*, 2000, 11(4): p. 342-365.
[34] Von Hipple, E., "Learning from Open Source Software," *Sloan Management Review*, 2001, Summer: p. 82-86.
[35] Wright, P.L., "Consumer Choice Strategies: Simplifying Vs. Optimizing," *Journal of Marketing Research*, 1975, 11(60-67).
[36] Zack, M., "Social Context and Interaction in Ongoing Computer Supported Management Groups," *Organization Science*, 1995, 6(4): p. 394-422.
[37] Zeithaml, V.A., "Consumer Perceptions of Price, Quality, and Value: A Means-End Model and Synthesis of Evidence," *Journal of Marketing*, 1988, 52: p. 2-22.

Exploiting the Collaboration between Open Source Developers and Research

Giancarlo Succi, Paolo Zuliani
Centre for Applied Software Engineering, Free University of Bolzano-Bozen
piazza Domenicani 3, 39100 Bolzano-Bozen, Italy
{giancarlo.succi, paolo.zuliani}@unibz.it

Abstract

In this paper it is argued that open source developers and research projects carrying out user requirement analysis should collaborate at a closer level. In particular, we discuss how developers could take advantage of the knowledge generated by COSPA, a research project aimed at studying and supporting the introduction of open source software in the Public Administration. COSPA focuses on office automation and desktop system software and it could thus provide developers with user requirements from one of the main "corporate" users of software. To this aim, the project has established an "observer" status, by means of which interested parties may access COSPA's results, thereby fostering collaboration and increasing dissemination of knowledge.

1. Introduction

Open Source Software (OSS) has grown a lot in popularity. Linux and the Apache web server are found in respectively 30% and 66% of the Internet's public servers, according to Netcraft's survey [1]. We thus have some empirical evidence that OSS can work well, at least for the server side of the client-server architecture. On the other had, it seems that OSS is not well suited for desktop and client applications, for which we know that Microsoft Office is the *de-facto* standard. The FLOSS study [3] showed that only 8% of business and public institutions use some kind of OS desktop software. If we consider office automation tools only, the percentage drops to a mere 4%.

Why such a difference, with respect to system software? Some authors suggest that it might be due to the fact that developers of OS system software "*knew what they were doing and how to do it*" [6]. In the case of Linux, the group of developers headed by Torvalds had a clear idea of the *requirements* of the system being developed, plus they all shared knowledge of the Unix structure [7]. Indeed, research in software engineering has clearly proved the importance and impact of requirements in software development [8]. In the case of OSS, requirements elicitation and sharing is an even more critical activity, as development teams are usually geographically distributed.

Another reason for the difference in popularity between desktop and system OSS is the fact that the former is actually much younger, and therefore it has not benefited from the same amount of user testing and feedback as OS system software has had. However, it looks nonetheless very important that desktop OSS exploits and incorporates user requirements as soon as possible, in order to develop products grounded on users' expectations. At the moment it is not clear how user needs are taken into account by OS developers.

In this paper it is argued that open source developers and research projects carrying out user requirement analysis should collaborate at a closer level. In particular, we make the case for a close interaction between OSS desktop developers and the COSPA research project. COSPA studies the introduction of OS desktop software in the Public Administration and could therefore provide the OS developers community with valuable information about the requirements of a "corporate" sector.

2. The COSPA project

The Consortium aims at introducing, analysing, and supporting the use of Open Data Standards (ODS) and Open Source (OS) software for personal productivity and document management in European Public Administrations (PA).

The Consortium will analyse and support the introduction of ODS and OS solutions in the PA by:

- Deploying ODS and OS software solutions in several European PAs, and benchmarking their effectiveness through a cost/benefit analysis;
- Building a European, multilingual, freely-accessible knowledge and experience base by comparing and pooling knowledge;
- Disseminating the results and the experiences of the study through a series of workshops at regional and European level.

In particular, the project focuses on the OpenOffice suite: a set of key desktop applications which includes a word processor, a spreadsheet, a presentation manager, a drawing program, and an equation editor [4].

COSPA is funded by the 6th Framework Programme (FP6) of the European Union and will run from January 2004 to December 2005.

2.1 The Consortium

COSPA is a consortium of fifteen European partners:

- **Academia**: Free University of Bolzano-Bozen (Italy, Coordinator), Computer and Automation Research Institute MTA-SZTAKI (Hungary), University of Aalborg (Denmark), University of Limerick (Ireland), University of Sheffield (UK);
- **Public Administrations**: Consortium of the Municipalities of the Province of Bolzano-Bozen (Italy), Torokbalint City Council (Hungary), Hanstholm Kommune (Denmark), Society of IT Management (UK), Beaumont Hospital (Ireland), South-West Regional Authority (Ireland), Province of Pisa (Italy), Province of Genova (Italy);
- **Industry**: Conecta srl (Italy), IBM Belgium SA (Belgium).

The structure of the Consortium is centered on university-PA couples. In fact, every PA is co-located with an academic partner, in order to constantly follow the evolution of the transition to OSS.

2.2 Workplan

The Consortium will introduce, analyse, and support the introduction in the PA of OSS and ODS. The workplan of the project can be divided in five main activities:

1) gathering and analysis of user requirements from the partner PAs, in order to devise possible OS solutions. The focus of this task is not to develop brand new applications, but rather to identify and combine OS software and ODS which fulfill the PA requirements;

2) pilot projects for deploying in the partner PAs the OS desktop solutions developed on the basis of the previous requirement study, in order to enable the subsequent cost/benefit analysis. Deployment will follow a two-step strategy: in the first step we focus on desktop applications only (mainly OpenOffice). In the second step we will also deal with desktop operating systems (Linux);

3) benchmarking of the deployed OS solutions, through a statistical and cost/benefit analysis. Financial, economic, reliability, effort, cost, and time aspects will be considered and integrated;

4) building a European knowledge and experience repository by comparing and pooling knowledge acquired in the previous phases of the project. The knowledge base will be placed on the Internet and made freely accessible;

5) dissemination of the results and the experiences of the project through the knowledge base and a series of workshops at regional and European level, with the aim of stimulate:
    - the exchange and sharing of knowledge among the partners of the Consortium;
    - public and business' awareness on the project and on OS in general;
    - interaction between users and OS developers communities.

3. Synergy with developers' community

The activities which might be of interest to OSS developers are mainly two:

- analysis of requirements for OSS/ODS in the PA (activity 1); and
- pilot projects introducing OSS in the PA (activities 2, 3).

3.1 Requirement collection and analysis

This activity aims at finding out what OSS can be used, why and what problems have been experienced in adopting/managing it, what parts of an application are actually used, what parts are too sophisticated or inappropriate for the PA, what critical PA applications use OSS. Various requirement gathering techniques will be used, including questionnaires, interviews, development of user stories, and so on.

Developers might benefit from the knowledge acquired in this phase of the project by checking how their applications are used in a corporate environment. Of course, developers may also consider PA's requirements as a driver for further evolution of their projects.

Work in this activity might possibly include the development of *ad-hoc* tools which will enable the successful integration of OS desktop software in existing PA environments. In particular, such tools would be expected to address interoperability issues between legacy databases and desktop applications. This could result in a direct involvement of the OS developers community. For example, a project's partner has already tackled this issue by developing a library for "bridging" Oracle databases with OpenOffice. Such library is expected to be released to the OS community.

3.1 Pilot projects

The objective of this activity is to run experiments on the introduction of OSS in the partner PAs, and to benchmark the effectiveness of the deployed OS solutions through a statistical and cost/benefit analysis. The analysis will consider financial, economic, reliability, effort, cost, and time aspects. The deployed OS solutions will of course be chosen on the basis of the previous requirement analysis.

Data on usage and satisfaction will be collected in the partner PAs by the universities, both manually and automatically. The automatic data collection of process and product metrics [5][9] is carried out using the PROM tool [10]. The data collected will form the core data source for the analysis of the effort required in the transition to OS.

The results of the pilot project phase would be of great interest to developers, as they could check how their applications and tools perform in a corporate environment. The analysis could identify strengths and weaknesses of OSS, bugs, security pitfalls, etc.

4. Observers

In order to increase dissemination of knowledge and to promote best practices in Public Administrations, the Consortium has established the role of *observer*. An observer can access the project's results and experiences in a privileged way. It may also attend project meetings and thus give useful advises on the implementation of the project itself.

At the time of writing COSPA has the following observers:

- University of Alberta, Canada;
- Victoria University of Wellington, New Zealand;
- UNESCO.

New observers may join COSPA at any time. On the project's website [2] it is available the application form for becoming a COSPA observer.

Other research projects involved in OS could well foresee and equivalent observer role, in order to foster collaboration with developer communities and spread knowledge.

5. Conclusions

Successful OSS seems to be so far confined in the system software area, of which Linux, Apache and Sendmail are notable examples. Desktop OSS does not share the same amount of popularity. We argued that this might be due to the fact that OS developers do not have access to user requirements: for example, Linux developers knew exactly what they needed in the system and knew how to do it. On the other hand, it seems that developers of desktop OSS are not in the same situation. In this paper we make the case for a close interaction between OSS desktop developers and the COSPA research project. COSPA studies the introduction of OS desktop software in the Public Administration and could therefore provide the OS developers community with valuable information about the requirements of a "corporate" sector. That might clearly help and guide the development of useful and user-friendly desktop applications.

6. References

[1] Netcraft Survey, http://www.netcraft.com/survey/

[2] http://www.cospa-project.org

[3] Free/Libre/Open Source Software: Survey and Study. June 2002. http://www.infonomics.nl/FLOSS/

[4] The OpenOffice project, http://www.openoffice.org

[5] W. Humprey, *Introduction to the Personal Software Process*, Addison-Wesley, 1997.

[6] A. Fuggetta, "Open source software - an evaluation", *Journal of Systems and Software*, 66 (2003), 77-90.

[7] E.S. Raymond, *The Cathredal & the Bazaar,* O'Reilly, 2001.

[8] R.S. Pressman, *Software Engineering: A Practitioner's Approach,* 6th ed, McGraw-Hill, 2004.

[9] N.E. Fenton and S.H. Pfleeger, *Software Metrics: a Rigorous and Practical Approach,* Thomson Computer Press, 1994.

[10] A. Sillitti, A. Janes, G. Succi, T. Vernazza, "Collecting, Integrating and Analyzing Software Metrics and Personal Software Process Data", *Proceedings of EUROMICRO '03*, 336-342.

Control Objectives in Open Source Projects

Kris Ven and Jan Verelst
University of Antwerp
Faculty of Applied Economics – Department of Management Information Systems
Prinsstraat 13, B-2000 Antwerp, Belgium
{kris.ven,jan.verelst}@ua.ac.be

Abstract

Many studies on Open Source Software Development (OSSD) have been published in the past years. In these studies, OSSD has received positive comments and is proposed as a new way of developing software. We investigate how project management is used in Open Source Software (OSS) projects to exercise control of development activities. Project management in OSSD is worth studying, because research has shown that project management is a critical success factor in traditional software development. We use the COBIT framework to audit project management practices commonly found in OSS projects with the aim of determining which practices are currently missing or can be further improved upon. The framework identifies several potential threats to the long term continuity of OSS projects.

1. Introduction

In most literature concerning Open Source Software Development (OSSD), the OSSD approach is presented as a new method of developing software, with the distinct characteristic that the source code of the project is freely available to everyone. Open Source Software (OSS) is frequently reported to be of equal or higher quality, compared to non-OSS [6,17,10]. Moreover, some studies have shown that the methodologies used in OSSD are based on best practices in software engineering [21], and that certain elements of the OSSD approach could be beneficial to the development of commercial software [1,10].

We certainly acknowledge these advantages of OSSD. We are however interested in how we can further increase the success rate of OSS projects. Research has shown that control exercised through project management is a critical success factor in traditional software development [2,5,15]. We think that these critical success factors can also be relevant for OSS projects. Therefore, we study project management in OSS projects. The aim of this paper is to show which concepts of control are currently present and/or missing in OSS projects. We will hereby make use of the COBIT framework, since this framework lists several control objectives for IT management. We will first briefly present the COBIT framework. Next, we will apply the COBIT framework to OSSD practices, and determine which processes are currently missing in OSS projects. Finally, we will discuss the implications of our findings for OSSD.

2. COBIT

COBIT stands for *Control Objectives for Information and related Technology* and is developed by the *IT Governance Institute* [14]. The aim of COBIT is to provide a framework that assesses the alignment between IT infrastructure and business processes and is an important tool in IT audit. Additionally, it provides guidelines for good management practices concerning IT.

The COBIT framework considers four different domains: *Planning & Organisation*, *Acquisition & Implementation*, *Delivery & Support* and *Monitoring*. For each domain, high-level control objectives are specified. There are a total of 34 high-level objectives, which can be further specified in 318 detailed control objectives. These control objectives have an impact on different *Information criteria* (effectiveness, efficiency, confidentiality, integrity, availability, compliance and reliability) and *IT resources* (people, applications, technology, facilities and data). Each objective can be relevant for one or more of these information criteria and IT resources. A schematic representation of the COBIT framework can be found in figure 1. The meaning of the different objectives (PO1 till PO11) can be found in table 1 and in section 3.

	Information Criteria							*IT Resources*				
objectives	effectiveness	efficiency	confidentiality	integrity	availability	compliance	reliability	people	applications	technology	facilities	data
PO1	P	S						Y	Y	Y	Y	Y
PO2	P	S	S	S					Y			Y
PO3	P	S								Y	Y	
PO4	P	S						Y				
PO5	P	P					S	Y	Y	Y	Y	
PO6	P					S		Y				
PO7	P	P						Y				
PO8	P					P	S	Y	Y			Y
PO9	P	S	P	P	P	S	S	Y	Y	Y	Y	Y
PO10	P	P						Y	Y	Y	Y	
PO11	P	P		P			S	Y	Y	Y	Y	

P = Primary, S = Secondary, Y = Applicable

Figure 1: The COBIT framework – Planning and Organisation

3. Application

In this section, we will apply the COBIT framework to OSSD. For our purposes, the *Planning & Organisation* (PO) domain is the most relevant. Hence, we will limit ourselves to the study of the 11 high-level control objectives that are defined for this domain. The conclusions we will present here are based on an exploratory study of OSS projects investigating which management practices are present or missing in OSS projects.

In some cases, we had to give the COBIT control objectives a more meaningful interpretation within OSSD context. This is due to the specific nature of OSSD. Nevertheless, our experiences show that even in its current form the COBIT framework is already capable of highlighting some fundamental issues in OSSD.

PO1 : Define a strategic IT plan This objective states the need to develop a strategic IT plan to maximise the fulfilment of the requirements and to ensure the further accomplishment of the project. Strategic planning should result in long-term plans which should be translated into operational plans and concrete short-term goals.

The importance of planning in ICT projects has also been illustrated in studies by Curtis et al. and Guinan et al. [2,5]. This means OSS projects as well should have a long term vision. More concrete, this could be in the form of a development *roadmap*. A short term planning is also required, for example, a concrete list of improvements and must-fix bugs for each upcoming release should be made so that each developer can concentrate his or her efforts on these issues.

The Mozilla project follows these guidelines, whereby the roadmap can be found on the project homepage, while some bugs can be marked as "blocking" a certain release. MySQL also has a roadmap in their documentation, illustrating which features should be implemented in which release.

PO2 : Define the information architecture This objective requires that all data available within a company is inventoried so that information use can be maximised, or in more simpler terms: ensuring that everyone in the organisation can access the data he or she needs.

Although primarily intended for management information systems, this guideline requires OSS projects to identify relevant information for developers and users. It will be important for example that everyone can easily access the latest source code (this is mostly done by CVS) and there is sufficient documentation for developers (on the software design) as well as for end users. Unfortunately, OSS projects generally provide little documentation, since it is commonly known that developers don't like to write documentation [16]. An example of this is ArgoUML of which several sections of the help file still have to be written. There are exceptions, such as OpenOffice.org which ships with an extended help program.

PO3 : Determine technological direction According to this objective, one has to fully maximise the potential of emerging technologies. New technologies can help to achieve the business goals or better support the current processes.

As OSS projects mostly involve IT enthusiasts, it seems reasonable to assume that they will be aware of new technological advantages and how they can be used in the development process. It is however doubtful that there are formalised processes to see how new technologies can help the development of the project. For changes in the technological infrastructure, money can be a limiting factor for OSS projects.

PO4 : Define the IT organisation and relationships This objective requires structuring the technical organisation of a project or company – where roles and responsibilities are clearly set for each position – and ensuring that these positions align with the business goals. This includes creating an organisational chart, setting job descriptions and staffing the positions.

This means that OSS projects should have at least a clearly defined structure of responsibilities, maybe in the form of an organisational chart. In certain projects such delegation of responsibilities is present. In the Mozilla project for example, there are module owners who are responsible for a specific part of the project. Although little in-depth research is performed on this topic, we assume that the forming of the organisational structure is an *ad hoc* process. In commercial enterprises, these structures usually form taking into account several contingencies in the environment [19].

PO5 : Manage the IT investment Managing investment costs include defining budgets, control of the actual spending and analysing and making investment decisions.

Little information is available on how OSS projects manage their finances. Since they are mostly dependent on gifts, the budgeting will probably consist of checking which infrastructure is needed and can be acquired (f.e. web servers).

PO6 : Communicate management aims and direction The goals and direction the organisation is heading (coming from PO1) need to be communicated throughout the entire organisation. Accompanying these communication channels are policies and practices employers should adhere to, in order to ensure the realisation of the goals.

This objective refers to how top management in enterprises communicate their goals to employees, to make sure everyone understands where the organisation is heading. In OSS projects we find a similar mechanism: the core team or project leader must be able to communicate the project priorities so that developers can focus their efforts accordingly.

Mailing lists, notices on the project website and IRC are possible channels through which the goals of OSS projects can be communicated to developers and users. Guidelines for developers can be explicit (f.e. coding standards published in the project home page) or implicit (rules that are learned by being involved in the project).

PO7 : Manage human resources Human resource management has to attract and motivate people in order to maximise their contribution to the project. This includes amongst others recruitment, compensation, training and promotion.

Looking at OSS projects, we see that in general, there is no formal HR management. Some informal means of promotion (such as taking on someone as core developer in a project) exist, but these promotions are solely based on efforts made in the past (the so-called *meritocracy* model of OSS projects [18]).

PO8 : Ensure compliance with external requirements This refers to which degree the project meets external obligations, such as legal and contractual restrictions.

An important goal for OSS projects is their compliance to (semi-)official standards set by organisations such as the World Wide Web Consortium (W3C). In general, OSS projects will try to use open standards where possible. On the other hand, OSS projects seem to frequently suffer from legal actions. Examples are the recent name change of Lindows to Lindash [9] and the name changing of Mozilla's Phoenix to Firebird and eventually to Firefox [4].

PO9 : Assess risks Risk management mainly concerns looking for ways to reduce threats – and therefore the risks – in the project. This consists of making an inventory of possible project risks and finding solutions.

A well-known source of risk is uncertainty about end-user requirements. This risk seems more limited than in traditional software development, since OSS is usually written by user/developers. A second source of risk are technical uncertainties. Unfortunately, little information exists on how risk assessments are made for OSS projects.

PO10 : Manage projects The management of projects is concerned with the timely and within-budget delivery of projects. This is basically project management in the strict sense of the word.

Several best practices of project management can be found in several OSS projects. The setting of milestones, setting quality targets and finding developers responsible for functional parts of the project are examples of this. Time and budget on the other hand, play a much smaller role in OSS projects. In general, there are no strict delivery dates that have to be respected. Therefore, the releases can be feature based. How this can lead to problems for commercial vendors was illustrated by the release of Netscape 6, which was based on Mozilla 0.6 [12]. Money is also a lesser concern, since most development in OSS projects is done by unpaid volunteers. Increasing manpower will therefore not lead to a high increase in cost.

PO11 : Manage quality Managing quality means ensuring that the software produced meets the expectations and requirements of the client or user. Quality control is an important part of any project.

Several studies have argued that OSS projects produce software of higher, or at least equal, quality compared to non-OSS projects [6,17,10]. Some of the tools and practices used in OSS projects, such as code review, seem to support this. In the Wine project this is taken to a higher level, where the priority of each core developer is code review. Each patch made by contributors must be sent by a mailing list where anyone can give comments on the patch. Only if it is certain that the patch will not introduce regression bugs, the patch is committed in the CVS repository [10]. Other quality assurance techniques are daily builds. Mozilla Tinderbox is an automated build system that compiles the program each day, and runs a series of tests to check whether regression bugs have entered in the last day.

Objective	Present
PO1 : Define a strategic IT plan	Y
PO2 : Define the information architecture	P
PO3 : Determine technological direction	P
PO4 : Define the IT organisation and relationships	P
PO5 : Manage the IT investment	N/A
PO6 : Communicate management aims and direction	P
PO7 : Manage human resources	N
PO8 : Ensure compliance with external requirements	P
PO9 : Assess risks	N/A
PO10 : Manage projects	P
PO11 : Manage quality	Y

Y = Yes, N = No, P = Partly, N/A = No data available

Table 1: Overview

4. Discussion

Table 1 shows an overview of our findings. In general, we find that most objectives are more or less covered in OSSD, although this is very different from project to project. Due to space limitations, we want to focus on three important observations.

First, we find that OSSD does not pay attention to Human Resource Management. This is at least a worrying fact, given the dependence of OSS projects on the *motivation* of their contributors. Several studies suggest that intrinsic rewards, such as peer recognition and personal satisfaction are the primary motivations of developers [7,3,11]. However, we feel there are several threats to this motivation. Some projects for example are heavily dependent on a project guru, such as Linus Thorvalds or Richard S. Stallman. The question is what will happen if these gurus leave the project. This will certainly be a problem in projects with a small number of core developers. Another example concerns the nature of the development task. While taking a look at OSS project archives such as Freshmeat or SourceForge, it shows that most of the active projects are still in development phase, where new features are added to the program. However, in traditional software engineering 50–70% of development efforts goes to maintenance [8,13], which is considered to be a less attractive task. It is therefore not impossible that when software becomes more mature, it will become more difficult to motivate people to work on maintenance. We hope that the factors that are motivating people at this moment to work on OSS projects, will continue to do so in the future. However, we feel that this motivation can be seriously threatened in the future by scenarios such as the two described above. Without capable and enthusiastic contributors, there is no future for OSS projects.

A second observation is that no matter how well control objectives, processes and guidelines are defined, the *effective control* of development activities still remains a problem. Most developers have no formal ties with OSS projects, and there is no way to force people to do a certain task [20]. This means that there is no way to force people to comply to certain guidelines that are set by the project team. It appears that in OSSD, control is largely based on *trust*. A project administrator for example must trust his co-developers and contributors that their contributions to the project will be worthwhile. A contributor on the other hand will need to know that the effort he will make in writing software for no fee, will be compensated by the benefits of using the resulting software. The role of trust in OSS communities has been downplayed by Gallivan [3], however we feel that some degree of trust will be present in OSS communities, albeit implicitly. An article by Markus et al. [11] further suggests that social control and self control are sufficient control mechanisms, and that company employees should even be managed as unpaid volunteers. In our opinion, Markus et al. are overly optimistic: currently, there is little evidence that these management practices will work in practice.

A third observation concerns the use of the COBIT framework. The need for an *audit framework* for OSS projects is large. Potential users could base their decision on it whether to adopt the software or not. Companies considering to fund OSS projects might use it to assess the stability or potential of the project. Our study shows that COBIT in its current form is capable of highlighting a number of fundamental issues in OSSD. Further research is required to determine whether audit frameworks tailored to OSSD are required.

5. Conclusions

In this paper, we applied the COBIT framework to OSSD in order to study how control is exercised by using project management. This leads us to three important conclusions.

First, a basic level of support for the high-level control objectives in the Planning & Organisation domain seems to be present in OSSD.

Second, COBIT identifies a number of threats to the long term future of OSS. We consider it possible that the initial motivation of developers will disappear as project gurus find new challenges and the programming work changes from developing exciting new systems to cumbersome maintenance. Furthermore, in such difficult times, effective control would enable the project team to lead the project in a new and better direction. However, it is this control that is missing. Practising good project management may result in certain improvements, but project management alone won't be sufficient if motivation decreases. More research on how motivation can be stimulated in OSS projects seems appropriate.

Research in the field of non-profit organisations relying on volunteer work might prove useful in this context.

Third, more research is required on audit frameworks for OSSD. They would be useful to users deciding to adopt a certain OSS project, as well as to companies assessing the project potential. Such frameworks should try to incorporate as much traditional management practices as possible, but take into account the specific nature of OSSD.

References

[1] A. W. Brown and G. Booch. Reusing open-source software and practices: The impact of open-source on commercial vendors. In C. Gacek, editor, *Proceedings: Seventh International Conference on Software Reuse*, volume 2319 of *Lecture Notes in Computer Science*, pages 123–136. SpringerVerlag, 2002.

[2] B. Curtis, H. Krasner, and N. Iscoe. A field study of the software design process for large systems. *Communications of the ACM*, 31(11): 1268–1287, Nov. 1988.

[3] M. J. Gallivan. Striking a balance between trust and control in a virtual organization: a content analysis of open source software case studies. *Information Systems Journal*, 11(4): 277–304, 2001.

[4] S. Garrity, G. Markham, B. Goodger, B. Decrem, et al. Mozilla firefox – brand name FAQ. http: //www.mozilla.org/projects/firefox/firefox-name-faq.html.

[5] P. Guinan, J. G. Cooprider, and S. Faraj. Enabling software development team performance during requirements definition: A behavioural versus technical approach. *Information Systems Research*, 9(2): 101–125, June 1998.

[6] T. Halloran and W. L. Scherlis. High quality and open source software practices. In *Meeting Challenges and Surviving Success: Second ICSE Workshop on Open Source Software Engineering*, Orlando, Florida, USA, May 25, 2002.

[7] J. Lerner and J. Tirole. Some simple economics of open source. *Journal of Industrial Economics*, 50(2): 194–234, June 2002.

[8] B. P. Lientz and E. B. Swanson. *Software Maintenance Management: A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations*. Addison-Wesley, Reading, MA, 1980.

[9] Lindash. http: //www.lindash.com/.

[10] S. Lussier. New tricks: How open source changed the way my team works. *IEEE Software*, 21(1): 68–72, 2004.

[11] M. L. Markus, B. Manville, and C. E. Agres. What makes a virtual organization work? *Sloan Management Review*, 42(1): 13–26, 2000.

[12] Mozilla Project. Old mozilla milestone plan. http: //www.mozilla.org/releases/milestones.html.

[13] J. T. Nosek and P. Palvia. Software maintenance management: changes in the last decade. *Journal of Software Maintenance*, 2(3): 157–174, 1990.

[14] IT Governance Institute. *COBIT framework*. Information Systems Audit and Control Foundation, 3 edition, July 2000.

[15] The Standish group. The CHAOS report, 1995.

[16] J. Sametinger and G. Pomberger. A hypertext system for literate c++ programming. *Journal of Object Oriented Programming*, 4(8): 24–29, 1992.

[17] W. Scacchi. Is open source software development faster, better and cheaper than software engineering? In J. Feller, B. Fitzgerald, F. Hecker, S. A. Hissam, K. R. Lakhani, and A. van der Hoek, editors, *Meeting Challenges and Surviving Success: Second ICSE Workshop on Open Source Software Engineering*, Orlando, Florida, USA, May 25, 2002.

[18] W. Scacchi. Free and open source development practices in the game community. *IEEE Software*, 21(1): 59–66, 2004.

[19] J. D. Thompson. *Organizations in action: social science bases of administrativetheory*. McGraw-Hill, New York, 1967.

[20] J. Warsta and P. Abrahamsson. Is open source software development essentially an agile method? In J. Feller, B. Fitzgerald, S. A. Hissam, and K. Lakhani, editors, *Proceedings of Taking Stock of the Bazaar: The 3rd Workshop on Open Source Software Engineering, the 25th International Conference on Software Engineering (ICSE)*, pages 143–147, Portland, Oregon, USA, May 3, 2003.

[21] L. Zhao and S. Elbaum. Quality assurance under the open source development model. *Journal of Systems and Software*, 66(1): 65–75, Apr. 15, 2003.

Open Source and Closed Source Software Development Methodologies

Vidyasagar Potdar, Elizabeth Chang
School of Information System, Curtin University of Technology, Perth, Australia 6845
PotdarV@cbs.curtin.edu.au, ChangE@cbs.curtin.edu.au

Abstract

Open source software development represents a fundamentally new concept in the field of software engineering. Open source development and delivery occurs over the Internet. Developers are not confined to a geographic area. They work voluntarily on a project of their choice. As new requirements emerge, the software is enhanced by the user/developers. In this paper we show a comparative study of open source and closed source software development approaches and present a software life cycle model for open source software development.

1. Introduction

The concept of free software is not new. It has been around since the 1960s in universities such as MIT and corporate firms such as Bell Labs who freely used source code for research [1,2,3,4]. Software was not a means of revenue generation but was used to hook more and more customers to buy new computers [5]. In the early 1980s, Microsoft started writing software for the sole purpose of profit. It gave only compiled code; source code was hidden from the user. This move had a great impact and could be considered as the birth of open source. Richard Stallman, researcher at MIT, founded the 'Free Software Foundation' (FSF) to develop and distribute software under the General Public License history (GPL) while Bruce Perens defined a set of guidelines that a software license must grant its user, and he called this Open Source Initiative (OSI). In this paper, we briefly describe the open source software development and compare it with closed source software development. The paper is organized as follows; in section 2, we describe how open source software is developed. In section 3, we compare open source and closed source software development approaches. In section 4, we propose a life cycle model for open source software development.

2. Open Source and Who, Why, What?

Bruce Perens defines that Open Source is a specification of what is permissible in a software license for that software to be referred to as Open Source [2].

2.1 Who is an open source developer?

Simply put, "any one who contributes to the open source project is an open source developer", such as a user of the software, a developer who develops the software, a debugger or hobbyist who likes spending time on open source, or a promoter who funds such a development.

2.2 Why do they produce open source?

Eric states that developers are attracted towards open source development because that gives them an opportunity to demonstrate their ability. So they voluntarily select a project and start contributing. When programmer's, code gets accepted, it boosts their ego and they get recognized for their effort in the community, [7] Peer recognition creates reputation and a reputation as a good programmer is a great achievement.

2.3 What do they do in open source?

Open source developers are involved in a variety of activities such as designing, coding, debugging and utilizing. Each activity occurs simultaneously. Parallel development and debugging is the key to open source success. Users also play a vital role in the debugging process by reporting bugs to developers or sometimes fixing it themselves. Developers are well aware that users are the best beta testers [7,8].

2.4 What are the major customer concerns?

2.4.1. Maintaining consistent software architecture. This is one major concern, but it's outside the scope of this paper [12, 13].

2.4.2. Support and coordination for deployment. When end users utilize open source component they want to know whether the component is fully automated or does it need any integration with other components. However when end users come up with bug reports, feature

requests or installation guidelines they expect a full support and coordination from the developers.

2.4.2. Managing upgrade and complexity. The end users are concerned with the future upgrades. They are not sure, as components get more and more powerful and large, whether the complexity of the code can be automatically managed. Managing complexity is a big concern because OSS often has parallel developments going on [1].

3. Open Source versus Closed Source Software Development

Open Source Software Development (OSSD) is a recent phenomenon, while traditional closed source software development (CSSD) has been here since the dawn of software development. One major difference between these two models is source code visibility. In this section we will point out most of the differences between these approaches. We begin with their process models.

3.1. Process Models

CSSD normally follows a spiral or iterative model of development i.e. software development goes through all phases like planning, design, implementation [14] whereas OSSD follows an evolutionary model for development where the software never reaches a final state and keeps on evolving according to customer needs [15]. It's more of a concurrent or parallel process. CSSD has clear cut milestones using which the progress of the project can be tracked but in case of OSSD it not possible although CVS do help to keep some track.

3.2. Requirements Definition and Specification

CSSD starts with requirements definition and specification. Here requirements are vague. Project developers are not aware of the actual requirements. They need to interview stakeholders to elicit requirements and then start implementing [1, 14]. On the contrary OSSD starts with a motive of requirements satisfaction. Requirements are clear, as developer is fully aware of the requirements [15]. In case of CSSD all the user requirements may not be implemented because of time or budget constraints [16, 14] where as in OSSD it possible because user is often the developer [1, 15]. In case of CSSD system architect and project manager decide which requirements will be incorporated while core members of the open source project decided which requirements to be implemented [14].

3.3. Documentation

In case of CSSD project plan is document and followed. Once the requirements are clear they are documented [14]. Even the designing and testing procedure is documented. Where as in OSSD there may or may not be any official documentation. [14,16]

3.4. Analysis and Design

In CSSD, system architects and project mangers spend a lot of time in designing the project [14], whereas in OSSD designing is often merged with implementation.[17, 12, 15].

3.5. Software Architecture

Maintaining consistent software architecture is enforced during the development phase itself, there is rarely a drift between software's conceptual and concrete architecture. [12, 13, 14]. In case of OSSD this has been recognized as a major concern. Maintaining consistent software architecture is difficult because of its highly collaborative and distributive nature. [12, 13]

3.6. Implementation

Only one implementation is possible for one requirement [14] whereas in case of OSSD multiple implementations are possible for the same requirement. This is considered as a major issue as it results in code forking. The rate of development is comparatively slower that the open source because the number of developers assigned to a CSS project can never match a full-scale open source project like Linux [6, 7, 14]

3.7. Source Code

In CSSD source code is hidden from the user while in OSSD source code is open as a result user can view and modify the code to suit individual needs [2, 15]. Such freedom is not available in closed source software.

3.8. Testing

In OSSD, users act as bug reporters and beta testers. Whenever a user finds any bug in the software they either try to solve it or bring it to the notice of the community [1]. But commercial closed source products use service packs to repair bugs [15]. OSS community believes "*no bug can survive wide testing*" [7].

3.9. Release and Delivery

Open source products are released quite often on a daily or weekly basis whereas closed source products are released on a yearly basis. In commercial softwares,

product is often released due to marketing pressure and tight schedule whereas open source products are released once the developer thinks that it has reached a functional stage [1, 14].

3.10. Maintenance

Service packs are needed quite often to repair bugs in commercial closed source products, whereas bug reporting and bug fixing takes care of maintenance in case of open source products.

3.11. Product

Close source products may soon reach a finalized state once the documented requirements are implemented whereas open source products are always in a evolutionary phase because as requirements emerge they get implemented [14].

3.12 Type of Software

Commercial software development is more of a solution kind of development. Developers create solutions for a big company. It is more like customized development whereas open source development is more components-based i.e. plug-n-play type software. Developers create small programs, which work on a variety of platforms.

3.13. Security

It is a common belief that commercial closed source product is highly secure because it is developed by a group of professionals confined to one geographical area under a strict time schedule. But quite often this is not the case, hiding information doesn't make it secure, it only veils weaknesses [10]. In CSS security is achieved through obscurity, however in OSS security is achieved through 'open source'. The ability to modify the source code works to your advantage if you want to deploy a highly secure system. One can ask for a third party security certificate or get the system scrutinized by a professional security expert for possible back door entries. [10, 11], such freedom is not available with commercial closed source products. Another argument that supports open source security is community reaction to bugs. Community reaction to bug reports is much faster compared to commercial closed source which makes it easier to fix bugs and make the component highly secure.

3.14. Productivity, Quality and Cost

Developer's productivity may decrease if they are forced to work on a project in which they are not interested, which is contrary to OSSD where developer is free to choose the project on which they want to work [15]. Developing open source software is faster, better and cheaper. All the three factors can be satisfied simultaneously. Cost is reduced because no one is paid for the job everyone is a volunteer. Speed is increased because development is parallel and collaborative in nature. And finally quality is maintained because the product is released only when the developer think the product is stable and functional [16]. However with CSSD this doesn't work well. At one time only one factor can be satisfied fully. E.g. if speed is maintained quality and cost may go up or if cost is to be maintained quality and speed may go down [9, 14, 16] Hence CSSD can be considered slow and expensive.

3.15. Environment

Often we find centralised, single site development in CSS while decentralised, distributed, multi-site development in OSS. In CSS development happens in a geographically confined area, while in OSS development occurs on the Internet [1, 14].

3.16. Group work and Communication

Open source is co-operative and need high level of co-ordination over the Internet and multi-site. Lack of coordination among developers results in code forking [1]. However in case of CSS, inconsistency is easily managed by face to face or weekly team meeting.

4. New Life Cycle Model for Open Source

Software life cycle determines the set of activities that constitute a software project. It may not be surprising that the OSSD life cycle differs from CSSD [15].

4.1. Traditional life cycles do not suit open source

In case of waterfall model, after every stage documentation is done. But in case of OSSD often simultaneous development occurs. OSSD has more of a modular approach. Several individual developers can simultaneously work on several different modules without worrying about the final integration. This is one of the reasons why waterfall model doesn't suit OSSD. Waterfall model is linear in nature. It has clear milestones. But OSS has very vague milestones. Because of its evolutionary nature it never reaches a stage where we can confidently say it has finished a particular phase. As OSSD has vague milestones it's really difficult to point out its progress. This is another reason why we feel the existing waterfall model is not suitable for OSSD.

4.2. Some phases in traditional software life cycle do not apply to open source

Commercial CSSD is confined to a geographic location. During planning stage project leader organizes teams, schedules meetings, and assigns roles and responsibilities to the team members. When we compare this with OSSD we figure out that there are no teams, no meetings at all as all the communication happens through mailing lists or project web sites. User or developer is free to choose a project of their own choice. This removes the need of assigning roles to developers.

In CSSD planning is followed by requirement elicitation, requirement documentation and feasibility analysis. While in case of OSSD the requirements are not vague, they are clear to the developer, as most often a user is also a developer. Apart from that, open source development sites like www.sourceforge.net, www.kde.org have a feature where users can request new features to be implemented in particular software, which reduces the need for requirement elicitation. Since the requirements are clear to the developer they just need to be analysed and implemented. This removes the requirement elicitation phase from OSSD. Design is generally followed by implementation. Since there is a large list of requirements, normally each and every requirement does not need detailed design. In such cases requirements are directly implemented. So we remove the detailed design phase from OSSD.

Multiple implementations create complexity and raise the issues for proper coordination. Traditional life cycle didn't tackle this problem but this is a phase which we think should be introduced in open source SDLC.

Concurrent implementations create architectural defects. As the user requirements change, the software changes to fulfill these requirements. Often these changes are done without considering the conceptual architecture. As changes go on adding up; the software's concrete architecture starts drifting from its conceptual architecture. OSSD is highly prone to this because of its collaborative and distributive nature. Hence we need a phase to tackle this issue as well [12, 13].

OSS lacks coordination and hence it is sometimes difficult to manage. This necessitates the need for a separate or simultaneous phase to manage complexity and maintain coordination. This phase is normally not taken care off in traditional life cycle [1].

We hope it is clear from the above discussion that there is a necessity for some phases while others are not required. The issues outlined above need to be considered when a SDLC model for open source is proposed.

We propose a requirement oriented pendulum model to solve some of the issues highlighted above. The proposed model tries to provide guidelines on how open source project should proceed. Our main aim is to try and estimate the progress of an open source project.

5. Requirement Oriented Pendulum Model for Open Source Software Development

We propose the requirement oriented pendulum model, to provide a mechanism to track which requirements have been submitted and which of those have been implemented and tested. The proposed model has four stages which we described below.

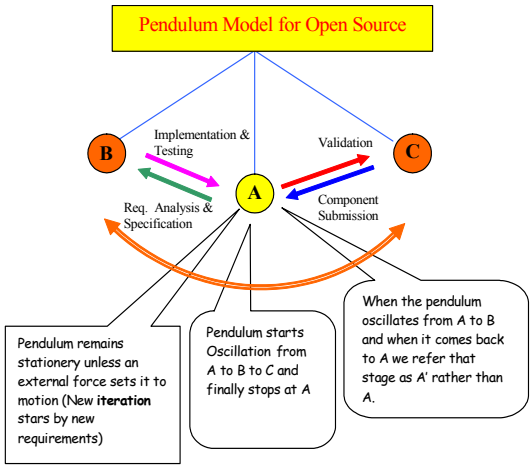


Figure 1: The Pendulum model for Open Source

5.1. Requirement Analysis and Specification.

In the pendulum model this is the first stage in which requirement analysis is done. Different users in the system have different requirements. Users submit feature requests which become new requirements. At this stage the pendulum is in the center. When the requirement analysis and specification starts the pendulum gets into motion. Once requirements are analyzed it reaches the left most end, which is represented by state B.

Initial State	Oscillation from	Reason for transition	Final State
A	A to B	Req. Analysis & Spec.	B
B	B to A'	Implementation & Testing	A'
A'	A' to C	Validation	C
C	C to A	Component Submission	A

5.2. Implementation and Testing

Two main activities in this phase are implementation and testing. Major outcome of this stage is source code. Coding involves writing source code to implement a requirement. Once the requirement is implemented it should be tested. So in the second half of this phase testing should be conducted. It requires careful planning and coordination. Major goal should be to make sure all components created during implementation function properly. Apart from that, this stage should include alpha

testing and beta testing. The testing should start with standard test cases. Each test case should contain information like test case name, action to be performed, data to be used, expected outcomes, actual outcomes. Normally this is done by the users or by the developers. When the implementation and testing starts the pendulum moves from B to A'. It reaches the centre, which is represented by A'.

5.3. Validation

This is the next iterative phase in the pendulum model. Major goal of this phase is to carry out field testing and validation and then generate reports. Bug reports are a major result of validation. Bug reporting should be done by user or developers. This can be done in many ways, but it is suggested that one should use a bug reporting software like Bugzilla or emailing the bug reports in a pre-specified format to the project leader. However before reporting a bug one should make sure that someone has not reported it earlier. When validation starts, the pendulum moves from A' to C. It reaches the other end, which is represented by C.

5.4. Component Submission

This is the final phase of the pendulum model. This phase has two activities submitting new feature requests and submitting tested and validated components. Once the validated components are submitted a new version of open source component is developed. At the same time new feature request which are submitted by the users or the developers sets the pendulum into motion once again. Feature request becomes the basis of new requirement and as discussed above sets the pendulum in motion. When component submission starts, the pendulum moves from C to A. It reaches the initial position once again. This is when one full oscillation completes.

6. Conclusion

From the study that we have conducted, it has come to our notice that OSSD is similar to its traditional counterpart in many aspects, but there are many areas in which it differs tremendously and these features make it different from the CSSD. As a concluding remark, we can say open source software is a competent alternative to CSS and the pendulum model that we proposed can very well demonstrate the functioning of OSSD.

7. References

[1] Steven Webber, "The Political Economy of Open Source Software", California, June 2000.

[2] Bruce Perens, "The Open Source Definition" Available at: http://perens.com/Articles OSD.html Acc. on: Oct 2002

[3] Malcolm M. "Profit Motive Splits Open Source Movement", Aug 26th, 1998. Available at http://content.techweb.com/wire/story/TWB19980824S0012 Acc. on: Oct 2002

[4] "What is Free Software Foundation"? Available at: http://www.gnu.org/fsf/fsf.html Acc. on: Oct 2002

[5] "Overview of GNU Project" Available at: http://gnu.j1b.org/gnu/gnu-history.html Acc. on: Oct 2002

[6]Ko Kuwabara, "The Bazaar at the Edge of Chaos" Chap 2: A Brief History of Linux. December 1999. Available at: http://www.cukezone.com/kk49/linux/chapter2.html. Acc. on: Oct 2002

[7] Raymond, E.S., "The Cathedral and the Bazaar" O'Reilly & Associates, 2000.

[8] Vinod Valloppillil, "Open Source Software, A (new?) Development Methodology" Nov 1998.

[9] Lerner, Joshua and Tirole, Jean, "The Simple Economics of Open Source" (February 2000). http://ssrn.com/abstract=224008

[10] Ferrara Linux User Group, "Open Source and Security" 2001 Available at: http://members.ferrara.linux.it/pioppo/aeronautica2001/opensecurity-2x1.pdf Acc. on: Nov 2002

[11] Dare Obasanjo, "The Myth of Open Source Security Revisited v2.0", 2002. Available online: http://softwaredev.earthweb.com/sdopen/article/0,,12077_990711,00.html Acc. on: Nov 2002

[12] Tran, J.B., Holt, R.C., "Forward and Reverse Architecture Repair" *Proc. Of CASCON '99*, Toronto, pg 15-24, 1999

[13] Tran, J.B., Godfrey,M.W., Lee, H. S., Holt, R.C., Architectural Repair of Open Source Software. *In Proceedings of International Workshop on Program Comprehension*, Limerick, Ireland, June 2000.

[14] Satzinger, Jackson, Burd "System Analysis and Design in a Changing World", Thomson Learning, 2000.

[15] Scott H, Charles W, Plakosh D., Jayatirtha A., "Perspectives on Open Source Software", Software Engineering Institute, Pittsburgh, Nov 2001 p49.

[16] Walt Scacchi, "Is Open Source Software Development Faster, Better and Cheaper than Software Engineering?", *23rd International Conference on Software Engineering*, Toronto, Ontario, Canada, 2001.

[17] Tran, J.B., "Software Architecture Repair as a Form of Preventive Maintenance", Masters Thesis, University of Waterloo, 1999.