2011

Proceedings of the OSS 2011 Doctoral Consortium

Charles M Schweik, *University of Massachusetts - Amherst*
Imed Hammouda, *Tampere University of Technology*

Tampereen teknillinen yliopisto. Ohjelmistotekniikan laitos. Raportti 20
Tampere University of Technology. Department of Software Systems. Report 20

Charlie Schweik & Imed Hammouda (Eds.)

Proceedings of the OSS 2011 Doctoral Consortium, October 5, 2011, Salvador, Brazil

TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

Tampereen teknillinen yliopisto. Ohjelmistotekniikan laitos. Raportti 20
Tampere University of Technology. Department of Software Systems. Report 20

Charlie Schweik & Imed Hammouda (Eds.)

Proceedings of the OSS 2011 Doctoral Consortium, October 5, 2011, Salvador, Brazil

Tampere University of Technology. Department of Software Systems
Tampere 2011

ISBN 978-952-15-2722-7
ISSN 1797-836X

Preface

We are honored to introduce the proceedings of the OSS 2011 Doctoral Consortium (DC). This Consortium was collocated with the 7th International Conference on Open Source Systems (OSS 2011) held in Salvador, Brazil on October 5th, 2011. Like DCs that have preceded this one, our goal was to provide doctoral students conducting research on open source systems an opportunity to share and discuss their goals, methods, and in some cases results before completing their PhD studies. Over the course of the day, we had seven accepted student papers presented and roughly thirty participants in the audience. As should be in an international conference, student participants came from all over the world, including the USA, Europe, Asia and South America.

What follows are revised papers presented by our participants. Each student gave twenty-minute presentations of their research proposal and in some cases elaborated on their current status. Each presentation was followed by twenty-minutes of open discussion during which PhD students were provided with feedback on their work from faculty members as well as other PhD students in the audience. The students themselves then worked with us to read and edit the papers in the process of producing this volume.

We hope that all PhD students who participated benefited from this experience, and we wish them all the best as they work to complete their research.

November 2011

Charlie Schweik
University of Massachusetts, Amherst, USA

Imed Hammouda
Tampere University of Technology, Finland

Acknowledgements

We would like to acknowledge the contributions of several people and organizations for helping make the 2011 OSS Doctoral Consortium possible. First, we are grateful to Alberto Sillitti and Paula Bach for their assistance in the planning and selection of submitted papers. Second, thanks go to to the relatively large group of international faculty (and others) who took an extra day to attend the DC and provide feedback to the presenters. Walt Scacchi, Greg Madey, Fabio Kon, Klaas_jan Stol and Bjorn Lundell come immediately to mind, but there were quite a few others. In retrospect, we regret not taking audience attendance for there were too many in the room for us to remember (or even meet). If you were there – this "thanks" goes to you. We are also grateful to the larger OSS 2011 conference organizing team – Scott Hissam, Barbara Russo, Fabio Kon, Manoel de Mendonca Neto, Bruno Rossi and Greg Madey for their help both before and during the DC and the team of people in Brazil who helped organize the meeting room. A special thanks goes to the United States National Science Foundation for providing fellowship support to several of our participants under award number IIS-0447623, and to the Tampere University of Technology in Finland for helping to officially publish these proceedings. Finally – and perhaps most importantly – thanks to the PhD students who, through their hard work and effort made the DC a success.

Doctoral Consortium Organization

Doctoral Consortium Co-Chairs

Paula Bach, Microsoft Corporation, USA
Charles M. Schweik, University of Massachusetts, Amherst, USA
Alberto Sillitti, Free University of Bolzano, Italy

Program Committee

Björn Lundell (University of Skövde, Sweden)
Fabio Kon (University of São Paulo, Brazil)
José Carlos Maldonado (University of São Carlos, Brazil)
Joseph Feller (University College Cork, Ireland)
Kevin Crowston (Syracuse University, USA)
Tony Wasserman (Carnegie Mellon University, USA)
Walt Scacchi (University of California, USA)

Contents

The Emergence of Quality Assurance in Open Source Software Development

Adina Barham[1]
1 Hitotsubashi University, Graduate School of Social Science, 2-1 Naka, Kunitachi, Tokyo 186-8601, Japan
adina.barham@yahoo.com

Abstract. An increasing number of open source software projects are formally defining a QA step in the development cycle. This research seeks to establish what kinds of open source software projects are adopting QA practices and at what stage in their maturity, how projects define QA and which community members are undertaking QA.

Keywords. QA, software quality assurance, social network analysis, communication patterns.

1 Introduction

More and more open source software projects have dedicated quality assurance teams. This suggests that the open source development model is changing, and furthermore that the structure of open source software communities may be changing as well. Research on the identity and dynamics of these emerging QA groups within open source software projects is necessary in order to keep our understanding of open source development practices up to date. Although much research has been done on open source communities, little attention has been paid to the people performing the quality assurance phase, for example regarding their backgrounds, expertise, and history of involvement with other software projects. This research aims to analyze how quality assurance is integrated in the open source movement, how it is impacting the traditional open source development model and who exactly is performing it.

2 Background and Motivation

Initially, software quality assurance was performed by software companies or it was carried out in a shallow fashion using black-box testing techniques [4]. This made it difficult to evaluate quality attributes without proper access to information. However, the emergence of QA in open source software, with its transparent communications and results, promises to allow scholars to study quality assurance processes more thoroughly than was previously possible.

Adina Barham1

Software development processes have evolved substantially in recent decades, becoming more complex and requiring a more structured and rigorous quality assurance process in order to produce stable solutions that meet the high standards demanded by users and customers. For this purpose a clear definition of quality was needed in order to endorse improvement in software; hence the ISO/IEC 9126-1 [10] which was updated and replaced in March 2011 with the ISO/IEC 25010 [11] standard. The first part of the standard ISO/IEC 9126-1 classifies software quality in a structured set of characteristics and sub-characteristics such as: functionality, reliability, usability, efficiency, maintainability and portability. The significance of this classification is that the software quality assurance process implies not only testing the application but also a series of steps necessary to ensure high quality standards. We would expect that as a project matures so does the testing process around it and according to Dibona [2], this is indeed true for both open source and proprietary software.

The importance of quality assurance in open source projects has recently been recognized as a major issue that needs further study. For example, the European Commission has started the QualiPSo [14] project (Trust and Quality in Open Source Systems http://www.qualipso.org). QualiPSo aims to increase the level of trust in open source software by defining and implementing technologies, processes and policies to facilitate the development and use of open source software components. Among research in other areas, the QualiPSo project is attempting to define trustworthy QA processes for open source software by developing various models, strategies and tools that assist in performing QA.

The structure of groups of individuals performing quality assurance, as well as their links with other groups are important aspects that need to be further analyzed due to the fact that it may impact the classical open source development model used so far. Previous research has addressed the structure of the open source communities and communication patterns. For example, Crowston and Howison [3] analyzed the social structure of 124 open source software projects, and found no consistent pattern of centralization or decentralization in the FLOSS projects that they studied in relation to bug fixing. However they suggested that there is a negative relationship between project size and centralization due to modularity. In a paper of particular relevance to this research, Mockus et al.[5] found that in the Apache **httpd** project bug reporting was quite decentralized in contrast to development; this raises the question of how QA is performed in other projects, and particularly when it is undertaken as an organized activity.

If we understand who is performing the quality assurance process and how this process is influenced then we can use and further develop these findings into best practices that may come in useful when performing QA and team building for both open source software as well as proprietary software.

3 Research questions

Which OSS projects have formal QA procedures, and how do they define QA? This question covers topics such as the ways in which QA teams operate, the backgrounds and skill-sets of people doing QA, the time commitment for an individual performing QA tasks, and the extent to which a few people perform most of the tasks while a "long tail" makes only occasional contributions. Unfortunately, due to time and resource constraints it is impossible to cover all these topics in this research.

How do QA contributors fit into project communities? According to the onion model of open source projects[3], community members can be split into three main categories: active users, co-developers and core developers. However, more recent studies [17] have suggested that the transition between the core developers and outer layers of the onion is not "as smooth" as we thought. According to Masmoudi et. al [18], 20-25% of bugs are initiated by outsiders and migration patterns can be observed in the sense that outsiders become members or the other way around. This research project aims to investigate the extent to which QA is a step on the road from end-user to developer, or whether it is become established as a separate category of contributor.

What are the dynamics of QA teams? This research seeks to establish the ways in which core developers and QA work together, the mechanisms for coordinating their work, and the extent of migration between quality assurance teams and developer teams.

Why do projects establish QA teams? Answering this question requires an understanding of how QA practices and diffuse between projects, which in turn requires a survey of how quality assurance contributors work across multiple projects. It would also be relevant to consider the stage in projects' evolution at which quality assurance comes to be considered a necessary pre-release step.

The aim is to explore the relationships between users performing quality assurance on different open source projects. Another important question in this context is whether there is a direct link between firms' involvement and the existence of dedicated quality assurance teams. Do independent projects have a formal quality assurance step?

Does project type affect QA practices? Another variable that might influence the existence and character of a formal quality assurance step is the project type. In particular, projects developing software intended for use by non-technical users might tend to evolve a particular set of QA practices. On the other hand, widely-used, mission-critical software applications aimed at IT professionals, such as web-servers, may develop a different set of formal QA steps.

4 Proposed research method

This research employs quantitative research methodologies to shed light on these issues. Mailing lists, bug trackers, forums, wikis and any other form of

communication used by the open source software communities to store actions performed in the quality assurance phase will be primarily analyzed. By including all communication channels, members performing QA tasks and members performing other activities can be easily categorized using their activity levels or actions as opposed to directly asking community members who may or may not respond or give biased answers. Also, determining what projects have a formal QA step, how they define it and how they implement QA in the development process can be analyzed by mining the same datasources. Applying quantitative methods in the first phase of the research process will provide a larger amount of data that can be further analyzed with social network analysis methods that will lead to answering many of the questions raised in the previous section regarding communication patterns, project migration, evolution within project community as well as central figures that coordinate the QA effort.

The first important step in gathering data will be identifying the projects that are most relevant to this research. These projects should have reached a certain maturity level in order to analyze their evolution over time. Also they should have at least an associated bug tracker and a mailing list dedicated to communication between members that are performing quality assurance. Size, programming language in which they were developed and popularity rate will also be an important factor in deciding which projects will be included in the data sample. The main focus will be on projects in which the main communication language is English but if relevant data is available for projects in which communication is in any other languages then they will not be excluded from the data sample.

The FLOSSMetrics [8] database is one of the largest databases of quantitative information regarding open source software; it covers about 2800 projects including ones from forges such as KDE, SourceForge, ObjectWeb or OSOR. However, the preliminary analysis suggests that the FLOSSMetrics database will not provide sufficient data, especially mailing list archives, of many of the world's most widely used open source projects.

From previous experience and other related research articles, it is expected that considerable cleaning of mailing list and bug tracker data will be necessary in order to carry out reliable, automatic data analysis. This step will be performed by writing cleaning scripts, running already available tools and correcting manually the remaining errors and inconsistencies.

To find a possible link between the existence of a defined QA team and type, projects will be divided into categories. For example, some projects will have a clearly defined quality assurance team, while in other projects core developers must perform certain quality assurance steps.

The operationalization process will include methods used by researchers that analyze social networks as well as methods previously used in the analysis of open source software communities. For example, the social network analysis approach will include global or node specific metrics such as closeness centrality, betweenness centrality, degree centrality and so on as well as network visualization that allows comparison between similar networks, peculiarity discovery and so on.

Projects participants will be represented as nodes (vertex) while interactions will be represented as edges (arc). Defining interactions between members could impose

some difficulties in the sense that data will be collected in different formats and in some cases thread-based analysis could be difficult to implement. An alternative solution would be quotation-based analysis in which participants quote e-mails to which they are replying. Previous research successfully used both quotation-based analysis as well as thread-based analysis [15,3], for example, Crowston et. al defined a link between two developers as a reply (or follow up) to the previous message posted on the projects' bug trackers. The number of e-mails sent from one member to the other will be represented as weight (value) of the graphs' arcs. For example, for the graph represented in figure 1, Peter replied to Amy's messages five times, while Mike never replied to any message.

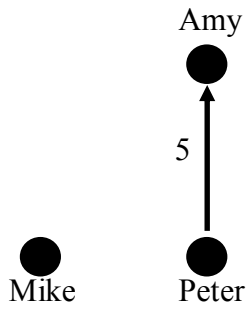

Fig. 1. Network representation

Users that are active on the QA mailing list may be anything from core developers to end users. It is therefore important to find out precisely to which group or groups each participant belongs. This implies that other mailing lists and communication channels should be taken into consideration. For example, if a member is highly active on bug trackers and at the same time submitting code frequently then it is safe to assume that he is not only performing QA assurance tasks. Also, in order to track migration patterns between teams, the participants' activity history in different teams in different periods of time must be checked. Bug trackers will be a useful source of data regarding communication between QA members and developers which will allow the author to identify how QA contributors fit in project communities.

The structure of the groups taken into consideration will be represented using specialized software. Interactions will be represented separately for each project taken into consideration in order to visualize special properties. To find out if people move from one project to another or interact with members from other projects, a graph containing all the projects will be analyzed. On the other hand a collapsed graph may bias certain values such as centrality and for this reason, examining the network evolution over time is very important. For example, in case a leader (a node with a large number of connected edges) leaves a project and is replaced with another leader and the centrality level is maintained, in case we collapse the two states of the graph we will obtain a lower overall centrality value. Another advantage of using this approach is that we will get an idea of when and how new members are added to the team, how and when they are eliminated or how the links start to form.

At a later phase of the research it may be possible to use questionnaires, short interviews or other research instruments in order to confirm previous findings such as who are the coordinating members or what are the QA practices within a certain project. Such surveying methods might be used only on a random set of projects or on ones which contain certain ambiguous data.

To answer the questions regarding team dynamics previously acquired data will be processed with Pajek, social network visualization software, after which user clusters (if any), connections, migration patterns will be identified and evolution at different maturity points of the analyzed projects will be tracked. Pajek is a very powerful tool that allows to quickly determine statistical information such as clustering coefficient, distances between nodes and other relevant data. Based on the results obtained at this stage, other tools and algorithms will be identified in order to maximize data potential.

5 Preliminary Analysis

Outlining the current state of QA activities in open source projects is the first step in this research. For that purpose, data availability and QA existence within the FLOSSMetrics data was analyzed by comparing with other possible data sources. In addition, a preliminary case study should provide enough data in order to determine possible issues encountered during the research or flaws in the methods used.

5.1 Data

In order to get a sense of the situation a data dump from FLOSSMetrics [8] containing the mailing lists for 581 open source software projects was downloaded. This dump was first restored on a local machine using a MYSQL database. After performing a simple query in order to identify messages containing the word “QA” and other derivations 46728 messages in 334 projects were found. The next step, which is currently being prepared, is to analyze manually each mailing list in order to determine if there is indeed a quality assurance team, at what stage of maturity of the projects was this decision taken and most importantly how QA is implemented.

It was necessary to check the extent to which the FLOSSMetrics project contains mailing lists associated with 'big names' of the open source movement, in other words successful or popular projects. To perform this verification task, a list of the top 50 most downloaded applications as well as a list containing the top 50 applications ranked by number of users were downloaded from https://www.ohloh.net [9]. Unfortunately, no mailing lists associated with any of these projects is included in the FLOSSMetrics project. Therefore the author will have to attempt to obtain an archive of mailing lists or messages from other communication channels for each of these projects. However, a preliminary verification within the first 50 'popular' applications (ranked by number of users) was

conducted and it was found that at least one third have dedicated quality assurance teams (see Table 1).

Table 1. Top 50 open source software projects ranked by user number (*www.ohloh.net*)

Software Name	QA status
Mozilla Firefox	Yes
Subversion	No - tested before release
Apache HTTP Server	No - community structure contains bug hunter
MySQL	Yes
PHP	Yes
Linux Kernel 2.6	No - tested before release
Firebug	No - use test bots
Bash	No - tested before release
OpenOffice.org	Yes
Ubuntu	Yes
PuTTY	No
GIMP	No - some tasks performed by developers
GNU Compiler Collection	No - some tasks performed by developers
phpMyAdmin	No
Vim	No - some tasks performed by developers
TortoiseSVN	No
GNU grep	No
Thunderbird	Yes
Python programming language	No
VLC media player	Yes
sudo	No
X.Org	Yes
GNU tar	No
Git	No
Eclipse Platform Project	No
OpenSSH	No
GNU Make	No - alpha testing
jQuery	No; Bug triage team
7-Zip	No
GNU Core Utilities	No
GNOME	Yes; Bug squad
Pidgin	No
Wget	No; QA might be performed by third parties
GNU GRUB	No

Software Name	QA status
GNU Screen	No
OpenSSL	No
PostgreSQL Database Server	Yes
Debian GNU/Linux	Yes
FileZilla	No; further analysis needed
CakePHP	No; further analysis needed
rsync	Yes
Trac	No; further analysis needed
Subclipse	No; further analysis needed
WordPress	No; tests performed by SVN and nightly build users
man	No
Tomcat	No
MPlayer	Yes
GNU findutils	No
Inkscape	Yes
bzip2	No

Of course these results are based on a preliminary search which was conducted in order to grasp possible categories and data availability. Further analysis is required for the projects in which a formally defined quality assurance team was not easily identified. For example, it is possible that in certain projects the QA team is defined differently, has a different name or is performed by third parties. Some of the projects taken into consideration have multiple mailing lists associated and quality assurance is defined in a particular way which may bring some difficulties in assessing if the development process contains a QA step or not. Nevertheless, the significant number of major OSS projects now running QA teams supports the basic assumption of this thesis, that we are witnessing the emergence and evolution of QA in the open source software community.

5.2 Case study

The next step in the research was to analyze the mailing lists associated with the Mozilla quality assurance team. The reason why Mozilla was chosen as a first case study is because Mozilla produces one of the best known FLOSS applications (Firefox browser) and has a dedicated quality assurance team since 2006 which provided the right amount of data in order to perform a preliminary study.

Mailing list data was collected in the summer of 2011 and according to the Mozilla Quality Assurance (QMO) website [16], at that time, there were 5 sub teams:

Web QA, Desktop Firefox, Browser Technologies, Automation, Services[1] which each had a dedicated forum. The activity on the forums was low compared to the mailing list activity and for that reason the mailing lists[2] were analyzed as a starting point. Web QA, Desktop Firefox, Browser Technologies, Services teams used the mozilla.dev-quality mailing list while Automation team used the Mozmill developer mailing list.

After downloading the data, in order to obtain results as accurate as possible the data was cleaned by performing the following actions:

- spam was marked as such and removed
- double posts were removed
- a single username was assigned to participants posting with multiple usernames[3]
- authors who did not post in reply to other authors were ignored

After cleaning the data, it was stored in a PostgreSQL database and queries were ran in order to obtain general statistics. As expected, the traffic and number of users is higher on the Mozilla.dev-quality mailing list (Table 2)[4].

Table 2. Mozilla.dev-quality: 2006/17/2-2011/6/30, Mozmill developer 2008/10/1-2011/7/21

	Mozilla.dev-quality	Mozmill developer	Total
Topic	1042	313	1299
Messages	2535	1155	3690
Thread initiators	199	47	233
Distinct authors	293	61	332

If we consider that 5 messages is the lower limit for highly active users then Pareto's principle is somewhat applicable in the sense that only 9.8% of the users post more than 5 messages and 21% of users receive more than 5 replies. Another interesting detail that can be noticed after analyzing the number of messages posted

[1] The structure of the Mozilla Quality Assurance teams is dynamic considering the fact that the Services team was ulteriorly dropped and that since data was collected, other changes have also taken place.

[2] Two mailing lists are at the disposal of the QMO teams: mozilla.dev-quality which contains more general discussion topics and Mozmill developer which contains more technical discussions topics that are mostly about the Mozmill testing tool.

[3] Users that would post only with substrings of the username they would usually use (for example first name instead of full name and so on) were manually checked so it is possible that some participants using multiple usernames were not detected at this phase.

[4] Difference in the total is due to cross posting and users belonging to both lists.

per year (Table 3) and the number of threads started per year (Table 4) is an increase in traffic in the year 2009.

Table 3. Messages posted per year

	Mozilla.dev-quality	Mozmill developer	Total
2006	343	-	343
2007	361	-	361
2008	401	155	556
2009	881	426	1307
2010	411	328	739
2011	138	246	384

Table 4. Threads started per year

	Mozilla.dev-quality	ozmill developer	Total
2006	89	-	89
2007	167	-	167
2008	190	50	238
2009	324	103	415
2010	219	92	282
2011	63	71	121

The next step was to prepare the data for further analysis with Pajek by ignoring authors who replied to their own messages, or in other words eliminating 62 loops and eliminating multiple lines (arcs) between pairs of authors by summing up their values. The resulted directed simple graph contained 301 vertices connected by 1068 arcs.

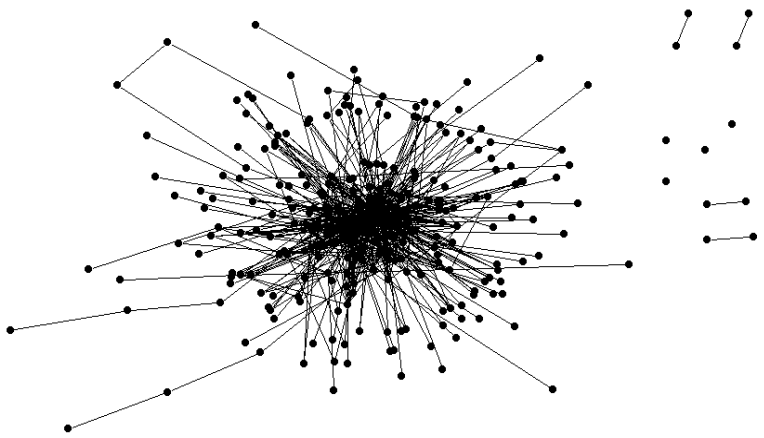

Fig. 2. QMO network

The average degree of the network is 7.09 which means that the average number of connections a participant has is approximately 7. On the other hand, 742 arcs have value 1 while only 326 have a value greater than 1 which means that the majority of participants sent or received only one e-mail or in other words that the majority of links created between distinct pairs of participants were created by sending only one e-mail.

The density of a network is the number of lines expressed as a proportion of the maximum number of lines which means that the greater the density the tighter the network structure is. The density of the QMO network is 0.011 which means that only 1.1% of possible connections between participants are present. The pairs of participants (Table 5) that sent the highest number of e-mails are also the participants with the strongest ties in the network.

Table 5. Arcs with the highest values

Rank	Value	Names
1	53	Skupin ➔Rogers
2	45	Skupin ➔Talbert
3	41	Rogers ➔Skupin

4	34	Skupin ➔Darche
5	29	Darche ➔Skupin
6	27	Talbert ➔Rogers
7	24	Rogers ➔Talbert
8	22	Talbert ➔Skupin
9	18	Christian ➔Rogers
10	15	Desai ➔Skupin

In order to analyze the network's communication structure we need to find the degree centralization of the network which represents the variation in the degrees of vertices divided by the maximum degree variation which a network of the same size could possibly have. In this particular case the all degree centralization is 0.20, input degree centralization is 0.18 and output degree centralization is 0.22. The maximal degree centralization score a network this size could have is 1 while the minimal score is 0. This means that the mailing list participants' network has low degree variation; in other words the gap between vertices with a high number of neighbors and vertices with low number of neighbors is not that large.

While the degree of a vertex is the number of lines incident to it, the indegree of a vertex is the number of arcs it receives and the outdegree is the number of arcs it sends. In this network the highest value for outdegree is 72 and the highest value for indegree is 59, both with an with an arithmetic mean of 3.54.

A semiwalk from vertex u to vertex v is a sequence of lines such that the end vertex of one line is the starting vertex of the next line and the sequence starts at vertex u and ends at vertex v while a walk between vertex u to vertex v is a semiwalk in which none of its lines are an arc of which the end vertex is the arc's tail. Similarly, a semipath is a semiwalk in which between u and v no vertex occurs more than once while a path is a walk that respects the same condition. We can say that a network is weakly connected if each pair of vertices is connected by a semipath and strongly connected if each pair of vertices is connected by a path. A weak component is a maximal connected subnetwork while a strong component is a strongly connected subnetwork. The QMO network contains both strong and weak components as follows:

- 129 strong components where the largest component contains 173 vertices
- 9 weak components where the largest component contains 289 vertices

The distance from the vertices Skupin (the participant with the most links in the network) to all others (k-neighbours) has values between 1 and 5 except for 12 vertices to which there is no connection.

A k-core is a maximal subnetwork in which each vertex has at least degree k within the network and by eliminating the lowest k-cores from a network we can easily detect the existence of cohesive subgroups. By eliminating the lowest 3-cores from the network the subgroup displayed in Figure 2 was obtained.

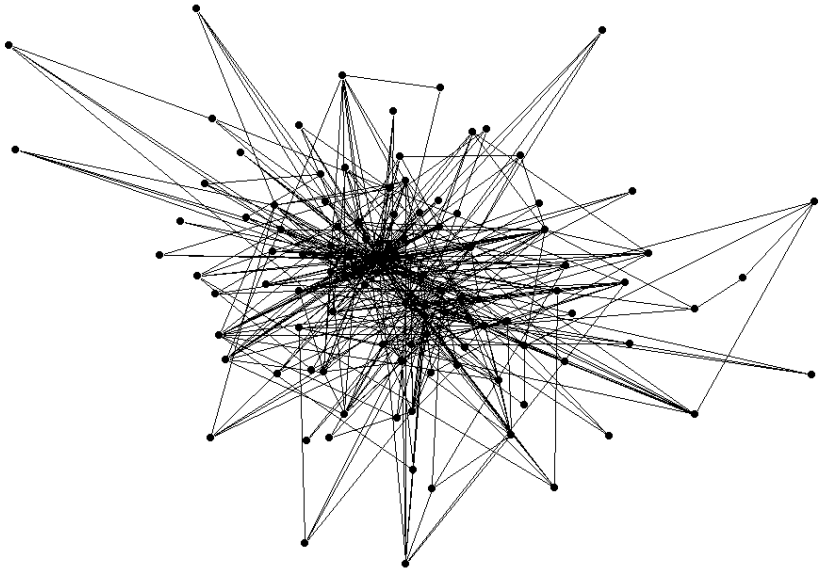

Fig. 3. Lowest 3-core eliminated image

In order to analyze the communication patterns of more active users, a subgraph was extracted by performing the following operations:

- transformed the directed graph to a simple undirected one
- eliminated lines with a value less than 2
- eliminated isolated vertices

So basically people without at least one reciprocal connection in the network or people that communicated with an another participant by sending/receiving less than 2 e-mails were eliminated. The resulting graph has 164 vertices and 345 lines with a value greater than 1. Density (no loops allowed) is 0.0258 while the average degree is 4.207.

A clique is a maximal complete subnetwork containing three vertices or more while complete triads are complete subnetworks consisting or three vertices. These triads represent a strict definition of a cohesive unit, a basic measurement unit. The total number of triads contained by the active users' graph is 238 while the highest number of triads to which a vertex belongs to is 77 while the lowest is 0. Vertices that belong to at least one triad represent approximately 50% of the total number which means that half of the active participants belong to at least one cohesive group.

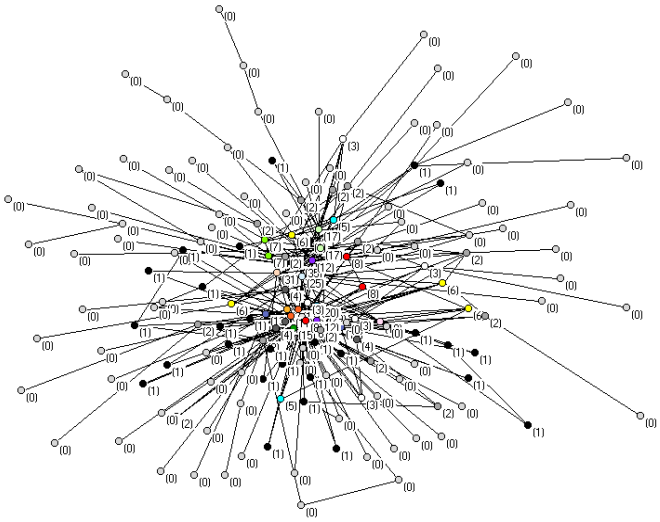

Fig. 4. Triads

A geodesic is the shortest path between two vertices. The betweenness centrality of a vertex is the proportion of all geodesics between pairs of other vertices that include this vertex while the betweenness centrality score is the variation in the betweenness of the vertices divided by the maximum variation in a network of the same size. The active users' graph has a betweenness centralization score of 0.354 while the betweenness centrality values range from 0 to 0.366 with 54.268% of the vertices with a betweenness score equal to 0. This means that over 50% of the participants are not well connected with other participants and they do not influence the information flow.

The distance from u to v is the length of the geodesic from u to v. The closeness centrality of a vertex is the number of other vertices divided by the sum of all distances between the vertex and all others while the closeness centralization score is the variation in the closeness centrality of all vertices divided by the maximum variation in a network of the same size. The active users' graph has a closeness centralization score of 0.386 while the closeness centrality values range from 0.192 to 0.517 with 68.2927% of the vertices with a centrality over 0.301 and 6.0976% with closeness centrality over 0.409.

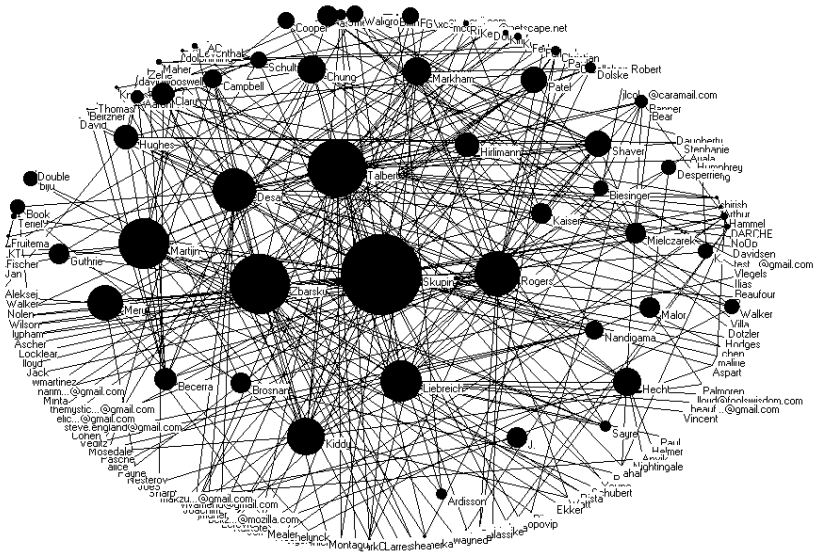

Fig. 5. Central figures

A bridge is a line whose removal increases the number of components in the network while a cut-vertex is a vertex whose deletion increases the number of

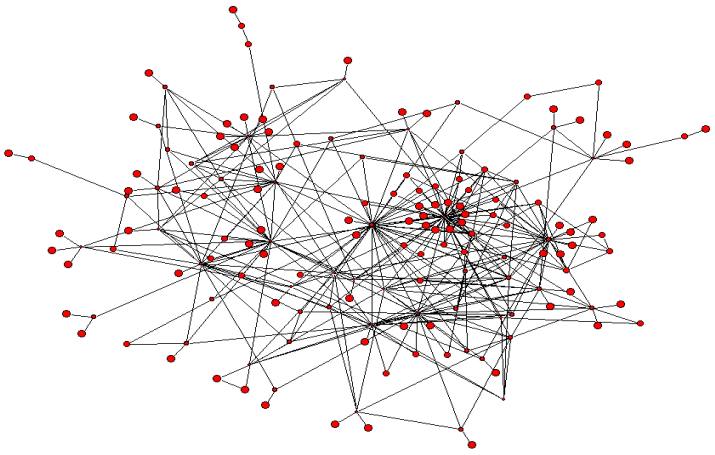

Fig. 6. Energized constraint

components in the network. A bi-component is component of minim size 3 that does not contain a cut vertex. The active users' graph contains 73 bi-components from which 71 are bridges. The remaining two bi-components have a size of 3 and 91.

The dyadic constraint on vertex u exercised by a tie between vertices u and v is the extent to which u has more and stronger ties with neighbors who are strongly connected with vertex v while the aggregate constraint is the sum of the dyadic constraint on all ties of a certain person. The aggregate constraint has values between 0.098 and 1.085.

If a relation between two participants is created when they exchange an e-mail and that relation doesn't deteriorate time then the network evolution can be visualized. For that purpose the timeline was split into 6 month periods and the correspondent graph was generated for each period.

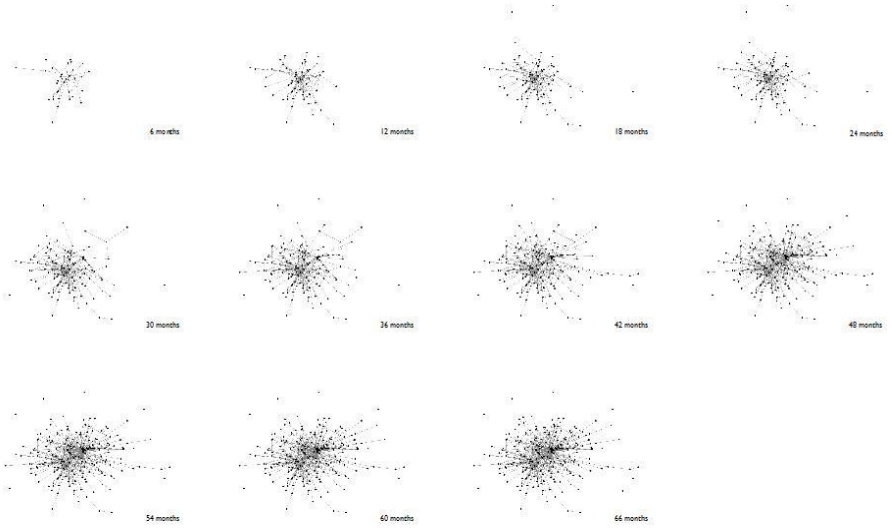

Fig. 7. Network evolution

If, on the other hand, relations can be deteriorated over time and users become inactive in certain time frames then the network evolution looks completely different. In return, it is much easier to observe the network's state in a certain moment in time and compare it with previous steps.

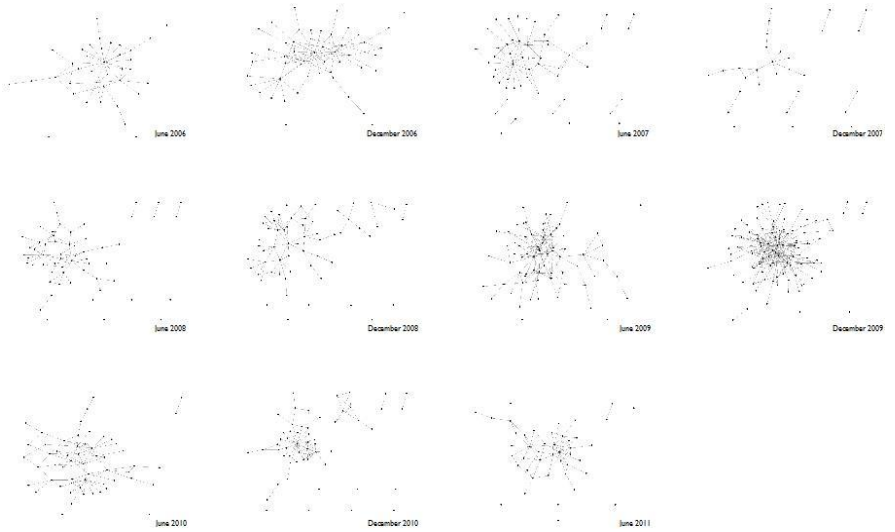

Fig. 8. Network state

6 Conclusions and issues

Based solely on the analysis performed so far on the mailing lists used by the Mozilla quality assurance teams it can be concluded that user activity is not linked to time progression but there are fluctuations which means that there are other variables that must be found. There are basically no small groups of individuals working together and participants from all the teams form a large group of 163 active users spanning both mailing lists. 6.0976% of the individuals forming the active users' graph have a high closeness centrality score (0.409) which indicates that a small number of users represent the center of the network.

Regarding the Mozilla case study, the next phase is to attribute positions to key users. For that purpose data available from code repositories and issue trackers will be analyzed and in addition participants will be categorized. For example, if a user is active on the mailing lists and issue tracker but has never submitted code then we can assume he is not a developer, on the other hand, if a user is always committing code to various projects (except Mozmill) then we can assume he is developer. Of course, this step needs to be performed carefully because this participant categorization will be used as a starting point for the research. Also, by analyzing the data available on other channels, users migration between teams can be tracked by measuring their activity levels in certain time frames.

To answer at what stage of the Mozilla projects' evolution was the QA team officially formed it is necessary to analyze the version history and analyze data from websites, tops, and blogs about the maturity level at that point.

Of course, the steps performed for the test case should be performed for other OSS projects. For this purpose, the next phase consists of retrieving projects featured on the www.ohloh.net website and cross referencing them with projects that have associated mailing lists and are included in the FLOSSMetrics database. Depending on the results, any relevant communication channels will be analyzed and the available archives will be added to the database.

When analyzing data, if a group of people can be clearly identified as performing QA tasks, then it will assumed that QA is present. It is important that the communication channels contain data that allow users to be uniquely identifiable in order to measure the level of activity within the project, user migration from one project to another and fluctuations in team size, in order to depict as accurately as possible the dynamics of this group. A huge challenge will be identifying users that migrated from one project to another considering that users may not use the same e-mail address or that some might not divulge their real identity in which case the only solution will be finding actual references to the users' past project involvement.

At this point, it is important to categorize projects in order to identify links between the existence of a formal quality assurance team and type for example. During the process of identifying individuals performing quality assurance tasks and data cleaning, data should be slowly added to Pajek in order to gather preliminary network statistical data as soon as possible and identify possible further processes that are needed to obtain unbiased results.

Another issue is represented by committers and core developers who are also performing quality assurance tasks. It is important to identify these projects and include them in a separate category and follow their evolution as they might be in a transitional state. Of course, this step will also be achieved while analyzing the message exchange associated with different projects. At this stage, it is still unsure if this case should be included in the research as a special category due to the fact that the actual number might be insignificant.

From the preliminary assessment it was concluded that separating quality assurance performed into different categories is necessary in order to get a clear picture of what activities each category covers. Some quality assurance teams write official documents such as test cases and test plans whilst others just test the application or help with the issue tracker and it is important to make a clear difference between them. For example, users that only help with triaging issue tracker bugs can't be considered as performing quality assurance, which means that they will be included in a separate category.

References

[1] Raymond, E.S. (1999). The cathedral and the bazaar, O'Reilly.

[2] DiBona C., Cooper D., Cooper M. (2006), Open Sources 2.0: The Continuing Evolution, O'Reilly, USA.

[3] Crowston K., Howison, J. (2004), The social structure of Free and Open Source software.

[4] Spinellis D., Gousios G., Karakoidas V., Louridas P., Adams P. J, Samoladas I., Stamelos I. (2009), Evaluating the Quality of Open Source Software, Electronic Notes in Theoretical Computer Science, Volume 233, Proceedings of the International Workshop on Software Quality and Maintainability (SQM 2008).

[5] Mockus A., Fielding R. T, and Herbsleb J. D (2002), "Two Case Studies Of Open Source Software Development: Apache And Mozilla," ACM Transactions on Software Engineering and Methodology, volume 11, number 3, pp. 309–346.

[6] Lee S. T., Kim H., Gupta S. (2009), Measuring open source software success, Omega, Volume 37, Issue 2, April.

[7] David Paul A., Waterman A, Arora S., FLOSS-US – The free/libre/open source software survey for, (http://www.stanford.edu/group/floss-us)

[8] http://www.flossmetrics.org/

[9] https://www.ohloh.net/

[10] http://www.iso.org/iso/catalogue_detail.htm?csnumber=22749

[11] http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=35733

[12] http://www.opensource.org/

[13] http://oss2011.org/

[14] http://www.qualipso.org/

[15] Barcellini F., Détienne F., Burkhardt J-M, Sack W (2005), Thematic coherence and quotation practices in OSS design-oriented online discussions. Proceedings of the 2005 international ACM SIGGROUP conference on Supporting group work

[16] https://quality.mozilla.org/

[17] Oezbek C., Prechelt, L. & Thiel F. (2010). The Onion has Cancer: Some Social Network Analysis Visualizations of Open Source Project Communication. *Psychology*, (Section 4), 4-9. FLOSS'10

[18] Masmoudi H., den Besten M. L., De Loupy C. and Dalle J. M. (2009), 'Peeling the Onion': The Words and Actions that Distinguish Core from Periphery in Bug Reports and How Core and Periphery Interact Together . Fifth International Conference on Open Source Systems - OSS 2009.

Essential Properties of Open Development Communities

Terhi Kilamo[1]

Tampere University of Technology terhi.kilamo@tut.fi

Abstract. Open development is reaching beyond the scope of open source. At the same time open source software is gaining ground not only as a development method but as a business model in the software industry. What lies at the heart of development is a developer community; a heterogenous, freely formed group of people working on different aspects of their joined project. A sustainable product is dependent on a functional community, especially if business success relies on it. The focus of this research is to find answers to what makes it possible for the community to form and grow – to identify the properties that are key at growing an active and sustainable community.

Key words: open development, open source software, sustainability, development communities

1 Introduction

Open source is continuously gaining ground both as a development platform and as a business model. An increasing amount of companies such as Google have their software products available as open source. In addition, companies such as Nokia, are also releasing their formerly proprietary software to gain benefit from open development. This means that an increasing amount of development communities are there to attract participants – where do they come from and what makes them stay?

At the same time, open development communities have reached beyond software development into new areas. One such area is innovation where open innovation environments are providing improved ways for companies to test their ideas and forge ideas into prototypes in a fast pace and with a versatile, enthusiastic group of people from a wider range of expertise than what is available within the company [4]. Even entertainment content ranging from music videos to feature films[1] and commodities such cars[2] are developed in open development communities.

The development is heavily dependent on the developing community. Especially in the early stages of open development crucial decision that directly affect

[1] http://www.wreckamovie.com/
[2] http://www.sahkoautot.fi/eng

the future of the community need to be made. Open development is a multi-faceted challenge ranging over aspects such as governance, legality, sustainability in addition to the the development field itself. What is needed is framework for leveraging success for new communities and following their progress. There is a need for identifying the actions and properties that indicate likelyhood for the community to succeed.

This paper describes the current plan for the resarch intended for the author's doctoral thesis. The aim is to investigate the essential elements of viable open development communities addressing the challenge from each of its key angles. The thesis research also sets out to indentify elements that are central for gaining success and in avoiding tragedy while there is no guarantee for fame and glory.

The rest of the paper is structured as follows. Section 2 gives the background for the research discussed in this paper. In Section 3 the research is motivated and Section 4 discusses the research design. Section 5 gives an overview of the completed parts of the research topic and finally the paper is concluded in Section 6.

2 Background

Open source software is developed by a community of stakeholders: developers, users, business partners and other individuals. The community behind the software is a key component that affects the success of the project. The community structure is often modeled with an onion model introduced in [6]. In the model each member of the community is assigned with a distinct role. The community is viewed to have an onion-like structure, where the most involved and thus most influencial community members occupy the core layers, while the outer layers hold the less active ones. The onion structure supports the view taken into the open development communities in our research also.

Ever though the community is at the core of open source, it is not just a structured community. Even the development community has organizational issues, for example in decision making, release planning, acceptance of contributions and so forth, that need attention. There is a need for governance practices. The community needs a technological infrastructure for communication, planning, coordination and maintenance of the product. Legal issues need to be considered on the lines of intellectual property rights (who wrote which parts of the software for example), copyright issues and possible licensing schemes. The product itself needs to suit open community driven development. Finally, there are business decisions that effect the decision making. These facets influence each other and cannot be viewed completely separately from each other. However, each have several possible directions and angles to them.

Some studies on open source sustainability have been done. There are frameworks for assessing the open source projects proposed [7, 3, 1] but their focus is from the point of view of adoption and they provide processes of several steps

to evaluate and select viable projects out of seemingly suitable ones. There are studies that identify success and tragedy after the fact [8, 5]. What we are interested in are the dos and donts that may determine the success or tragedy and essentially make the likelyhood of success greater.

3 Motivation for Research

Open source has marched to the forefront of practical software development and software business models in the recent years. In addition different flavours of open source development are now applied and can be identified in several other fields as well. Open source, defined by a set of principles, practices and development culture, consists of a range of aspects that must be addressed when establishing a community. These aspects are identifiable not only in open source, but in open development communities at large. Our main focus is software development and hence open source communities but a wider view on open development as a whole is also taken.

Despite the apparent opportunities open development offers, the generic ability of open source to act as the basis for development, business and systems has, however, gained only some research interest. Moreover, since the trend to release proprietary software as open source is relatively recent, there are few guidelines on how to create and maintain a sustainable open source community [2]. Consequently, there is little evidence on how the different facets of an open source platform should be taken into account.

4 Research Approach

The doctoral research focuses on the growing and developing open development communities. How communities are born and how this process can be supported before and after going open are studied. The main focus is software development but the research aims at indentifying success factors for development communities in general.

4.1 Research Questions

The aim of the research is to answer the following questions:

Q1 What issues must be addressed in order to grow an open development community?

Q2 How these are distributed according to different aspects of open development?

These can be divided into subquestions in order to see the different things to investigate under both of them. The first question (Q1) can be broken into the following:

Q1.1 What properties indicate success?
Q1.2 What are the possible warning signs for tragedy?
Q1.3 How do the different actions affect the development community?

The second question (Q2) must further address issues such as:

Q2.1 What role does the product play?
Q2.2 How should the legal aspects be handled?

The goal of the research conducted in this work is not to give a checklist for success. On the contrary the initial claim is that there is no paved road to immortality. However, certain set of aspects are essential in growing a development community that can be viable and the research aim is to identify these.

Characteristics of open development communities that range over several dimensions and which are depicted in Figure 1 are investigated. These are governance of open development, open development community organization, possible business aspects, legal matters, technological infrastructure, software or other product developed by the community and sustainability of the community. These are overlapping and interleaving and cannot always be separately addressed. It is however apparent that a viable development community needs to give consideration to each. The goal is to form a framework to address the essential properties of open development on the lines of each of the dimensions.

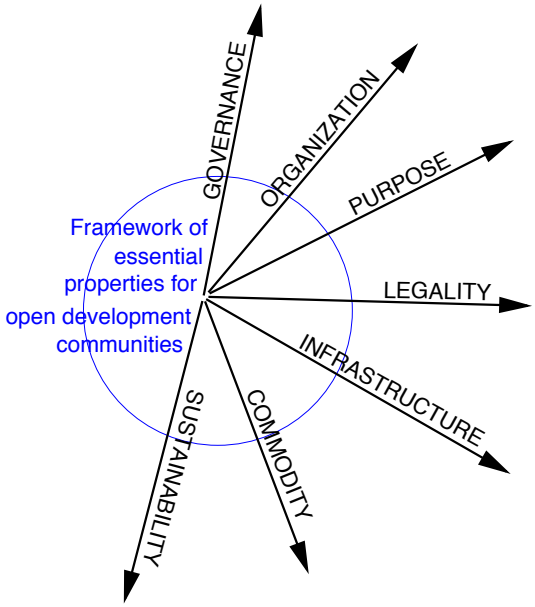

Fig. 1. Essential Community Properties

4.2 Research Method

A case study based approach is taken in the research. The research consists of three community cases that are investigated both through a constructive approach as well an analytic viewpoint. The constructive part relies on work started in [12] where a model for analyzing the progress of a community based on a set of measures is introduced. The analytic part focuses on evaluating practical cases from industrial partners with open development communities of different ages. The research may be somewhat hindered by failure in one of the communities. However, the risk is small. One open community focusing on open innovation provides a case for analysis of methods and practises. Additionally it provides a large data set to study. In total there are three separate community types studied through cases shown in Figure 2 with open education complementing the two more industrially driven ones. The completed research on the case areas is discussed further in Section 5

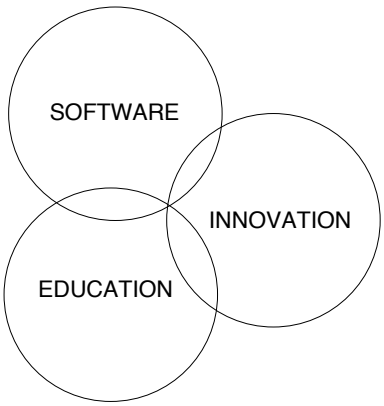

Fig. 2. Communities Studied

5 Completed Activities

The thesis work is now close to two years in running. Some parts of the research lean on work done earlier in the TUTopen research group and to work done in collaboration with the group. In addition the author has worked on combining the work and focusing on studying different types of settings for open development. Currently three different kinds of open communities – software business, innovation, educational – have been studied. All the research and publications so far lay a foundation for the thesis work but there is still a need to complete the picture with further research. A mostly unanswered question is the role

of the type of the community, what role does the product or commodity produced play. More work on voluntary based communities is needed. Completely voluntary based communities have so far not been studied at all.

Open business ecosystems The main focus so far and thus also the work furthest along is concerning open source software business and open software ecosystems. There are two published articles, one on readiness for going open source [11] and one on following the progress of the community in numbers [12].

The first paper introduces a framework for indentifying possible bottlenecks and places for improvement when planning a release of a proprietary software product as open source. The paper explores the research questions:

- What kind of evaluation criteria could be used to assess software readiness for open source development?
- How the evaluation should be planned and which stakeholders are involved?
- How to obtain data for the evaluation process?
- How to exploit the results of the evaluation process?

The second paper in turn focuses on monitoring the evolution of the community by a continuous measurement of a selected set of key data sources answering the question: how can a large set of community data be utilized over time to support decision making. The data sources are specific to the community of interest. The model distributes the gathered data over the layers of the onion to give information of the amount or activity of the community stakeholders. The idea is to give a time dependent, continuous and easy method of following the state of the community and to aid decision making especially when business aspects and the community are concerned.

One minor article has been published in collaboration with a main author. The paper discusses evaluation of open source communities and how welcoming they are [9] for new community members. The paper addresses the issue directly by reporting the participatory effort of one developer in getting more involved and gaining bigger responsibilities in an open source software community that has business importance.

There is a key article that combines the work done within the research group on releasing a proprietary software product which is about to be published in the Journal of Systems and Software. It provides a foundation for the thesis work and is thus a major publication for the dissertation. The article brings together the different stages of releasing software and discusses supporting processes, guidelines and best practices for stages prior to the release, for preparing the release and following the progress of the development community directly after the release. It gives a good foundation for the work intended for this thesis.

Open innovation communities One open development community studied as we speak is an open innovation environment working in the Pirkanmaa region and in collaboration with the niversities and polytechnic in the area. So far one paper has been written on the open innovation community and it was presented at OSS2011 conference. The paper focuses on identifying open source

best practices as an essential part of a successful open innovation environment. At the conference workshop an initial position paper on how the shortcomings of the open source approach can be overcome through a set of practices known from agile software development and self-controlling teams.

Open education University students and free learners form a open development community in their studies when working on suitable course projects. This provides a case to study communities from an open education perspective. Some research focus has been given to how open source development practices can be taught in such a community setting [10]. Even though this is not directly related to the thesis work described in this paper, it plays a major supporting role. A paper on open and collaborative online learning environment was presented in OSS2011. The work there acts as a basis for the open education case.

6 Conclusions

Open source provides a viable platform for development that addresses not only technical but economical, social and legal aspects as well. It provides development methodology, infrastructure and environment. It addresses legal issues such as IPR and acts as a business model. Open source can even be seen as a clever marketing strategy. All these aspects have inherent threats to them. A simple overlooked issue may have far reaching consequences. This research aims to look into the different facets of open development and form a framework of essential properties of them that enforce the possibilty of future success and growth.

References

1. Golden B. *Succeeding with Open Source*. Addison-Wesley, 2004.
2. Lundell B., Forssten B., Gamalielsson J., Gustavsson H., Karlsson R., Lennerholt C., Lings B., Mattsson A., and Olsson E. Exploring health within oss ecosystems. In *In Proceedings of OSCOMM 2009*, Sweden, June 2009.
3. BRR. `http://www.openbrr.org/`. Last visited March 2009.
4. Chesbrough H. *Open Innovation: Researching a New Paradigm*, chapter Open Innovation: A New Paradigm for Understanding Industrial Innovation. Oxford University Press, 2006.
5. Crowston K., Annabi H., and Howison J. Defining open source software project success. In *in Proceedings of the 24th International Conference on Information Systems (ICIS 2003*, pages 327–340, 2003.
6. Nakakoji K., Yamamoto Y., Nishinaka Y., Kishida K., and Ye Y. Evolution Pattern of Open-Source Software Systems and Communities. In *IWPSE '02: Proceedings of the International Workshop on Principles of Software Evolution (2002)*, pages 76–85. ACM Press, 2002.
7. QSOS. `http://www.qsos.org/`. Last visited March 2009.

8. English R. and Schweik C.M. Identifying success and tragedy of free/libre and open source (floss) commons: A preliminary classification of sourceforge.net projects. In *Proceedings of the First International Workshop on Emerging Trends in FLOSS Research and Development (FLOSS'07: ICSE Workshops 2007)*, May 2007.
9. Mikkonen S., Kilamo T., and Mikkonen T. My summer as a mole: Evaluating an open source community via participation. In *In proceedings of OSW 2009*, October 2009. Open Source Workshop 2009 in conjunction with the 4th IEEE Systems and Software Week.
10. Kilamo T. The community game: Learning open source development through participatory exercise. In *In Proceedings of Academic Mindtrek 2010*. ACM Press, October 2010.
11. Kilamo T., Aaltonen T., Hammouda I., Heinimäki T.J., and Mikkonen T. Evaluating the Readiness of Proprietary Software for Open Source Development. In *OSS2010*, volume 319 of *IFIP Advances in Information and Communication Technology*, pages 143–155. Springer, 2010.
12. Kilamo T., Aaltonen T., and Heinimäki T.J. BULB: Onion-Based Measuring of OSS Communities. In *OSS2010*, number 319 in IFIP Advances in Information and Communication Technology. Springer, 2010.

Open Source: From Mythos to Meaning

Alexander C. MacLean and Charles D. Knutson

Computer Science Department, Brigham Young University, Provo, Utah
amaclean@byu.edu, knutson@cs.byu.edu

Abstract. Free open source software (FOSS) projects expose rich development, evolutionary, and collaborative data from which researchers have formed theories and conclusions about the FOSS development ecosystem. However, little work has been done to determine whether FOSS projects are analogous to proprietary development efforts. We propose several axes along which taxonomies of FOSS and proprietary projects may be created and compared, and preview several future studies that will begin to populate these taxonomies.

1 What is "Open Source"?

Open Source proponents have long touted the advantages, be they financial, social, ethical, or moral, of developing software using an "open source" paradigm. However, even among some of the earliest players in the "open source" space, there is little agreement about its exact definition. Some claim that open source is a convenient (and possibly more efficient) way for developers to collaborate on major software projects.

> Linus Torvalds: *Me, I just don't care about proprietary software. It's not "evil" or "immoral," it just doesn't matter. I think that Open Source can do better... it's just a superior way of working together and generating code.*
> *It's superior because it's a lot more fun and because it makes cooperation much easier (no silly NDA's or artificial barriers to innovation like in a proprietary setting)...* [24]

Others elevate the paradigm to pseudo-religious stature.

> Richard Stallman: *We like to think that our society encourages helping your neighbor; but each time we reward someone for obstructionism [not sharing code], or admire them for the wealth they have gained in this way, we are sending the opposite message* [21].

Still others herald the potential business advantages.

> Eric Raymond: *[Open Source is] the process of systematically harnessing open development and decentralized peer review to lower costs and improve software quality* [19].

> Bob Young: *[Open Source] gives customers control over the technologies they use, instead of enabling the vendors to control their customers through restricting access to the code behind the technologies* [19].

Others are confused by what we're even talking about in the first place.

> Inigo Montoya: *You keep using that word. I do not think it means what you think it means.*

It is clear that open source has changed the landscape of software development. But what do we, as researchers, mean when we refer to "open source"? You may notice that the preceding quotations aren't actually definitions, but rather descriptions of the attributes or benefits of an "open source" product or organization. The problem is that when we conduct research, loose definitions and vague conceptual notions aren't good enough.

To illustrate the point, here are several definitions of "open source" we have heard and/or used while discussing the open source movement with other researchers:

1. A licensing model that requires redistribution of code along with a product.
2. A mythos concerning the operation and constitution of open source communities that encourage volunteer participation by developers.
3. A mythology concerning the origins of open source that contrasts open source operations with traditional, closed source operations.
4. A convenient licensing model that allows organizations to collaborate on infrastructure (plumbing, if you will) while differentiating themselves in higher level software.
5. A direct attack on "the man" and the closed source restrictiveness of imperialist software companies.
6. A marketing ploy by large corporations to engender good will with customers.
7. Any number of other definitions, depending upon the circumstance and audience.

In each case (with the possible exception of Item 5), the definition is sufficient for the particular case to which it is applied. However, none of these is broad enough to capture the many variations of the open source paradigm.

A taxonomy that considers open source development organizations, as well as the developers and organizations that contribute to open source projects, would allow researchers to qualify results within the confines of the taxonomy of the organization from which they were drawn, rather than from an arbitrarily broad set of ambiguous definitions. Moreover, a taxonomy would ground research findings within a common theoretical framework and provide a mechanism for determining the degree to which such results can be extrapolated to other projects and organizations (whether open or closed source).

2 Open Questions

For the moment, let us ignore the vague definitions and distinctions of open and closed source. Instead, let's start with a broad definition of open source software that simply requires that the source code is available to the end user... eventually[1], and refer to this as Free and Open Source Software (FOSS). This definition only distinguishes between organizations that restrict access to their source code and those that don't.

Definitions:

FOSS: Software for which the source code is eventually available to the user.
Proprietary Software: Any software that is not FOSS.

Although some work has been done to explore the behaviors of open and closed sourced development organizations, little effort has been expended to understand the differences between the developers in such organizations. Many questions persist:

1. Who are the developers who spend their time working on these projects?
2. Which organizations employ open source developers, and what are their motivations? Although studies have examined this question, more work is required to build a taxonomy of the results.
3. Are open source developers somehow different from those that work on closed closed projects? More formally, what is the taxonomy of developers who choose to (or are employed to) work on open source projects?
4. Is the taxonomy of open source developers significantly different than the taxonomy of the general population of software developers? If so, how? Along what axes are these two taxonomies analogous?
5. What does it mean if open source developers are not significantly different from the general population of developers (along certain axes)?
6. What does it mean if open source developers *are* significantly different from the general population developers (along certain axes)?

In the following subsections, we address some questions and posit theories that arise from these potential lines of inquiries. In Section 3 we propose methods for answering some of these questions.

2.1 The FOSS Developer

Who are these developers that work on FOSS projects? Bird, et al. report that much of Eclipse is written in-house at IBM. On the other hand, "[c]ontributions

[1] The "... eventually" clause in this definition is required because the publicly available trunk for Android, a major open source project, is typically several months out of date.

to Firefox come from a myriad of sources and no single commercial entity completely controls or owns the development process" [1]. A more complete understanding of the developer taxonomy would allow researchers to design better studies and draw more accurate conclusions about the state of FOSS and proprietary development organizations and practices. In this section we list several axes of a potential taxonomy of FOSS developers.

Developer Motivation Lakhani and Wolf, in a study of 684 software developers in 287 FOSS projects, found that the key factor in developer participation was "enjoyment-based intrinsic motivation." However, they did not take into account the power law distribution of developer contributions which describes a common phenomenon in FOSS projects: a small (relative to the project size) set of core developers often develop most of the functionality (see Figure 1). Many questions remain regarding the core developers within FOSS organizations. For example, are the central figures within these projects paid, while the ancillaries are motivated by a desire for creative outlet?

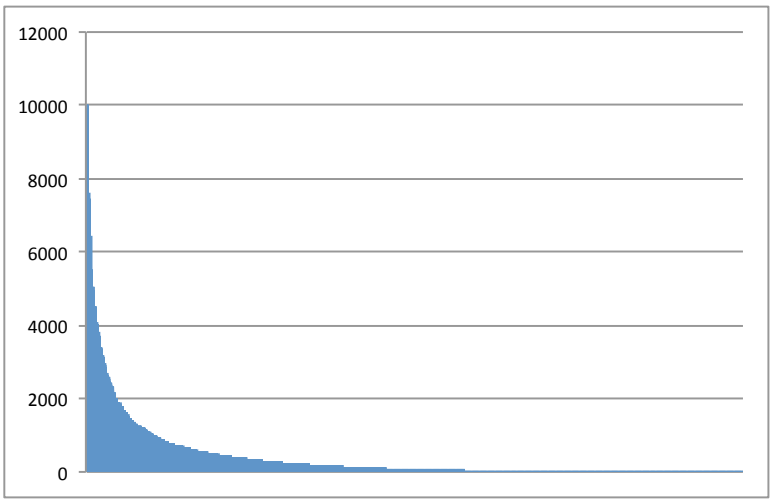

Fig. 1. Developer commit volume on the Apache HTTP Server project. The x-axis is individual developers, sorted by commit volume.

Job Tenure During the dot com boom, conventional wisdom held that a developer remained in a job for 18 months. By 2003, three years after the bubble burst, IT workers who earned Computer Science degrees had an average job tenure of 6.2 years [5]. In contrast, the median tenure of developers on the Apache HTTP Server project is 3.7 years[2], with a strong right tail (see

[2] Tenure in this case is defined as the length of time between a developer's first and last commit. Potential issues with this definitions are addressed in Section 3.

Figure 2), just more than half the tenure of the general developer population. However, the top 15 committers to the Apache HTTP Server project have a median tenure of just under 8 years.

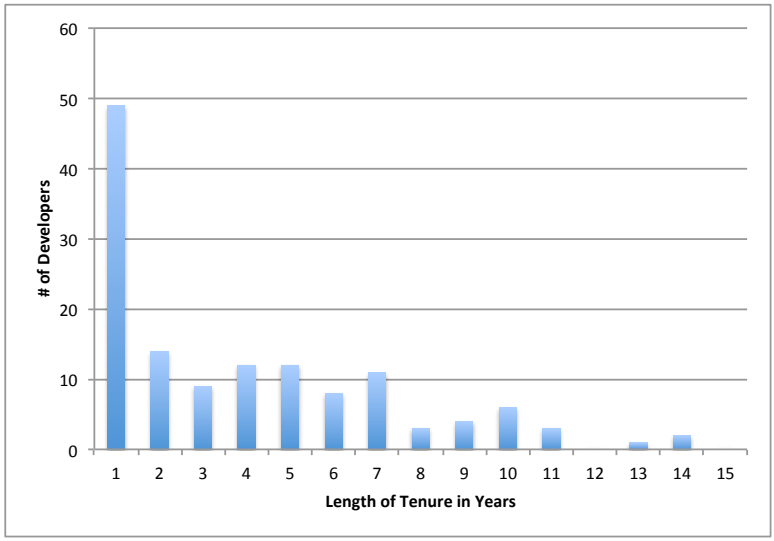

Fig. 2. Tenure of developers on the Apache HTTP Server project.

Job tenure is illustrative of expertise and seniority in a development project. Tenure could vary based upon differences between an open source meritocracy and traditional, corporate approaches.

Developer Background (Work Related) Two developer characteristics related to job tenure—age distribution and industrial tenure—reveal some measure of industrial, commercial, and organizational expertise. Other attributes, such as education, illustrate the ability of an organization to attract top talent. With respect to age distribution and industrial tenure, a major question arises: is open source development dominated by youthful exuberance or aged wisdom? Or is it composed of a healthy amalgam? We propose three theories based upon the potential answers to the preceding questions.

Ghosh, et al. found that in 2002 the median age of FOSS developers was 26 and that only 10% were older than 35 [7]). If youth typifies the development environment, we theorize that it reflects two attributes of the open source economy:

1. Organizations that support FOSS projects view the projects as plumbing, not an area in which to differentiate themselves, and therefore allocate less-experienced developers, and/or
2. Outside of regular employment, younger developers are more likely to have the free time required to make meaningful contributions to FOSS projects.

Lakhani and Wolf found that "formal IT training... reduces the number of hours spent on a project" [11]. In their study, this result was incidental, and therefore received little attention. Nevertheless, it leads to questions related to the role of industrial experience in FOSS contribution and the shape of the FOSS developer taxonomy. However, the aforementioned age distribution does not consider developer prominence or role in an organization. On the other hand, if FOSS projects, or more importantly, leadership within FOSS projects, are the purview of elder hackers, we theorize contrasting economic attributes:

1. Organizations depend upon the success of a FOSS project in order to differentiate their services or products, and therefore assign high priority to its development, and/or
2. Older, more experienced, developers find time to contribute, even when not compensated.

Of course, neither of these two polarized theories is likely to be found entirely applicable in all projects. Instead, a mixture may exist where, for example, organizations tend to employ older developers, while younger developers tend to have the flexibility to contribute code on their own time. Whatever the distribution across developer age and experience, a more clear understanding would provide insight into the motivation and monetization structures of companies that contribute to open source projects, as well as to the experience level of the developers.

If the distribution of developer age and experience does not, in fact, gravitate towards one of these poles, it could indicate that leaders of these organizations don't discriminate for or against their FOSS collaborations, but instead view them as parallel and complementary to their other development efforts.

Developer Background (Non-Work Related) Gender, race, and other personal attributes are discoverable in a typical development organization. In fact, even in proprietary, distributed development organizations, these attributes are known because hiring and promotion interviews are still performed face-to-face. On the other hand, unless explicitly exposed by the developer, these attributes are largely latent in meritocratic organizations such as the Apache Foundation. Other attributes, such as socio-economic background, familial status (single, married, married with children, single parent, etc.), and religion, are often latent in both types of organizations.

A taxonomy and understanding of the personal attributes of developers within a meritocratic FOSS development organization and within proprietary organizations would allow us to analyze whether the latent nature of these attributes protects FOSS projects from flexible definitions of merit [25] and other types of discrimination. A positive result could suggest a fascinating feature of distributed, FOSS development: insulation from discriminatory practices, both intended and unintended [26].

Specialization and Private Information Previous work indicates both the presence of private information[3] in large organizations [10] and a lack of specialization (a subtype of private information) in the Apache HTTP Server project [13]. Lack of private information and specialization in open source development organizations contrasts starkly with many proprietary environments where specialization is often associated with efficiency and job security. If this contrast holds across open development organizations as a whole, it represents a fundamental difference in both communication requirements and organizational behavior.

Programming Languages Although not a core developer attribute, differences in programming language fragmentation, or the degree to which a developer utilizes more than one programming language, would illustrate differences between imposed technical environments and less-structured, self-organizing, distributed communities [8, 9]. We suspect that along this axis, FOSS and proprietary *organizations* will appear similar (organizations must standardize upon some set of tools and languages in order to be productive, whether FOSS or proprietary). However, FOSS developers who work on multiple projects may have higher language fragmentation than proprietary developers who only work on a single project.

Salary Ghosh, et al. note that 52% of FOSS developers (in 2002) earned no more than 2,000 U.S. Dollars or Euros per month [7]. In contrast, Choy et al. reported that the average monthly income for all U.S. Computer Science graduates 10 years after graduation[4] (in 2003) was 6,050 U.S. dollars [5]. This disparity is exaggerated by a high level of student contributions to FOSS projects: 17%. Also, only 14% of respondents live in North America, so it would be unwise to make direct comparisons between the two studies. However, the large difference certainly suggests inequalities in our open and closed source taxonomies.

Or does it? Ghosh, et al. don't take developer prominence or position into account. Although many of the developers on the Apache HTTP Server project are most likely students, it does not follow that the most or any of the 15 core developers are as well [16]. Salaries for core FOSS developers may be on par with their proprietary counterparts. If so, this seeming discrepancy fades in the light of taxonomic clarity.

2.2 Contributing Organizations

Many organizations that build proprietary products also contribute to FOSS projects [4].

[3] We utilize a definition of *private information* as "...the challenge of utilizing distributed knowledge in an organization...where *private* refers to information possessed by a relatively small segment of the population—as opposed to information that is widely held" [10].

[4] According to Ghosh, et al., the median age of FOSS developers is 26 [7], close to the average age of computer science graduates 10 years after graduation.

Motivations Lerner and Tirole present three motivations for organizations to expend resources on products that don't directly generate revenue: 1) Living symbiotically off an open source project (Red Hat, Sun, and Oracle); 2) Code release to benefit a complementary market segment (Hewlett-Packard and IBM); and 3) Acting as intermediaries (Collabet.net) [12]. In essence, the existence of the product produces side effects that positively affect the bottom line of the company.

Capek, et al., in explaining the genesis and integration of open source ideals at IBM, stated "... open source did not pose an immediate threat to our existing business, and in fact, our products could benefit from supporting and building open source." They note that participating in open source projects yielded two key benefits (for IBM) [4]:

1. Reusing open source components decreased overhead versus building proprietary solutions. Example: using the Apache HTTP Server as a key component of the WebSphere Application Server.
2. Collaborating with others in the community to develop necessary but low margin tools frees up resources for more lucrative projects. Example: Eclipse.

In 1996, Apple acquired NeXT in an effort to modernize its aging operating system and salvage its dwindling market share. The company then began using NeXT's BSD-based Unix variant, NextStep, as the base for its operating system. It released the core components of its operating system as a BSD-licensed FOSS project named Darwin, but kept the GUI and many of the APIs (including the Java API) proprietary. This "layered" open-closed approach allowed Apple to reap the benefits of using a proven FOSS technology while maintaining control over many of the distinguishing components of its operating system [28].

Additionally, Sun slowly moved into FOSS in an attempt to stymie advances by Microsoft and Linux [28].

Organizational Attributes As with developers, understanding the taxonomical attributes of organizations that contribute to FOSS ventures—such as size, revenue, location, and industry—would allow researchers to draw parallels to the general population of organizations. In addition, such a taxonomy would provide a standard against which to measure claims that attempt to extrapolate results from open source projects.

2.3 The FOSS Organization

A taxonomy of developers is incomplete unless married to a taxonomy of the structures within which they operate. On the surface, FOSS and proprietary development patterns may appear different (methods of joining the organization and metrics for defining prominence, for example). However, Mockus, et al. describe developer communication and collaboration patterns in the Apache HTTP Server and Mozilla Firefox that sound very similar to those found in

modern proprietary development organizations of comparable size [17]. In interviews with FOSS developers, Schweik and English found similar results [20].

Commit Patterns Organizational commit patterns are indicative of types and levels of developer collaboration. Monolithic commits may cause sweeping changes, while small commits indicate incremental development. In Production/Stable or Maintenance phase projects on SourceForge we found both patterns of small commits and patterns of large, monolithic commits [14, 15]. Further work has refined our notion of the reasons behind large commits through the development of a taxonomy of large commits [18]. However, correlation between commit behaviors in FOSS projects and commit behaviors in proprietary projects has not been adequately defined.

File Level Collaboration and Code Ownership Bird, et al. demonstrated that, in Windows Vista, Mozilla Firefox, and Eclipse 1) "software components with many minor contributors will have more failures than [those] that have fewer" and 2) ownership only had a consistent, statistically significant effect on Windows Vista [1]. In addition, previous studies have uncovered patterns of author contributions that may or may not be analogous to those in proprietary development [22, 23].

Communication Patterns Crowston and Howison found that FOSS projects on SourceForge exhibit myriad communication patterns. They contend that employing metaphors such as "the cathedral and the bazaar" [19] doesn't capture the complexity and variance within the many projects on SourceForge [6].

3 Toward a Taxonomy

We propose to begin to develop a taxonomy and theoretical framework of FOSS developers and organizations. This framework should provide a common foundation upon which researchers may base their dialogue. Specifically, we will create sub-taxonomies of FOSS and proprietary projects along the following axes 1) Job Tenure, 2) Developer Specialization, and 3) Code Ownership. In addition, we propose future work developing taxonomies of Work Related Developer Background and Non-Work Related Developer Background.

3.1 Data Sources

We use several data sources to draw conclusions about the state of these development communities: 1) the Current Population Survey, 2) a local, small company with which we have a research relationship, 3) two potential large companies, and 4) online FOSS repositories.

Current Population Survey Aside from the decennary census, the U.S. Census Bureau compiles a monthly *Current Population Survey*. The Bureau collects survey responses regarding numerous subjects, ranging from labor to living conditions, from a "multistage stratified sample of approximately 72,000 assigned housing units from 824 sample areas" [3]. Most importantly, this survey collects information relating to job tenure, job satisfaction, occupation, job industry, salary, number of hours worked at each job that a respondent holds, race, age, familial status, living conditions, and religion.

Small Companies We have affiliations with a small (around 30 developers) local company with whom we have run organizational studies. This company has been a leader in its market for over 20 years and employs both long tenured and newly hired developers.

Large Partners We have relationships with two large development organizations that have research divisions. In both cases, we should be able to leverage those relationships to develop taxonomies of large, proprietary organizations and their developers.

FOSS Repositories Publicly available data from the Apache Foundation, the Eclipse Foundation, the Mozilla Foundation, SourceForge, other FOSS "forges," and the SourceForge Research Data Archive [27] provide rich information about developer interaction and productivity. Many of the studies in Section 2 used this data [1, 6, 8, 9, 14, 15, 16, 18, 22, 23].

3.2 Job Tenure

Determining job tenure from census data is straightforward, as is supplementing the census data with data from companies with which we have research affiliations. In contrast, before attempting to determine job tenure in a FOSS project, we must first define what "tenure" means. If a developer commits a patch, takes ten years off, and then commits another patch, is that considered a ten year tenure? Common sense says no. But where does one draw the line?

Instead of a single definition of job tenure, we propose a taxonomy of FOSS developer tenure that incorporates time spent on a project, productivity during that time period, periods of inactivity and developer centrality. We can draw this data from the commit logs and mailing lists of online FOSS communities.

3.3 Code Ownership

Previous studies have correlated code ownership with defects [1] and have explored patterns of code ownership in SourceForge [23], Eclipse projects [22], Mozilla Firefox, and Windows Vista [1]. The taxonomy of code ownership would contain attributes such as proportion of files owned by a single author, distribution of file ownership measured through authorship entropy [23], density of commits to a file, and frequency with which a developer replaces lines of code contributed by other, less prominent developers.

3.4 Developer Specialization

Developer specialization can be gleaned from commit histories by identifying the modules to which developers commit code. Commits can then be assigned to categories such as UI development, database connectivity, and operating system integration. When possible, developer interviews provide fine-grained insight into the perceived specialization within the organization. Our analysis of specialization hinges on our collaborative research efforts with outside companies and the availability of FOSS repositories.

In addition to direct approaches for FOSS projects, we can also utilize indirect methods such as topic analysis of bug comments and mailing lists to identify developers within a FOSS organization who hold specialized information [2, 13]. Using this information, in conjuction with commit behaviors, should uncover developers who have specialized information, but who use that information in the aid of others in a mentoring or leadership capacity.

3.5 Future Goals

Although not yet formalized, and likely outside the scope of the work of a single doctoral student, taxonomies of developer background, both work-related and not, are vital to understanding the differences (if any) that exist between the population in the FOSS community and the developer community as a whole.

Developer Background (Work Related) Work-related developer background would require interviews with developers in both FOSS and proprietary organizations, a daunting, but achievable task.

Developer Background (Non-Work Related) Ironically, the data to determine non-work related background (socio-economic status, race, familial status, etc.) is freely available for the general population of developers in the United States through the *Current Population Survey* [3]. Ghosh, et al. provided survey results on some of these attributes for the FOSS community, but, as noted in Section 2.1, the granularity of the data is insufficient to build a taxonomy of developers. Determining these attributes for just FOSS developers would require personal interviews as well as very personal questions. While we would love to conduct these interviews, finding cooperative subjects may prove challenging.

4 Conclusion

Proclaiming the supposed advantages or disadvantages of FOSS and proprietary development paradigms ignores the rich and diverse attributes of development organizations. Likewise, qualifying research as resulting from analysis of open or closed source organizations both overgeneralizes by extrapolating to projects that may not be similar and misses possible applicability to similar projects

that have different paradigms. In qualifying research results, the question is not "Are open and closed source projects the same?" Instead, we must ask several questions, such as

1. What are the taxonomies of FOSS and proprietary project?
2. How do the projects from which we draw our data fit into these taxonomies?
3. Now that we understand how these projects are similar to or different from other projects, how does that affect the interpretation of our results?

We propose taxonomies of FOSS and proprietary developers and organizations so that we, as a research community, can ground our findings within a common theoretical framework. Doing so will enhance the dialogue among researchers within the OSS research community as well as between the research community and development organizations, both FOSS and proprietary. The end goal of this effort is to more precisely define our lexicon and increase the practical applicability and acceptance of our work.

References

1. C. Bird, N. Nagappan, H. Gall, P. Devanbu, and B. Murphy. An Analysis of the Effect of Code Ownership on Software Quality across Windows, Eclipse, and Firefox. Technical report, Technical report, UC Davis, 2010.
2. D.M. Blei, A.Y. Ng, and M.I. Jordan. Latent Dirichlet Allocation. *The Journal of Machine Learning Research*, 3:993–1022, 2003.
3. U.S. Census Bureau. Design and Methodology: Current Population Study. Technical report, 2006.
4. P.G. Capek, SP Frank, S. Gerdt, and D. Shields. A History of IBM's Open-Source Involvement and Strategy. *IBM systems journal*, 44(2):249–257, 2005.
5. S.P. Choy, E.M. Bradburn, C.D. Carroll, National Center for Education Statistics, and Institute of Education Sciences (US). *Ten Years After College: Comparing the Employment Experiences of 1992-93 Bachelor's Degree Recipients with Academic and Career-oriented Majors*. National Center for Education Statistics, Institute of Education Sciences, US Dept. of Education, 2008.
6. K. Crowston and J. Howison. The Social Structure of Free and Open Source Software Development. *First Monday*, 10(2):1–100, 2005.
7. R.A. Ghosh, R. Glott, B. Krieger, and G. Robles. Free/libre and Open Source Software: Survey and Study, 2002.
8. Jonathan L. Krein, Alexander C. MacLean, Daniel P. Delorey, Charles D. Knutson, and Dennis L. Eggett. Language Entropy: A Metric for Characterization of Author Programming Language Distribution. *4th Workshop on Public Data about Software Development*, 2009.
9. Jonathan L. Krein, Alexander C. MacLean, Daniel P. Delorey, Charles D. Knutson, and Dennis L. Eggett. Impact of Programming Language Fragmentation on Developer Productivity: a SourceForge Empirical Study. In *International Journal of Open Source Software and Processes (IJOSSP)*, volume 2, pages 41–61, June 2010.

10. Jonathan L. Krein, Patrick Wagstrom, Stanley M. Sutton Jr., Clay Williams, and Charles D. Knutson. The Problem of Private Information in Large Software Organizations. In *International Conference on Software and Systems Process*, New York, NY, USA, 2011. ACM Press.
11. K.R. Lakhani and R.G. Wolf. Why Hackers Do What They Do: Understanding Motivation and Effort in Free/Open Source Software Projects. *Perspectives on Free and Open Source Software*, pages 3–22, 2005.
12. J. Lerner and J. Tirole. Some Simple Economics of Open Source. *The Journal of Industrial Economics*, 50(2):197–234, 2002.
13. Alexander C. MacLean, Landon J. Pratt, and Charles D. Knutson. Knowledge Homogeneity and Specialization in the Apache HTTP Server Project. *Publication pending*, 2011.
14. Alexander C. MacLean, Landon J. Pratt, Jonathan L. Krein, and Charles D. Knutson. Threats to Validity in Analysis of Language Fragmentation on SourceForge Data. In *Proceedings of the 1st International Workshop on Replication in Empirical Software Engineering Research (RESER '10)*, page 6, May 2010.
15. Alexander C. MacLean, Landon J. Pratt, Jonathan L. Krein, and Charles D. Knutson. Trends That Affect Temporal Analysis Using SourceForge Data. In *Proceedings of the 5th International Workshop on Public Data about Software Development (WoPDaSD '10)*, page 6, June 2010.
16. A. Mockus, R.T. Fielding, and J. Herbsleb. A Case Study of Open Source Software Development: The Apache Server. In *Proceedings of the 22nd international conference on Software engineering*, pages 263–272. Acm, 2000.
17. Audris Mockus, Roy T. Fielding, and James D. Herbsleb. Two Case Studies of Open Source Software Development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology*, 11(3):309–346, 2002.
18. Landon J. Pratt, Alexander C. MacLean, and Charles D. Knutson. Cliff Walls: An Analysis of Monolithic Commits Using Latent Dirichlet Allocation. *Publication pending*, 2011.
19. E.S. Raymond et al. *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. O'Reilly & Associates, Inc., 2001.
20. C.M. Schweik and R. English. Tragedy of the FOSS Commons? Investigating the Institutional Designs of Free/Libre and Open Source Software Projects. 2007.
21. R.M. Stallman. *Free Software, Free Society: Selected Essays of Richard M. Stallman*. Joshua Gay, 2002.
22. Quinn C. Taylor, Jonathan L. Krein, Alexander C. MacLean, and Charles D. Knutson. An Analysis of Author Contribution Patterns in Eclipse Foundation Project Source Code. *Publication pending*, 2011.
23. Quinn C. Taylor, James E. Stevenson, Daniel P. Delorey, and Charles D. Knutson. Author Entropy: A Metric for Characterization of Software Authorship Patterns. In *3rd International Workshop on Public Data about Software Development (WoPDaSD '08)*, September 2008.
24. Linus Torvalds. The Torvalds Transcript: Why I 'Absolutely Love' GPL Version 2, March 2007.
25. E.L. Uhlmann and G.L. Cohen. Constructed Criteria: Redefining Merit to Justify Discrimination. *Psychological Science*, 16(6):474–480, 2005.
26. E.L. Uhlmann and G.L. Cohen. "I Think It, Therefore It's True": Effects of Self-Perceived Objectivity on Hiring Discrimination. *Organizational Behavior and Human Decision Processes*, 104(2):207–223, 2007.

27. M. Van Antwerp and G. Madey. Advances in the SourceForge Research Data Archive (SRDA). In *Fourth International Conference on Open Source Systems, IFIP 2.13 (WoPDaSD 2008)*, Milan, Italy, September 2008.
28. J. West. How Open is Open Enough?:: Melding Proprietary and Open Source Platform Strategies. *Research Policy*, 32(7):1259–1285, 2003.

Semi-Automatic Evaluation of Free Software Projects: A Source Code Perspective

Paulo Meirelles[1], Fabio Kon[1], and Carlos Santos Jr.[1,2]

[1] FLOSS Competence Center - University of São Paulo
Rua do Matão, 1010, São Paulo, SP, Brazil
{paulormm,fabio.kon}@ime.usp.br
[2] Horizon Institute - University of Nottingham
Triumph Rd, Nottingham NG7, United Kingdom
carlos.denner@nottingham.ac.uk

Abstract. There has been a trend towards managing software development projects as any other production processes and systems. Consequently, important technical aspects of software development have not been explored in some software engineering communities. On the other hand, free software communities tend to work based on technical approaches. Specially, they look at the source code. However, despite the "show me the code" culture, source code metrics are often not perceived as an indicator of quality. To promote the use of source code metrics to optimize free software development, we are investigating the effect of source code metrics on free software attractiveness. Besides, we are defining a systematic approach and developing tools to use, interpret, and understand software metrics.

Key words: Free software, source code metrics, clean code, attractiveness.

1 Introduction

An issue in any software system is its complexity. Since there are not physical constraints in software (such as material wear, manufacturing costs, weight, etc), there are not external factors that might restrict its development. Thus, most non-trivial programs quickly reaches large complexity level, approaching the limit of the ability of a software engineer understand the source code [1].

A consequence of this complexity is a higher cost because software development involves the work of skilled professionals for long periods. Another consequence can be a lower quality since it becomes difficult to perform effective testing, eliminate bugs, and implement new features. However, free software[1] development is guided by code sharing, which may allow to identify and fix problems, as well as make improvements faster. It can involve a larger number

[1] In this work, we consider the terms "Free Software","Open Source Software" (OSS), "Free/Open Source Software" (FOSS), and "Free/Libre/Open Source Software" (FLOSS) equivalent.

developers and users than any other software development methodology [2], so having the potential for peer-review from different collaborators around the world [3].

In the context of methodologies used by free software communities, source code is the main product of software development activities. In general, software source code is written gradually and different developers make improvements on an ongoing basis [4]. New features are inserted and bugs are resolved during software development and maintenance tasks. Thus, features are constantly delivered to users.

However, there is a significant gap between the numbers of lines of code which a software engineer reads and writes. Usually, software engineers read hundreds of lines of code to understand a software implementation to make improvements [4]. Therefore, source code should be written to be read by people since software engineers need to analyze source codes to understand better software projects, as well as Software Engineering requires the full understanding of software, which is the result from the writing of source code.

In this scenario, software source code metrics can help software engineers to observe the source code quality. Also, they can support the development of clean code, i.e., clear, flexible, and simple [4]. However, many free software projects do not practice source code quality evaluation and have no tools available to do so. This lack of systematic code evaluation leaves a lot of room for improvement in the development of free software [3].

To address these issues, we are investigating an approach that software engineers should make decisions about their codes when they are programming at the method and class level. To make the best decisions, we argue that they should monitor attributes from their source code. The sum of these decisions influences the source code quality [5].

Our proposal is based on an automated evaluation of source code metrics and an objective way to interpret their values. Thus, software engineers can monitor specific characteristics of their code. For that, we are developing a tool called Kalibro Metrics. Moreover, we are selecting source code metrics according to a study of the effect of source code quality on free software attractiveness [6]. Also, we are defining a mapping among clean code concepts, troublesome scenarios, source code metrics, and attractiveness to provide a systematic approach to make decisions about the source code clarity, flexibility and simplicity.

To show our ideas and preliminary results, the remainder of this paper is organized as follows: Section 2 describes related work. Section 3 shows our research design. Section 4 presents the Kalibro Metrics tool. Section 5 concludes the paper and discusses future work.

2 Background

2.1 Related projects

In this work so far, we have found some projects related to automatic quality evaluation of free software:

- **FLOSSMetrics** (Free/Libre Open Source Software Metrics) is a project that uses existing methodologies and tools to provide a large database with information about free software development [7].
- **Ohloh** is a website that provides a web services suite and an on-line community platform that aims at building an overview of free software development [8].
- **Qualoss** (Quality in Open Source Software) is a methodology to automate the quality measurement of free software projects, using tools to analyze the source code and the project repository information [9].
- **SQO-OSS** (Software Quality Assessment of Open Source Software) provide a suite of that allows analysis and benchmarking of free software projects [10].
- **QSOS** (Qualification and Selection of Open Source Software) is a methodology based on 4 steps: used reference definition; software evaluation; qualification of specific users context; selection and comparison of software [11].
- **FOSSology** (Advancing open source analysis and development) is a project that provides a free database with information about the software license [12].
- **QualiPSo** (Trust and Quality in Open Source Systems) defined procedures to boost the use of free software and adoption of its development practices within software industry [13]. A set of tools was integrated with QualiPSo Quality Platform.
- **HackyStat** is an environment for analysis, visualization, interpretation of software development process and product data [14].

In short, we have identified from current available projects and their tools the lack of the following features: (i) to collect automatically source code metrics values considering different programming languages; (ii) to interpret measurement results, associating them with source code quality. Therefore, we are developing the Kalibro Metrics tool that can be configured to show metric results in a friendly way, helping software engineers to spot design flaws to refactor, project managers to control source code quality, and software researchers to compare specific source code characteristics across free software projects.

2.2 Source code analysis tools

During our research and development activities, we studied or used 11 free software source code analysis tools: Analizo [15], CCCC [16], Checkstyle [17], CMetrics [18], CPPX [19], Cscope [20] CTAGX [19], LDX [19] JaBUTi [21], MacXim [22] and Metrics (Eclipse plug-in) [23]. Also, we have defined the following requirements for our tool, according to our theoretical and practical needs, as well as our ideas to explorer better source code metrics:

- The tool should support source code metrics **thresholds** to provide different interpretations about metric values.
- The tool should support the analysis of different programming languages (**multi-language**).
- The tool should provide clear interfaces for adding new metrics and supporting different programming language (**extensibility**).
- The tool should be **free software**, available without restrictions to allow other researchers to replicate our studies and results fully.
- The tool should be supported by active developers who know the tool architecture (**maintained**).

Tools	**Languages**	**Extensible**	**Thresholds**	**Maintained**
Analizo	C, C++, Java	Yes	No	Yes
CCCC	C++, Java	No	No	Yes
Checkstyle	Java	No	Yes	Yes
CMetrics	C	Yes	No	Yes
CPPX	C, C++	No	No	No
Cscope	C	No	No	Yes
CTAGX	C	No	No	No
LDX	C, C++	No	No	No
JaBUTi	Java	No	No	Yes
MacXim	Java	No	No	Yes
Metrics	Java	No	Yes	Yes

Table 1. Existing tools versus our defined requirements

In Table 1, we can compare all of these tools. The Analizo [15] is the closest tool from our requirements. It is able to analyze source code from our initial three required languages (C, C++, and Java). However, also we want a tool with thresholds support. Thus, Analizo was selected as the default source code analysis tool integrated with our metric tool, called Kalibro Metrics (that will be described in details in the section 4).

2.3 Related works

To select which metrics we should study their thresholds and relate them to clean code concepts, we investigating which source code metrics influence in free software projects success, i.e., its attractiveness. Santos Jr. *et al.* [24] defined a theoretical model for attractiveness as a crucial construct for free software projects, proposing their (i) typical origins (e.g., license type and intended audience); (ii) indicators (e.g., number of members and downloads); (iii) consequences (e.g, levels of activity and time for task completion). They suggested that the success of any project depends on its level of attractiveness to potential contributors and users. Based on this model, we are exploring some of the

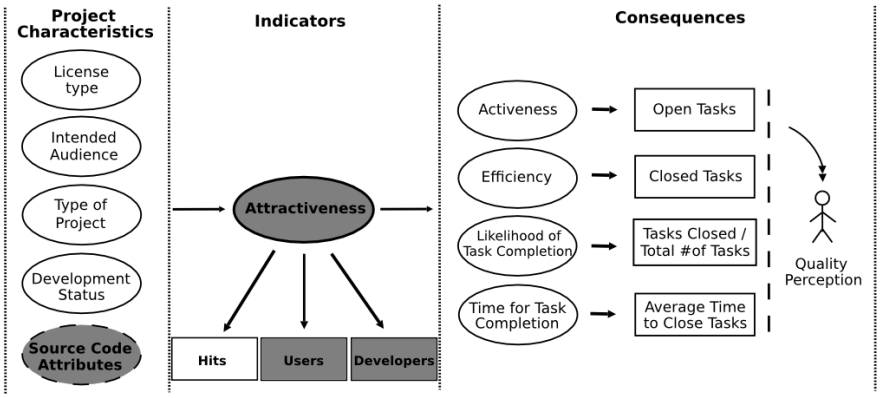

Fig. 1. Attractiveness research model – adapted from Santos Jr. et al [24].

factors that may enable projects to build a community by attracting users and developers.

In short, we have proposed a new element that can explain attractiveness partially. According to our first hypotheses, we added source code attributes (from source code metrics) and expect that they would work in the same causal chain manner as shown in Figure 1. To test our ideas empirically, we analyzed 6,773 projects written in the C language from SourceForge.net [6]. Our first study was able to explain 18% of software download and 12% of project members, through a set of four source code metrics. Currently, we are collecting data from 42.335 projects written in the C, C++, and Java languages.

A systematic review of 63 empirical studies showed that there is little research addressing the characteristics or properties of free software projects, such as their quality, growth, and evolution [25]. In this context, we are analyzing source code metrics from an unprecedented number of free software projects, linking their source code characteristics and attractiveness.

For example (comparing our samples to others from related works), Barkmann *et al.* [26] analyzed 146 free software projects written in Java, identifying the correlation between a set of object-oriented metrics and their theoretical ideal values. Stamelos *et al.* [27] compared quality characteristics of 100 applications written for GNU/Linux to industrial standards. Midha [28] analyzed 450 projects from SourceForge.net and verified that high values of MacCabe's Cyclomatic Complexity and Haltead's Effort are positively correlated with the number of bugs and with the time needed to fix bugs. Capra *et al.* [29] have shown that open governance is associated with higher software design quality on a study with 75 free software projects. Finally, Bargallo *et al.* [30] analyzed 56 free software projects, studying the relationship between software design quality and project success.

3 Research Design

3.1 Research Hypotheses

In our first study about the relationships between source code metrics and attractiveness [6], we investigated whether source code metrics might influence the attractiveness of free software projects. Thereby, we can later observe whether these attributes influence people's perception of quality as a consequence of attractiveness. We formulated our hypotheses according to the attractiveness model showed in Figure 1:

- H1: Free software projects with higher structural complexity (coupling and lack of cohesion) have lower attractiveness.
- H2: Larger free software projects (with higher lines of code and number of modules) have higher attractiveness.

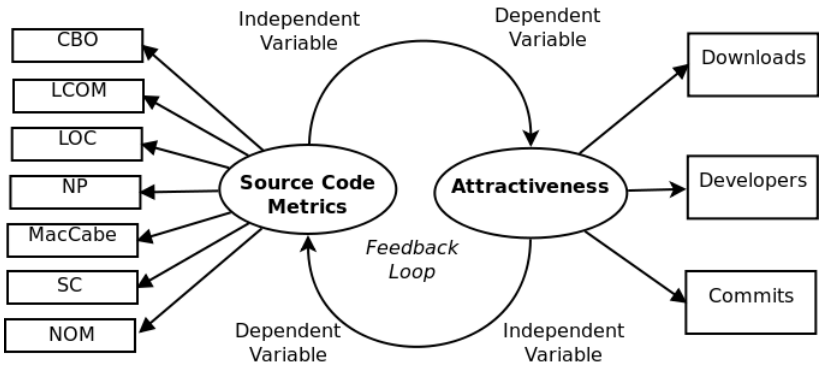

Fig. 2. An proposal of a reciprocal effect model.

We have defined a new attractiveness model that aims to explore the reciprocal effect between source code quality and free software project attractiveness. Metrics have been including in this model according to a mapping from clean code concepts to source code metrics. Thus, currently, we are working on the following hypotheses to investigate this feedback loop between source code metrics and attractiveness, as showed in Figure 2:

- H1: Free software projects with lower number of clean code problems have better source code metrics values (probably it can be applied to any software project).
- H2: Free software project with better source code metrics values have higher attractiveness.
- H3: Free software projects with higher attractiveness have better source code metrics values (feedback loop).

3.2 Data Sample

SourceForge.net shares its data to support free software researchers. At the first moment, we used the data available in a database managed by the University of Notre Dame[2] and another one provided by the FLOSSMole project[3]. Later, we developed a set of scripts that access SourceForge.net pages. They collect information such number of download, number of member, type of repository, etc, providing us our own database. After that, we select data matching the following criteria:

- *Source code language*: In our first study relating source code metrics and attractiveness, we select C projects. Currently, we included the C++ and Java programming languages.
- *More than one download*: Projects with no downloads are probably either non-development projects, or projects that have just started, or are other special cases.

At the first moment, this criteria provided us a list of 11,433 projects. When including C++ and Java, we obtained a list of 42.335 projects. Also, we have another script that gets a list of projects and visit the file pages of each project and download the last source code package available. However, when we download these files, some project had no available files (empty "files" section in the SourceForge.net project pages). Thus, we have observed that we can collect source code values to 60% of our original list of projects. For our next study, we expect to analyze about 25.000 projects from SourceForge.net.

After that, we run Analizo that collects source code metric values sequentially for all projects and stores the computed metrics in a single database. Finally, both project information and source code values databases were cross-joined, so we can perform the needed statistical analysis.

4 Kalibro Metrics

We are developing Kalibro Metrics to apply our research results about clean code concepts, source code metrics, and free software attractiveness. It is a free software and web-service-based tool to analyze and understand source code metrics. Kalibro Metrics can connect different kinds of repositories to download the source code from a software project.

Our final goal is to build a social network to monitor and analyze source code metrics, called Mezuro. This source code monitoring network has been developed. Mezuro is a Noosfero social network platform [31] instance with a Kalibro Plug-in activated, connecting the Kalibro Web Service. Figure 3 shows an interaction diagram, detailing all these possibilities to use Kalibro.

Kalibro Metrics provides the following features:

[2] `nd.edu/~oss/Data/data.html`
[3] `flossmole.org`

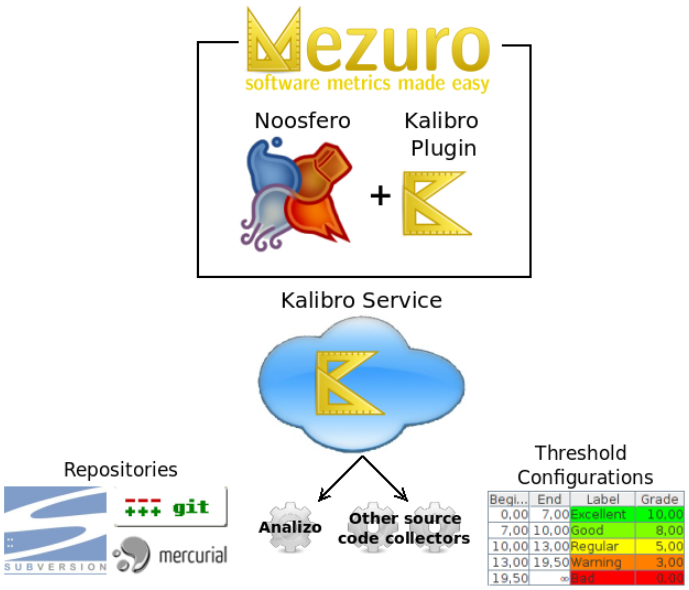

Fig. 3. Kalibro interactions diagram

- Download source code from Subversion, Git, Mercurial, Baazar, and CVS repositories.
- Download source code from local and remote zip and tarball (.tar, .tar.gz, and tar.bz) files.
- Creation of configurations, i.e., a set of metrics for being used in the evaluation of a software source code.
- Creation of ranges associated with a metric and a qualitative evaluation.
- Creation of new metrics (via JavaScript) based on the ones provided by the metric collector tool.
- Calculation of statistics results for higher granularity modules (e.g. average LOC of classes inside a package).
- Possibility of exporting results to a CSV (comma-separated values) file.
- Calculation of a grade for the source code analysis projects, based on given weights for each metric and grades for ranges. This allows cross-project comparisons.
- Possibility of making interpretation more user-friendly by associating colors with ranges.

4.1 Kalibro Plug-in for Mezuro networking

Mezuro is a social network to track source code metrics. This environment promotes an open and collaborative networking to analyze hundreds of thousands

software projects, specially Free Software, through an automated tracking of their source code repositories.

Mezuro is a powerful environment to enhance Kalibro features, using the social network potential. The idea is based on the fact that people improve their writing skills when they read good books and papers. Similarly, software engineers can increase the quality of their source codes reviewing good and clean codes. They can find related projects through source code and compare their source code characteristics. Thus, a social network associate with Kalibro Metrics features can provide a collaborative technology roadmap for an automated source code analysis approach.

Kalibro Metrics was adapted as a Noosfero plug-in. Kalibro plug-in development steps have led the Noosfero plug-in framework. Figure 3 presents that we have connected this plug-in to Kalibro Web Service. At this moment, we are implementing the source code analysis history with a graphical software visualization to complete the Kalibro/Mezuro source code metrics interpretation approach.

Mezuro provides a friendly service on the Internet from the user point of view. In short, users just need to give a source code repository URL. In addition, users can access the source code analysis report from an asynchronous way, i.e. when they wish or need. The history of source code metric values and analysis are recorded. Moreover, all free and public project analysis are available to any user.

Finally, any user can suggest metric threshold configurations and share them on the Mezuro network. This provides a Bazaar style, as defined by Eric Raymond [2] for Free Software development, but in this case to evolve and define the best way to explore the source code metrics potential. In other words, an semi-automated source code analysis approach via source code metrics interpretation.

5 Final remarks

This paper presented the current status and preliminary results of this Ph.D research. We are defining an approach that promotes the use of source code metrics to optimize free software development. Our first results indicated that source code metrics explain a relevant percentage of free software projects attractiveness. It is based on human perceptions and influenced by people's cognition, making it a complex issue, hard to understand and explain completely. Nevertheless, our first study was able to explain partially the number of downloads and number of members through a set of source code metrics.

Our first sample was restricted to projects written in C available. Currently, we included projects written in C++ and Java, as well as extend our study to other source code metrics. The next step is a study about the reciprocal effects between attractiveness and source code metrics in free software projects.

From a practical point of view, we are developing Kalibro Metrics, which represents a new generation of source code metrics analysis tool, to apply our research result and support a better use of source code metrics. Kalibro provides an environment where software engineers can define their own threshold configurations, according to software implementation context and their experiences in software development.

Future Kalibro Metrics features include the integration with other metric collector tools, especially to provide Perl, Python, and Ruby source code analysis. Moreover, we are developing the Mezuro source code network. It is an environment for source code tracking, analysis, and visualization. Mezuro connects the Kalibro Web Service via the Kalibro Plug-in for Noosfero social network platform.

Acknowledgment

The authors of this paper were supported by Brazilian National Research Council (CNPq). Also, a special thanks to Dr. John Pearson and his department to receive PhD. student Paulo Meirelles as visiting researcher at Southern Illinois University Carbondale (SIUC).

References

1. R. Stallman, "Software patents – obstacles to software development," Spoken presentation, 2002. [Online]. Available: http://www.cl.cam.ac.uk/~mgk25/stallman-patents.html
2. E. S. Raymond, *The Cathedral & the Bazaar* , T. O'Reilly, Ed. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 1999.
3. M. Michlmayr, F. Hunt, and D. Probert, "Quality Practices and Problems in Free Software Projects," in *First International Conference on Open Source Systems*, M. Scotto and G. Succi, Eds., Genova, Italy, 2005, pp. 309–310.
4. R. C. Martin, *Clean Code - A Handbook of Agile Software Craftsmanship.* Prentice Hall, 2008.
5. K. Beck, *Implementation Pattens.* Addison Wesley, 2007.
6. P. Meirelles, C. Santos Jr., J. Miranda, F. Kon, A. Terceiro, and C. Chavez, "A study of the relationships between source code metrics and attractiveness in free software projects," *Software Engineering, Brazilian Symposium on*, vol. 0, pp. 11–20, 2010.
7. FLOSSMetrics, "Flossmetrics - free/libre open source software metric," http://www.flossmetrics.org/, 2011.
8. Ohloh, "Ohloh - the open source network," http://www.ohloh.net, 2011.
9. Qualoss, "Qualoss - quality in open source software," http://www.qualoss.org, 2011.
10. SQO-OSS, "Sqo-oss: Software quality assessment of open source software," http://www.sqo-oss.eu/, 2008.

11. QSOS, "Qsos - qualification and selection of open source software," http://www.qsos.org, 2011.
12. FOSSology, "Fossology - advancing open source analysis and development," http://www.fossology.org, 2011.
13. "Qualipso: Quality platform for open source software," Web Site, April 2011. [Online]. Available: http://qualipso.org
14. HackyStat, "Hackystat - a framework for collection, analysis, visualization, interpretation, annotation, and dissemination of software development process and product data." http://code.google.com/p/hackystat/, 2011.
15. A. Terceiro, J. Costa, J. Miranda, P. Meirelles, L. R. Rios, L. Almeida, C. Chavez, and F. Kon, "Analizo: an extensible multi-language source code analysis and visualization toolkit." in *Brazilian Conference on Software: Theory and Practice (CBSoft) – Tools*, Salvador-Brazil, 2010.
16. "C and C++ Code Counter," Web Site, April 2011. [Online]. Available: http://cccc.sourceforge.net
17. "Checkstyle," Web Site, April 2011. [Online]. Available: http://checkstyle.sourceforge.net
18. "Cmetrics: Size and complexity metrics for c source code files," Web Site, April 2011. [Online]. Available: http://tools.libresoft.es/cmetrics
19. A. E. Hassan, Z. M. Jiang, and R. C. Holt, "Source versus object code extraction for recovering software architecture," in *Proceedings of the 12th Working Conference on Reverse Engineering (WCRE'05)*, 2005.
20. "Cscope," Web Site, April 2011. [Online]. Available: http://cscope.sourceforge.net
21. A. M. R. Vincenzi, W. E. Wong, M. E. Delamaro, and J. C. Maldonado, "JaBUTi: A Coverage Analysis Tool for Java Programs," in *XVII SBES — Brazilian Symposium on Software Engineering*, 2003, pp. 79–84.
22. "Macxim," Web Site, April 2011. [Online]. Available: http://qualipso.dscpi.uninsubria.it/macxim
23. "Metrics," Web Site, April 2011. [Online]. Available: http://metrics.sourceforge.net
24. C. Santos Jr., J. Pearson, and F. Kon, "Attractiveness of Free and Open Source Software Projects." in *Proceedings of the 18th European Conference on Information Systems (ECIS)*, Pretoria, South Africa, 2010, (forthcoming).
25. K.-J. Stol, M. A. Babar, B. Russo, and B. Fitzgerald, "The Use of Empirical Methods in Open Source Software Research: Facts, Trends and Future Directions," in *FLOSS'09: Proceedings of the 2009 ICSE Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 19–24.
26. H. Barkmann, R. Lincke, and W. Löwe, "Quantitative Evaluation of Software Quality Metrics in Open-Source Projects," in *AINA Workshops*, 2009, pp. 1067–1072.
27. I. Stamelos, L. Angelis, A. Oikonomou, and G. L. Bleris, "Code Quality Analysis in Open Source Software Development," *Information Systems Journal*, vol. 12, pp. 43–60, 2002.
28. V. Midha, "Does Complexity Matter? The Impact of Change in Structural Complexity On Software Maintenance and New Developers' Contributions in Open Source Software," in *ICIS 2008 Proceedings*, 2008.
29. E. Capra, C. Francalanci, and F. Merlo, "An Empirical Study on the Relationship Between Software Design Quality, Development Effort and Governance in Open

Source Projects," *IEEE Transactions on Software Engineering*, vol. 34, no. 6, pp. 765–782, Nov.-Dec. 2008.

30. D. Barbagallo, C. Francalenei, and F. Merlo, "The Impact of Social Networking on Software Design Quality and Development Effort in Open Source Projects," in *ICIS 2008 Proceedings*, 2008. [Online]. Available: {http://aisel.aisnet.org/icis2008/201}
31. "Noosfero: A free web-based platform for social and solidarity economy networks," Web Site, April 2011. [Online]. Available: http://noosfero.org

Reprogramming Open Source Ecosystems – case study of MeeGo

Jarkko Moilanen[1]
1 University of Tampere, School of Information Sciences, Kalevantie 4,
33014 Tampereen Yliopisto, Finland jarkko.moilanen@uta.fi,
WWW home page: http://blog.ossoil.com

Abstract. Research focuses on describing power variations in open source ecosystems with qualitative methods focus on MeeGo ecosystem. One of the goals is to describe how system and software developers understand open source ecosystem. Research adopts Castells' theory of Communication Power model, which includes four distinctive forms of power (yet named rather ambiguously) to include: 1) networking power, 2) network power, 3) networked power and 4) network-making power. Research data is collected with prolonged observation and multiple interviews inside MeeGo community. Living with the community and observing member activities enables selection of fruitful events community confronts for further research, which is based on themed interview. Theme based interviews are used for what has happened during the events what has changed and how. Selected interviewees are developers, whose work is open source related.

1 Introduction and motivation

Open source communities have had a tremendous affect in our societies since the 1990s. The features of Open Source development have challenged traditional closed source development. There can hardly be anyone to deny the importance of Open Source. Some scholars argue that firms and open source communities have been forced to rethink their business strategy, innovation process and way of operating. This is true at least in Operating System (OS) development. Current Operating Systems have become extremely complex and development requires a lot of resources due to demands created by markets. Yet at the same time markets demand rapid response and quick results. Hardly any company can afford to do OS development alone and behind closed doors. Of course there are exceptions, but tendency seems to be the opposite.

Open Source communities have been around for a few decades, but current development in open source have transformed to something else than what it used to be. Several scholars have written about this change in modus operandi or paradigm shift [see 11, 23, 6]. In the beginning of open source development, it was mostly based on volunteers also known as the bazaar model [28]. People participated development on their free-time without any monetary compensation. They were doing it 'just for the fun' [34].

In the same time, companies have had huge R&D departments with hundreds of highly trained and paid developers. Over the past decade this has changed due to various reasons. Internal R&D that has been a valuable strategic asset, has become too expensive and slow to meet the market need. Open Source development, at least concerning large applications and OS development, currently includes strong participation of companies and more complex modes of relationships between different participants of development. Governments, companies, non-profit foundations and universities have entered the world of open source for various reasons described in Schweik's 'Theory of Goods' [29]. Those reasons include just to mention a few saving in IT costs, to differentiate from competitors areas and grow, encourage interoperability, open standards and economic development, and build reputation. While most of open source developers are still volunteers, a significant amount (17%) of developers are paid to participate in different open source projects [15]. Fitzgerald has labeled this new form of Open Source development as 'OSS 2.0' [11]. The result of change can be located under the concepts of 'Open Innovation' or 'Open Source Software Ecosystems'. Difference between the concepts is point of view and emphasis.

Open Source development is in transition and more companies are getting involved in Open Source Ecosystems (OSE). Yet knowledge about OSE is rather limited and approach has been more or less economical rather than cultural and describing. This research tries to extend knowledge about Open Source Ecosystems from developers viewpoint: 1) how developers understand OSE, 2) what are fundamental parts of OSE, 3) how ecosystem partners can influence development process and outputs, 4) how decision-making functions and 5) how decision-making changes during crisis situations and other strategically important events. Results can be used to adjust strategies of companies participating in OSE development. Knowing how OSE functions, gives tools to make predictions about future and adjust company strategies.

In this paper, we provide an insight to goals and research methods for studying the balance of power in an open source ecosystem. Focus is on major strategic events during development, which affect system development in different ways. Selected ecosystem view is developer viewpoint, not economical.

In Sections 2 and 3, we discuss the background of the paper, and cover topics associated with ecosystems and power. In Section 4 we introduce research methods that will be used to carry out the research. In Section 5, we provide interview themes with questions and discuss current status of research. In Section 6 we draw some final conclusions.

2 About ecosystems

In this research, ecosystem is preferred concept since the focus is more general and not limited to innovation. Furthermore, ecosystem approach is more holistic and since the focus of this research is fluctuations of power, the choice is legitimate.

The terms community and ecosystem are understood in this research as follows. Community refers to all developers, users, designers, architects, board members, managers and volunteers who are behind all the 'goods' produced by ecosystem. In other words, community refers to humans involved in ecosystem. The concept of Software Ecosystems is more general in nature than Open Innovation. Messerschmitt and Szyperski [23] have written a book which explains the essence and effects of a "Software Ecosystem". According to them software ecosystem is a set of businesses functioning as a unit and interacting with a shared market for software and services, together with relationships among them. These relationships are frequently underpinned by a common technological platform and operate through the exchange of information, resources, and artifacts. More recently, Popp and Meyer [26] have discussed software ecosystems from software vendor's perspective.

Popp defines ecosystem from economical perspective as “a set of companies [...] exchange products or services to serve a common goal or to achieve higher levels of individual goals” [27]. Popp categorizes software ecosystems to three different types: 1) supplier ecosystem, 2) partner ecosystem and 3) customer ecosystem. [27] The partner ecosystem contains software partners including other software vendors and system integrators. The latter provides implementation services for different vendor's solutions. Customer ecosystem includes both existing and potential customers. Popp does not define what is included in the supplier ecosystem.

The term ecosystem in software development is not adopted or seen as appropriate by all. Stallmann who is often referred as the prophet of the free software movement, does not approve labeling free software community as an ecosystem [31]. According to Stallman, the word “implies the absence of ethical judgment” [31]. In other words, labeling community as ecosystem implicitly suggests that we use nonjudgmental observation. Some organisms consume other organisms, or as Stallman puts it “[w]e do not ask whether it is fair for an owl to eat a mouse or for a mouse to eat a plant, we only observe that they do so.”[31] Stallman also opposes the the word because it refers to ecological ecosystem where components which are weak, unreliable or are unable to adapt to changing circumstances quickly because of a lack of flexibility will often become less common overtime. In Software ecosystems components also evolve this way and may become effectively extinct if a more suitable replacement is found and adopted by all users. Stallman prefers to choose ethical stance where pure ecological phenomenons are opposed and beings can decide to preserve things that might otherwise vanish.

Ecosystems have been studied mostly from economical perspective. Therefore open source ecosystem related research is needed, especially from developer point of view. In this research focus is in defining open source ecosystem from developer's view. Another focus is on describing decision making variations and practices during major events which have strategic implications regarding ecosystem's future. In the latter, power as capability to influence others is one of the key elements.

3 About power

This research adopts Castells' theory of power described in his latest book 'Communication Power'. According to Castells communication power has four different forms: 1) networking power, 2) network power, 3) networked power and 4) network-making power.[4] The terms are somewhat ambiguous. In figure 1 the elements of power are illustrated as layers, which cover whole ecosystem, in this case general software ecosystem. In the above mentioned book Castells argues that communication networks are the key fields of power in the network society [4]. He describes two mechanisms - programming and switching - that turn networks into major sources of power.

Castells defines networks as "*a set of interconnected nodes [...] which are [...] complex structures of communication, constructed around a set of goals that simultaneously ensure a unity of purpose and flexibility of execution by their adaptability to the operating environment*" [4]. Their goals and rules of performance are "(re)programmed" to the values and interests of so called programmers, which are real people, not organizations. Programmers include those who engage in decision-making with the intention to create and manage or networks. Switching is about altering or modifying the networks and connections between them. In other words switchers (dis)connect networks different network to form strategic alliances and cop-operation to ward off competition.

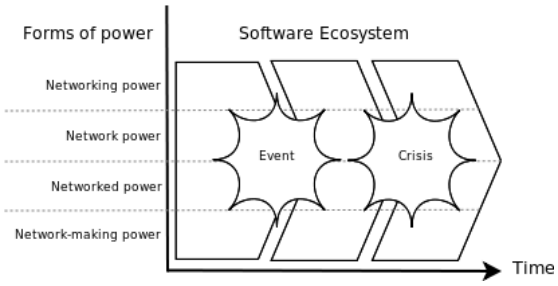

Fig. 1. Illustration of event selection combined with Castell's theory of power, which will be used is analysis.

Selected events, around which interviews build are illustrated as 'stars'. Event selection is described in details in separate chapter below.

4 Methodology

This research is qualitative and descriptive. In this research observation and interviews are the main data collection methods. Aim is to describe phenomenons, defined by research questions below, as profoundly as possible. The starting point for this study was grounded theory [32-33], which is a data-driven theory. This approach fits empirical situations and provides relevant predictions, explanations, interpretations, and implications. It might be suitable to inform the reader, that this study is not a pure application of grounded theory. A study or researcher does not live in a vacuum, or as Seale puts it ”[k]nowledge is always mediated by preexisting ideas and values, whether this is acknowledged by researchers or not” [30]. Some definitions and models of the phenomenon are taken from previous research results of various scholars.

4.1 Observation and interviews

Researcher is active member of MeeGo community both locally and internationally. Being part of the community which is essential part of the research as a resource and acting as part of it, is an example of ethnographic research setting. This thesis is focused on power fluctuations in one particular Open Source ecosystem, namely MeeGo. The aim is to describe the selected phenomenon, changing power relations, with precision. Living in the research subject world, in this case virtual and real world, for months and acting as community manager in Finnish MeeGo community, gives an advantage. To ignore the above strengths would not be wise. Therefore the research methodology includes some features of social anthropology and ethnographic approach. According to Apgar[1] ethnography “is essentially a decoding operation.” Furthermore, it is “a description of shared knowledge, or cognition,” which, “enables us to decode the observed behavior.” To do this the ethnographer must examine, and essentially learn the group’s language or discourse, the means through this knowledge is transmitted and intercommunicated. The ethnographer must identify the key concepts and their associated linguistic labels or lexemes [10]. The above process is hardly ever short or simple. Lincoln and Guba[25] describe “prolonged engagement” as the “investment of sufficient time” to truly learn a community’s culture. The learning process aims to enable testing for the misinterpretation of information and observations, and to build trust and establish rapport with the members of the community. Goffman goes as far as suggesting that the objective is to effectively penetrate the community, even to the point of becoming a member of the community or group [16]. This is is evidently true in this research. Researcher is member or community and acts as community manager for Finnish community. As a consequence of the above, ethnography as method or research discipline has the disadvantage of being time-consuming. Prolonged engagement is a tool to gain correct scope, but according to Lincoln and Guba [25] it is not enough if the aim is to achieve accurate descriptions or analysis of the subject.

Depth is needed and achieved with persistent observation. Persistent observation and prolonged engagement are in turn related to another important matter of validity and authority [30].

In this study, the role of researcher was between observer as participant and participant as observer though the latter was used in lesser occasions. Complete observer role was excluded because it would limit the depth and in some sense the width of data. Complete participant role was also excluded. The main reason for this was that it would take the researcher 'too' deep into to the world of informants. There was no real need to hide the researchers identity from the informants, which would have been somewhat unethical. Besides, most of the possible informants have been aware of the study for quite long time for at least two reasons. Firstly, the researcher has been active participant of the community since 2010 which has included several discussions about the research. The author has written several blog entries about MeeGo community which have gained some attention in the community, both locally and internationally. The author's blog includes a static description of the author, which identifies author as a researcher. Secondly, current research topics have been discussed briefly also in several IRC channels and discussion lists. Clearly for these reasons it is unlikely that the informants would not be (at least somewhat) aware of the authors researcher status. To enable covert research, a new identity should have been build, which would have taken time and required the author to live a life of a 'double agent'.

4.2 Selecting interviewees

Going 'there' and participation in community activities both in local and international level functions also as a route to contact the suspicious developers who might otherwise be somewhat reluctant to participate or hard to reach. By seeing and observing the researcher, through off-topic discussions with the researcher and other community members, the cautious developer becomes more familiar with the researcher and with the aims of the research [9]. This enables arrangements for interview in separate occasions such as in MeeGo conferences, Summits and local meetups.

The primary interview population is developers. According to Germain “primary informants are those who are directly associated with the focus of the research in a major way” [14]. The interviewees of the primary group were found with the help of researcher's personal network, which has formed as a result of being an active member of MeeGo community. The networked nature of MeeGo developers was also utilized. Snowball effect is desired but not initial form of acquiring interviewees.

Yet some interviews resulted to propositions of other persons to be interviewed. The interviewed who did not identify themselves as MeeGo developers or were found through other developers are labeled as the secondary group. The secondary group includes such as end-users, marketing related roles and legal issue specialists.

4.3 Selecting events

Selecting events which are analyzed through the Communication Power theory by Castells are selected by using statistical methods and observation. Statistical methods are used to analyze important moments or turning points in MeeGo ecosystem. IRC discussion logs (for example in #meego and #meego-dev channels), different mailing list discussions (for example meego and meego-dev) and IRC meeting logs and archives are analyzed while the research proceeds. By using statistical methods, candidate important moments can be identified[1]. From these candidate moments in time, the important and most fruitful ones are selected by the researcher. This is possible because the researcher is 'living' with the community and 'inside' the ecosystem. At the moment selected events are not limited to include only certain type of situations such as crisis. This leaves some more room for selection and thus does not limit the research.

It is nearly impossible to identify suitable events before hand. So far one event has been identified. Nokia's decision to minimize MeeGo's role in their strategy has affected whole ecosystem. Interviews regarding that event has started in May 2011. The following events to be selected are still unknown. Intention is to identify maybe two or three more events for interviews during the following year and a half.

After the moments are identified, researcher can read all the discussions around that moment and make the final decision whether that particular moment is 'interesting' enough for further analysis. Statistical tools and methods act as supporting method. To identify the moments worth researching, several ready-made tools exists, which can draw tables and figures from IRC logs and email archives. Some tools enable even connection map creation. I don't know are the tools still sophisticated enough or should I program my own. I really would love to see real-time analysis of: IRC discussions, email lists and forum discussions. For this research only, real-time analysis (statistical) tools testing and developing has been done already and integration is in progress. Intention is to combine my work with the tools and MeeGo community dashboard development. That would be the ultimate tool to observe MeeGo ecosystem or community.

5 Research themes and current status

Interviews will be theme based. Selected themes are:

1. **MeeGo as an open source ecosystem**
    - Structure
    - Ideal composition
    - Role of companies and community
    - Role of Linux Foundation?
2. **Power in MeeGo ecosystem (in your work)**
    - Who have most influence in MeeGo ecosystem?

[1] See figure 1 where selected events are drawn as 'stars'.

- Are there power groups? Are they stable?
- Openness and transparency of decision making process
- Problem solving and disputes
- How are new features & ideas brought up?
- Who makes and how the 'big' decisions about the directions MeeGo is heading?
- Communication – methods, amount, content.

5.1 Current status

Interviews began 3rd of May 2011. First strategic event around which interviews construct, is Nokia's decision to change hand-held OS strategy Feb 11th 2011. Initial interviewees were selected from finnish MeeGo community. Snowball effect has worked well and resulted to several new interview candidates. All of the interview candidates have agreed to participate. This might suggest that MeeGo community is maturing up, since it is interested about information produced about it. Each interview takes about an hour. Expected saturation is reached after 10 to 15 interviews, which is in about a month.

6 Conclusions

In this paper has been discussed how qualitative and descriptive research about the balance of power in an open source ecosystem will be conducted. Discussion about methodology included: selected observation method, interview themes, interviewee and target event selection process. Also current status of research was discussed briefly.

References

1. M. Apgar. Ethnography and Cognition. In R. M. Emerson (ed.), Contemporary field research: A collection of readings. 1983.
2. P.G. Capek, SP Frank, S. Gerdt, and D. Shields. A history of IBM's open-source
3. involvement and strategy. *IBM systems journal*, 44(2):249-257, 2005. ISSN 0018-8670.
4. Manuel. Castells. *Communication power*. Oxford University Press ; New York, 2009.
5. H.W. Chesbrough. *Open innovation: The new imperative for creating and profiting from technology*. Harvard Business Press, 2003. ISBN 1578518377.

6. H.W. Chesbrough, W. Vanhaverbeke, and J. West. *Open innovation: Researching a new paradigm*. Oxford University Press, USA, 2006. ISBN 0199290725.
7. J.F. Christensen, M.H. Olesen, and J.S. Kjær. The industrial dynamics of Open Innovation - Evidence from the transformation of consumer electronics. *Research Policy*, 34(10):1533#1549, 2005. ISSN 0048-7333.
8. M. Crang and I. Cook. *Doing ethnographies*. Sage Publications Ltd, 2007.
9. N.K. Denzin and Y.S. Lincoln. *The Sage handbook of qualitative research*. Sage Publications, Inc, 2005.
10. R.M. Emerson. *Contemporary field research: A collection of readings*. Little Brown and Company, 1983.
11. B. Fitzgerald. The transformation of open source software. *Mis Quarterly*, 30 (3):587-598, 2006.
12. O. Gassmann and E. Enkel. Towards a theory of open innovation: three core process archetypes. In *R&D Management Conference*, 2004.
13. M. Genzuk. *A synthesis of ethnographic research*. 2003. Southern California: University of Southern California Center for Multilingual, Multicultural Research, 2003.
14. C.P. Germain. Ethnography: *Nursing research: A qualitative. perspective*, pages 237- 312, 1993.
15. R.A. Ghosh. Understanding free software developers: Findings from the FLOSS study. *Perspectives on free and open source software*, pages 23-45, 2005.
16. E. Goffman. On Fieldwork. *Journal of Contemporary Ethnography*, 18(2):123, 1989.
17. J. Henkel. Selective revealing in open innovation processes: The case of embedded Linux. *Research policy*, 35(7):953- 969, 2006. ISSN 0048-7333.
18. S.M. Katz. Etnographic research in l.j. gurak & m.m. lay (ed) research in technical communication, 2002.
19. R. Kirschbaum. Open innovation in practice. *Research-Technology Management*, 48(4):24-28, 2005. ISSN 0895-6308.
20. R.V. Kozinets. *Netnography: doing ethnographic research online*. Sage Publications Ltd, 2009.
21. J. Gamalielsson, H. Gustavsson, R. Karlsson, C. Lennerholt, B. Lings, A. Mattsson, and E. Olsson. *Exploring health within In First International Workshop on Building Sustainable Open Source Communities* (OSCOMM 2009), Sk ovde, Sweden, 2009.
22. C. Mann and F. Stewart. *in Kozinets Netnography: doing ethnographic research* . Sage Publications Ltd, 2009.
23. D.G. Messerschmitt and C. Szyperski. *Software ecosystem: understanding an indispensable technology and industry*, volume 1. The MIT Press, 2005.
24. M.D. Myers and M. Newman. The qualitative interview in IS research: Examining the craft. *Information and Organization*, 17(1):2-26, 2007.
25. Y.S. Lincoln & E.G. Guba. Naturalistic inquiry. Beverly Hills, CA: Sage

Publications, Inc. 1985
26. K. Popp and R. Meyer. *Profit from Software Ecosystems: Business Models, Ecosystems and Partnerships in the SoftwareIndustry*. BoD-Books on Demand, 2010. ISBN 3842300514.
27. K.M. Popp. Goals of Software Vendors for Partner Ecosystems#A Practitioners View. 2010.
28. E.S. Raymond. *The cathedral and the bazaar: musings on Linux and open source by an accidental revolutionary*. O'Reilly & Associates, Inc. Sebastopol, CA, USA, 2001.
29. C.M. Schweik. *The Open Source Software Ecosystem.* 2009. http://www.umass.edu/digitalcenter/research/working_papers/09_002SchweikEcosystem.pdf.
30. C. Seale. Quality in qualitative research. *Qualitative inquiry*, 5(4):465-468, 1999.
31. R.M. Stallman, J. Gay, and L. Lessig. *Free software, free society*. Gnu Press, 2002. ISBN 1882114981.
32. A.L. Strauss and J.M. Corbin. *Grounded theory in practice*. SAGE Publications, Inc, 1997.
33. A.L. Strauss and J.M. Corbin. *Basics of qualitative research: Techniques and procedures for developing grounded theory*. Sage Publications, Inc, 1998.
34. L. Torvalds and D. Diamond. *Just for fun: The story of an accidental revolutionary*. Harper Paperbacks, 2002.

Understanding Code Forking in Open Source Software

Linus Nyman
Hanken School of Economics
Arkadiankatu 22, FI-00100 Helsinki, Finland
linus.nyman@hanken.fi

Abstract. The right to fork code is one of the central rights of open source software. Among the implications of this right, which can be used by anyone, are that entire codebases can be forked and used in the creation of new programs; or, if someone is not pleased with how a project is being managed, they can fork a new version of the project. Despite its significance, code forking in open source has seen little study. My doctoral thesis strives to increase our understanding of code forking in open source software. The two main areas of interest are 1) the motivations behind code forking: what reasons do developers have for forking? What are the main categories of reasons, and how common are they? And, 2) how is forking perceived: when is it acceptable to fork a program and when is it not? Are these perceptions tied to the forks ramifications for the community? In addition to these questions, my thesis seeks to explore the problematic issue of defining a fork and present a typology of forks for use in academic study of the phenomenon.

1 Introduction

During the past decade researchers have looked at many both significant as well as interesting aspects of open source software; however, code forking is among the topics in which there are still many gaps in our knowledge. A deeper analysis of code forking is significant not only due to its previous scarcity of study, but also due to the central role code forking – as well as the possibility of code forking – plays in open source software. Code forking is at the same time both the potential saviour and downfall of an open source project: while a fork may dilute the efforts put into a project, the potential to fork also insures that an open source project will never die as long as there is a community with an interest in keeping the project alive. Indeed, the fork is arguably a large part of what makes much open source software possible.

1.1 Purpose and aim

This paper outlines the purpose and approach of my doctoral thesis. The purpose of the thesis is to increase our knowledge about code forking in open source software. Based on this overarching goal, two specific sub-questions have been defined:

1. *What are the motivations behind forking, i.e. what reasons do developers have for forking a program?* A common perception appears to be that forks are largely driven by disagreements among the community; however, is the truth quite so one-sided?
2. *How is forking perceived – when is forking acceptable and when is it not?* Considering the importance of community participation in open source projects, perceptions of a fork is a central question. A further, related area of interest in this question is what affect a fork has on a community: this would seem to be causally linked with how the fork itself is perceived.

This paper is structured as follows: Section 2 describes and discusses code forking as well as the related concepts of code reuse, code fragmentation, software distributions, and, finally, software licenses, while Section 3 is an overview of my research approach, including a discussion on some of the relevant findings of my research to date.

2 Code forking in open source software

Open source software development can be seen as a "private-collective" innovation: open source project participants privately invest time and resources to create new code which they then make available to all as a public good [1]. The open source initiative (OSI) offers ten criteria which must be met in order for a program to be considered open source. The first three criteria cover the right to free redistribution, the necessity of the inclusion of a program's source code, and the allowing of modifications and derived works[1]. Open source software, then, is licensed in a way which gives users the right to not only use the program free of charge, but also to access its source code. Furthermore, users are stipulated the right to both change the source code as well as incorporate the code into other programs. One of the results of these rights is that open source programmers do not have to reinvent the wheel: if a program or section of code exists which fits their needs they are free to incorporate it into the software they are working on.

The success story that is the Linux operating system was made possible in no small part by these very rights. Operating systems, which enable users to interact with the computer, consist of a number of different components (among them the

[1] The full list is available at http://www.opensource.org/docs/osd

kernel, memory management, input/output, file management, and a user interface). Linus Torvalds had developed an operating system kernel; an important piece of an operating system but not the entire puzzle in itself. The GNU Project, started in 1984 by Richard Stallman[2], had completed much of what was needed for an operating system but did not have a kernel. Because of the nature of open source licenses Torvalds was able to combine these elements into the Linux operating system, also known as GNU/Linux. This same right has since been used by many other programmers, and there are now hundreds of different versions (also called distributions or 'distros') of Linux.

The right to reuse existing code takes many forms, not all of which are easy to distinguish from one another. The most central concepts are those of code reuse, code fragmentation (including different distributions of a program), and code forking.

2.1. Code reuse, code fragmentation, and code forking in open source software

Code reuse, using existing software in the construction of a new software program (often in the form of reusing software components), has long been a source of both academic and practitioner interest. During the late 1960s both the use of software components and code reuse were proposed as a solution to the problem of building large, reliable software systems in a controlled, cost-effective way [2, 3]. While early data has shown that capitalizing on the promise of code reuse has proven more challenging than perhaps originally thought [4], code reuse nevertheless remains a common practice in open source programming [5].

Code forking is a somewhat more recent concept[3] and one tied even more strongly to open source software. Fogel [6] identifies two different types of forks: one group due to amicable but irreconcilable disagreements and interpersonal conflicts about the direction of the project, the second – and, as the author notes, perhaps more common group – due to both technical disagreements and interpersonal conflicts; however, it is not always possible to tell the difference between the two types. The most obvious form of forking takes place when a program splits into two versions, due to a disagreement among developers, with the original code serving as the basis for the new version of the program.

More common than code forking is *code fragmentation*, where different versions or distros (distributions) of a program emerge. Raymond [7] sees the actions of the developer community as well as the compatibility of new code to be central issues in differentiating code forking from code fragmentation. He calls different distributions of a program 'pseudo-forks', noting that they look like forks but are not perceived as such since they 'can benefit from each other's development efforts completely enough that they are neither technically nor sociologically a waste'. Moody [8] mirrors Raymond's sentiments, pointing out that code fragmentation does not

[2] http://www.gnu.org/gnu/thegnuproject.html, accessed March 2, 2011

[3] Arguably the first big cases of open source software code forking were the variants of AT&Ts UNIX in the 1970s.

traditionally lead to a split in the community and is thus considered less of a concern than a fork of the same program. These sentiments both echo a distinction made by Fogel [6]: it is not the existence of a fork which hurts a project, but rather the loss of developers and users.

Code forking has seen little discussion in the academic literature, perhaps due to negative connotations associated with the word. Forking is often perceived to be a threat, the outcome of a community failure. These kinds of connotations may lead to reluctance to use the term. However, code forking – or at the very least the option to fork – is a vital part of open source, and something which ensures its survival. A deeper analysis and understanding of code forking is important because of the central role that code forking – as well as the possibility of forking – plays in open source software. Fogel [6] considers the potential to fork a program to be 'the indispensable ingredient that binds developers together', noting that since a fork is bad for everyone, the more serious the threat of a fork the more willing people are to compromise in order to avoid it. The potential to fork is also a strong element in the governance of open source programs, as no one has what Fogel calls a 'magical hold' over any project since anyone can fork any project at any time.

Both Weber [9] and Fogel [6] discuss the concept of forks as being healthy for the ecosystem in a 'survival of the fittest' sense: the best code will survive. However, they also note that while a fork may benefit the ecosystem, it is likely to be harmful for the individual project.

Another source of concern is the "hijacking" of code, which means that a commercial vendor attempts to privatize the source code [10]. However, and perhaps somewhat paradoxically, the potential to fork any open source code also ensures the possibility of survival for any project. As Moody [11] points out, open source companies and the open source community differ substantially in that companies can be bought and sold but the community cannot. If the community disapproves of the actions of an open source company, whether due to attempts to privatize the source code or other reasons related to an open source program, the open source community can simply fork the software from the last open version and continue working in whichever direction they choose. (A recent example of such an occurrence was OpenOffice.org, a trademark owned by Oracle, which was forked into LibreOffice[4].)

A point worth mentioning when discussing the consequences of forking is that, even in the case of an actual split of the developer and user base, a fork can potentially offer some benefit to the program. In a situation in which a programmer would be interested in working on a program, but be reluctant to work with a specific person or team working on the same project, a fork would solve such a problem. Also, given that a fork is kept under an open source license, anything the forked version of a program develops, the original program can reuse – i.e. incorporate into the original version of the program. Sometimes the forked versions either merge back together, or the fork becomes so popular among both developers and users that it becomes the new de facto main version[5].

[4] http://www.documentfoundation.org/faq/

[5] As was the case with EGCC which forked from GCC.

In addition to concerns regarding the dilution of the workforce, another concern regarding program forks is their compatibility with other programs, i.e. the ability of existing programs to "port" into the new program. As Meeker [12] notes:

> In the open source world, everyone has the unfettered right to change and fork a code base, but people tend not to. Although they possess the right, they forgo it voluntarily. Lack of standardization has obvious practical problems. If there are 500 [incompatible] versions of Linux, no one will write applications for it. So there is at the same time a legal freedom to fork and a social pressure to avoid forking.

Meeker's views are mirrored by Raymond [7], who notes that while open source licenses arguably encourage both forking and pseudo-forking, it is only pseudo-forking (i.e. different distributions of a program) which is common; due to the strong social pressure against forking projects it is rarely done.

Existing literature is limited as far as offering a clear distinction of a fork is concerned. The practice of differentiating between a forked and a fragmented code is not necessarily a clear one unless defined by elements outside of the actual use of the code itself, as for instance differentiating between the two by looking at their affect on the community of developers rather than the use, or movement of, the code.

2.2 Open Source Licenses

The right to fork is stipulated in the program license. Open source licenses can be divided into two groups: copyleft (also called hereditary, or viral) and permissive. The main unifying feature of the copyleft licenses is that certain obligations must "run with" the license much like an easement would run with the rights in land. Copyleft licenses are often divided into strong and weak copyleft. Strong copyleft licenses stipulate that any derivative works, including code forks, must be licensed under the same license. In practice this means that if a program is licensed under a strong copyleft license, while one is guaranteed the right to fork the program, one cannot re-license it under either a permissive or proprietary (i.e. commercial) license. A license can be considered a weak copyleft license if not all derived works need inherit the license. The permissive licenses, on the other hand, allow forked versions of the program to be re-licensed under a hereditary or, depending on the license, even under a proprietary license.

Due to the differences in licenses the question of license compatibility becomes significant when combining open source software components in the same software system. As an example, programs licensed under a strong copyleft license cannot generally be combined with programs licensed under a permissive license. A practice used to navigate this issue is that software components with different licenses are isolated from one another through architectural design, for instance by adding an additional program, which is compliant with both licenses, in-between the non-compliant licenses, which can then communicate information between the two [13].

While there are thousands of different open source licenses, the half dozen most common licenses account for over 80 % of all license use [14]. The most commonly

used open source license is the strong copyleft GNU General Public Licenses, commonly known by the abbreviation GPL; a license used by roughly half all open source programs (ibid.). The GNU Lesser general Public License, or LGPL, is the most common of the weak copyleft licenses. Among the most popular permissive licenses are the Apache, BSD, and MIT licenses. For a more in-depth discussion of open source licenses see, for instance, Meeker [12] or McGowan [15]. For a discussion of open source legality patterns and architectural design see Hammouda et al. [13].

2.3 Previous research

Over the past decade the open source software phenomenon has attracted a growing number of researchers. Von Krogh and von Hippel [16] reviewed existing research on the open source phenomenon, noting that it can be categorized into three areas of study: motivations of open source contributors; governance, organization, and the process of innovation in open source software projects; and competitive dynamics enforced by open source software.

Open source licenses have received interest particularly among legal scholars and practitioners [17], who have addressed such questions as the enforceability of the GPL under existing copyright and contract law and what role the law plays in enabling the production of open source software (ibid.), what the legal implications of open source are (e.g. [15]), and what the risks and opportunities of open source software are (e.g. [12]).

Aksulu and Wade [18] conducted a comprehensive literature review of the research done into open source during its first ten plus years, dividing the 1 355 peer-reviewed articles they found into seven categories: conceptual, performance metrics, legal and regulatory, OSS production, OSS applications, OSS diffusion, and “beyond software”. They note that research into open source software license types has focused on: questions relating to terms and risks as well as determinants of license selection, consideration of the potential impacts of license choice on activity and the success of the OSS project, and deliberation of steps that can be taken to ensure license compliance and related infringement risks.

3. Research approach

The overarching goal of deepening our understanding of code forking is divided into two sub questions: 1) what are the motivations behind forking, i.e. what reasons do developers have for forking a program; and, 2) How is forking perceived: when is forking acceptable and when is it not? To answer these research questions I will use both qualitative and quantitative methods. The thesis will be article based.

Answers to the first Question, identifying the motivations behind forking, is sought in (at least) two different ways: firstly, using software databases like SourceForge to gain information about a large sample of code forks; and, secondly, through a survey. The issue of motivations is both interesting and significant. What

reasons do developers have for forking? Certainly a common perception appears to be that forks are largely driven by disagreements among the community; however, is the truth quite that simple? As the findings from one of the first articles of my thesis, a recent paper by this author and Tommi Mikkonen [19] indicate, forking in practice is much more practical and pragmatic then is commonly perceived, with disagreements among developers consisting of only a fraction of the motivations behind forking.

In another paper[6] I analyze 451 cases of code forking, with a total of 491 cases of license use, among programs registered on the SourceForge.net database during 2000-2010. The license use of forked programs is compared to that of a sample representing the entire population of open source programs. A Chi squared probability value of <0.001 for the findings strongly suggest that forked programs are more likely than non-forked programs to be licensed under the GPL. Overall there is significantly less use of permissive licenses than hereditary licenses among forked programs. The findings imply that programs under permissive licenses, when forked, gravitate towards the GPL. These results speak to one motivating factor being a protecting of the code through forking programs licensed under permissive licenses and re-licensing them under the GPL. While further study is needed to confirm this, there is evidence from the study by this author and Mikkonen that one of the (less common) reasons for forking a program is, indeed, license-related. The spectrum of motivations behind forks, however, appears to be a broad one, and further study is needed to understand it in greater detail.

As the information available on sites such as SourceForge is limited, once the initial study is completed and the results analyzed, a survey will be put together and sent to programmers who have forked a program. This will enable both a deepening of the understanding gained from SourceForge, as well as make possible a further inspection of interesting aspects which emerge in the data from the first articles.

In order to answer the second question, deepening our understanding of perceptions of forking by answering when forking is considered acceptable and when it isn't, I will conduct semi-structured interviews. It is also significant to note if there is a difference in opinion about forking among different groups; therefore, interviewees will be chosen from both corporations and from the open source community. Through these interviews I will also seek to answer the related question of what effect, or what different kinds of effect, a fork has on a community.

An issue which has surfaced during my work on this thesis so far has been the problematic nature of defining a fork compared to, for instance, a quasi-fork, fragmentation, or distribution. One of the added goals which a thesis on code forking necessitates is, thus, to propose a definition of how to define a code fork[7].

---

[6] "License to fork: a study of license use in open source software code forks from 2000-2010". At the time of writing, this paper is in review.

[7] While I have a completed rough draft of an article which puts forth such a typology, my goal is to rework and rewrite it after I have had the chance to conduct the other research necessary for my thesis, and thus deepened my understanding of the topic. The article, however, is not so much a goal of my thesis as it is a necessary step in order to achieve the aim of the thesis.

References

1. von Krogh and von Hippel (2003). *Open Source Software and the "Private-Collective" innovation model: Issues for Organization Science*. Organization Science, Vol. 14, No. 2, pp. 209-223.
2. Naur, P. and Randell, B. (1969) Software Engineering, Report on a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October 1968, Scientific Affairs Division, NATO, Brussels, 138-155. Available at: http://www.cs.dartmouth.edu/~doug/components.txt, accessed 4 March, 2011.
3. Krueger (1992). Software Reuse. *ACM Computing Surveys*, Vol. 24, No. 2, pp. 131-183.
4. Lynex and Layzell (1998). Organizational considerations for software reuse. *Annals of Software Engineering*. 5, pp. 105-124.
5. Haefliger, von Krogh, and Spaeth (2008). Code Reuse in Open Source Software. Management Science, Vol. 54, No. 1, pp. 180-193.
6. Fogel, K. (2006) Producing Open Source Software. O'Reilly, Sebastopol, CA.
7. Raymond (2001) The Cathedral & the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary. O'Reilly, Sebastopol, CA. Available at: http://catb.org/~esr/writings/homesteading/, last checked 21 November, 2011.
8. Moody (2011) The Deeper Significance of LibreOffice 3.3. ComputerWorld UK, 28 January.
9. Weber (2004) The Success of Open Source. Harvard University Press, Cambridge, Massachusetts and London, England.
10. Lerner and Tirole (2002) Some Simple Economics of Open Source. The Journal of Industrial Economics, Vol. 50, No. 2, pp. 197-234.
11. Moody (2009). Who owns commercial open source and can forks work? *Linux Journal*, Apr 23. Available at: http://www.linuxjournal.com/content/who-owns-commercial-open-source-%E2%80%93-and-can-forks-work, accessed 24 February, 2010.
12. Meeker (2008) The Open Source Alternative: Understanding Risks and Leveraging Opportunities. Wiley, Hoboken, N.Y.
13. Hammouda, Mikkonen, Oksanen and Jaaksi (2010) Open Source Legality Patterns: Architectural Design Decisions Motivated by Legal Concerns. Published in the proceedings of the 14th International Academic MindTrek Conference: Envisioning Future Media Environments. ACM, New York, NY.
14. Black Duck Software, "Top 20 Most Commonly Used Licenses in Open Source Projects", available at: http://www.blackducksoftware.com/oss/licenses, last checked 21 November, 2011.
15. McGowan (2005) Legal Aspects of Free and Open Source Software. In: J. Feller, B. Fitzgerald, S. Hissam, and K. Lakhani (Eds.), Perspectives on Open Source And Free Software. MIT Press, Cambridge, Massachusetts. Available at: http://mitpress.mit.edu/books/chapters/0262562278.pdf, accessed 7 March, 2011.
16. von Krogh and von Hippel (2006) The Promise of Research on Open Source Software. Management Science, Vol. 52, No 7, pp. 975-983.
17. McGowan (2000) The Legal Implications of Open-Source Software (Undated). Available at SSRN: http://ssrn.com/abstract=243237 or doi:10.2139/ssrn.243237, accessed 6 March, 2011.

18. Aksulu and Wade (2010) A Comprehensive Review and Synthesis of Open Source Research. Journal of the Association for Information Systems, Vol. 11, pp. 576-656.

19. Nyman, L. and Mikkonen, T. (2011) To Fork or Not to Fork: Fork: Motivations in SourceForge Projects. Proceedings of the 7th International Conference on Open Source Systems (OSS 2011), 259-268, Springer.

FLOSS QUALITY: DEFINITION, ANTECEDENTS, AND THE ROLE OF MODULARITY

Claudia Ruiz
Georgia State University, Computer Information Systems Department
35 Broad Street, Atlanta, GA, 30302, USA
cruiz5@gsu.edu

Abstract. After Linus's law was formulated, which says that, "given enough eyeballs, all bugs are shallow," it has been assumed that open source software has high quality. This assumed quality is a major factor when businesses and individuals decide to adopt open source software products. To determine if this assumed quality is real, this dissertation will answer the following questions: (1) what is quality in open source software? (2) What determines open source software quality? (3) What is the role of modularity in determining open source software quality? To answer these questions, a three-paper approach is to be undertaken. Moreover, this dissertation will contribute to knowledge because there is no single definition of open source software quality, there is no model to predict open source software quality, and the actual role of modularity in determining quality has not been addressed. Understanding open source software quality in a way that can be measured will help researchers to assess and predict the quality of open source software and practitioners to evaluate different open source software products for adoption decisions.

1 Introduction

Linus's law, named after Linus Torvalds, the architect of the Linux kernel, says that "Given enough eyeballs, all bugs are shallow" [1], meaning that public peer review was the reason for open source software's higher quality over traditionally developed commercial.

Higher quality is now considered a characteristic of open source software that is taken for granted, with firms citing open source software's quality as a motivation for adopting it in their organizations [2].

It would seem that having a public peer review with many people, including developers and users, would produce a software product with less bugs and defects than a closed review with few people, all of them developers. However, studies out to prove the higher quality of open source software over closed source software have produced mixed results [3-6].

There is no single definition of quality for open source software with most research referencing theories developed for traditional software development, which is different from open source software development in terms of actors, methodologies, and expectations.

Researchers have operationalized quality as independent and dependent variable; they have defined it using object-oriented software product design measures, number of defects, defect resolution rates, etc. Without a common definition of quality, research on the antecedents of open source quality is being limited [7].

Since quality is one of the major determinants of system success in traditionally developed closed source systems [8], understanding open source quality is the first necessary step to understand the success of open source systems.

In order to understand open source software quality, one has to take into account the emergent nature of open source software, the lack of a discrete separation of roles between developers and users, and the collaborative nature of its elaboration. Studies that have measured defect rates and bug resolution rates take into account the emergent and community aspects of open source software, with some measuring them in cumulative terms [9-13] (the defect rate for the project since inception, or for a set period of time) and others measuring them by release version [14-18].

Using the release version as the unit of analysis is important because open source software is emergent and evolves with each release. Therefore, studies that look at open source software quality from a cumulative perspective fail to capture its true nature.

Past research on open source software quality has had the following characteristics: differing definitions of quality, differing operationalizations of quality by using different types of measures, lack of antecedents of quality.

Table 1. FLOSS Quality Definitions

Approach	Definition	Measures	Metrics Counts	Articles Mapped
Product	Software product structure and characteristics	Cohesion Complexity Size/effort Issue/bug Change/patch/version Compliance Documentation Deployment Interoperability Object coupling End user UI experience Data access Licensing Testing Maintainability Modularity	158	[3, 4, 13, 17-22]
Community	Developer, contributor, and user characteristics and interactions	Adoption/usage Contributions Community member activity Social network analysis Community culture	115	[3, 5, 6, 13, 19-21, 23, 24]

		Project management structure Community demographics Project distribution and inclusion Problem report activity Evolution Documentation		
Process	Established and repeatable procedures set in place to minimize defects and simplify work.	Testing Planning Versioning/branching Budget Bug/issue tracking Meetings Quality review Methodology/process description, execution, and compliance Consulting services Group consensus General project management Training	66	[9, 11, 16, 20, 25-28]

These differing definitions of open source quality are not surprising given quality's subjective nature. From indescribable excellence all the way to conformance to specification and customer satisfaction, quality can be different things to different stakeholders. If quality is defined differently it will also be measured and evaluated differently.

Traditional methods of quality assessment cannot be used with open source software because they do not account for its differences and do not incorporate them into their calculations, producing an incomplete and inaccurate picture of the quality of the open source project.

1.1 Research Questions

This dissertation addresses the limitations in the literature by proposing and evaluating a model of open source software quality. In order to achieve this, the following questions will be addressed: (1) What is quality in open source software? (2) What determines open source software quality? (3) What is the role of modularity in open source software quality?

The first step in proposing a model of open source software quality is to first develop a definition of quality to be measured and evaluated. This is what the first question will address.

The second question will address the antecedents of quality. Quality here will be the dependent variable and the goal will be to discover which factors affect quality and how.

The third question will address how modularity affects quality. Modularity has been touted as the key to open source software quality because it prevents the bugs introduced by one collaborator to impact the rest of the product [29]. However, there are many definitions and measurements of modularity in the literature. To address this question, I will first collect the many definitions of modularity, select the one that most addressed the nature of open source software and will analyze to determine its relationship to quality.

2 Design and Overview of Research

This dissertation proposes to answer the above postulated questions by following this approach: (1) define open source quality, (2) determine the antecedents to open source quality, and finally, (3) determine what role modularity plays in quality.

Table 2. Overview of Research

	Study 1	Study 2	Study 3
Chapter in this dissertation	2	3	4
Approach	Literature review	Factor analysis	Factor analysis
Chapter's contribution	Provide an understanding of FLOSS quality and a definition.	Determine the impact of certain factors on FLOSS quality.	Determine the impact of modularity on FLOSS quality

2.1 Define Open Source Quality

This part of the dissertation presents a literature review of the open source software research that deals with quality. Specifically, this chapter sought to understand how the community of open source researchers defined, measured, and evaluated quality.

This study defined quality as defect density and defect resolution rate. It states that the unit of analysis needs to be the software release. FLOSS is a dynamically developed product, which depends on the community's commitment to quality, which is evolving. That is why quality not only needs a static measure (defect density) but a dynamic measure (defect resolution rate).

2.2 **Antecedents to Open Source Quality**

This part of the dissertation will test a set of constructs for their power to predict the FLOSS quality measures that were defined in the previous part of the dissertation.

The main constructs are defect fixes and enhancements introduced into a particular software version. These constructs were chosen because they represent the work items that go into a software development project.

Other constructs that were included to determine if they can predict quality are maturity, popularity, age, release development time, and active contribution rate. These constructs were chosen from the literature review performed in part one.

The hypotheses to be tested are the following:

H_1: The greater the number of enhancements introduced in a release, the greater the defect density of the release.

H_2: The greater the number of enhancements introduced in a release, the greater the defect resolution rate.

These hypotheses would seem to be common sense but in open source software, new features are contributed by core developers, while defect fixes are introduced by periphery developers [20]. It is necessary to determine the effect of these different types of contributions by two different types of contributors have on quality and to confirm that adding new features does in fact negatively affect defect fixing [12].

Defect fixes should reduce the number of defects in a release, but they could increase the count if they break other functionality in the process.

H_3: The greater the number of defect fixes introduced in a release, the lesser the defect density of the release.

H_4: The greater the number of defect fixes introduced in a release, the lesser the defect resolution rate.

Comparing FLOSS projects with differing characteristics has produced mixed results. Here, the three main factors of FLOSS success are tested for their effect on quality. Age is how old is project (from the date it was first registered), popularity is how often it has been downloaded, which gives it a high ranking in SourceForge lists, and maturity, which specifies how many releases have been produced.

Also, it is important to note, in addition to the project age, the time it took to develop a particular release. Shorter releases should include less defect fixes and enhancements, thus reducing the change of generating new defects.

H_5: The greater the age, popularity, and maturity of a project at the time of a release, the lesser the defect density.

H_6: The greater the age, popularity, and maturity of a project at the time of a release, the lesser the defect resolution rate.

H_7: The lesser the release development time for a release, the lesser the defect density.

H_8: The lesser the release development time for a release, the lesser the defect resolution rate.

There has been controversy regarding the effect of the number of developers involved in a FLOSS project [30]. An important distinction is that not all registered developers contribute to a release. The rate of active contribution, the percentage of the total registered developers who actually contributed to the release, will help determine how many people are actually working on the release, since it is postulated that software that has too many people working on it will have more defects [13].

H_9: The lower the rate of active contribution of a release, the lesser the defect density.

H_{10}: The lower rate of active contribution of a release, the lesser the defect resolution rate.

The Sourceforge repository at Notre Dame University [31] will be source of the data. A random sample of projects will be select within a given a software type using the same programming language. This will allow the collection of successful and popular projects as well as those that are less so, while controlling for software type and programming language. By controlling these variables, variability that could be introduced by development complexity and difficulty will be removed.

After the projects to be examined are selected, their corresponding releases will be considered. Projects with less than three stable releases will be removed from the sample because less than that would not allow for much data to analyze.

The data will be collected per project per release. In the table above, to calculate the quality of stable release$_1$ of project$_1$, the defect density and defect resolution rate will be calculated using data from Date$_1$ to Date$_2$.

This data collection reflects the emergent nature of open source software. When a version is released, it is not known what defects it has. It is necessary to go backwards from the next release in order to determine the quality of the former release.

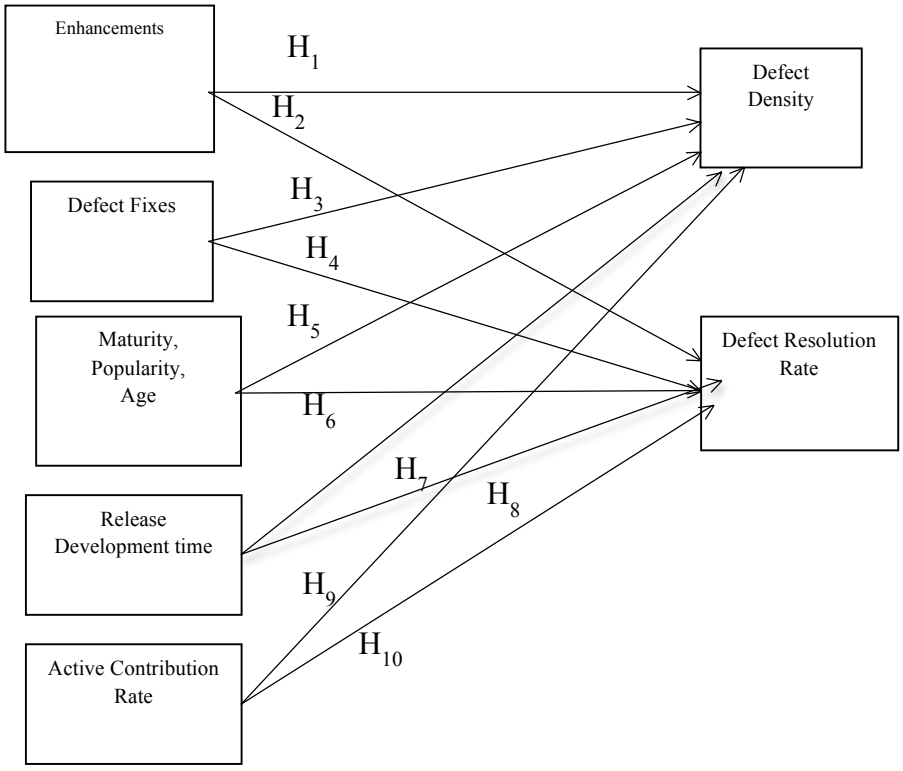

Fig. 1. Research Model

It could be argued that there is no benefit in knowing the defect count and the defect resolution rate of a previous version that only the latest version matters. However, given open source software's emergent nature, the pattern in the measures from release to release should reveal an improvement in defect resolution rate; defect count will vary depending on the number of enhancements introduced.

Project progress in in agile development is monitored using velocity rate and burndown charts [32]. Velocity can be used to estimate future completion rates based on past ones [33]. A pattern of increasing and then plateauing velocity will indicate a stable and cohesive team [33]. The burndown chart is a graphical representation of how a team is progressing against the estimated velocity along the spring (set development time.) A team will then add or remove backlog items depending on how it is meeting the estimate.

Backlog items are enhancements and defect fixes. The burndown rate will be higher than the estimated velocity if there are too many backlog items, while it will fall short of the estimated velocity if there are too few. The velocity and burndown chart provide a quick pulse check for the team progress: meeting or going below estimates means that progress is on track while going above the estimates means that a closer look needs to be taken to determine what is affecting the progress of the project.

A project's progress also depends on its efficient use of its human resources. The rate of developer engagement [34] has been used to determine the agility of FLOSS projects: failed projects will have a high rate of developer engagement in the beginning, but will fail to attract new developers and eventually phase out [34].

The table below contains a description of the constructs and their measures.

Table 3. Constructs and Measures

Construct	Description	Measure	Source

		Dependent Variables	
Defect Density	This will describe the quality of the software product. The lower the number, the better.	Number of defects added weighted by priority/KLOC added	[20]
Defect Resolution Rate	Measures the project community's commitment to quality. The lower, the better.	Average of the time it took to solve defects.	[5, 20]
		Independent Variables	
Enhancements	New functionality added to the project.	Number of enhancements included in release.	[12, 20]
Defect fixes	Code that changes existing code.	Number of bug fixes included in release.	[24, 35, 36]
Maturity	The stage of development the project is in. More mature projects have produced more releases than less mature ones.	Number of releases produced. Initiation – before first release (< 1). Intermediate Growth – after first release (1 < x < 3). Advanced Growth – at least three releases (> 3).	[18, 19, 23, 37, 38]
Popularity	Success of the project.	Number of downloads	[19, 23]
Age	Age of project.	Present date – Date registered	[38, 39]
Release development time	Time it took to make the release.	Release date – previous release date.	[13, 16]
Rate of active contribution	What percentage of all developers actually contributed to the release	Number of unique developers whose contributions were included in the release / Number of developer accounts	
	Number of developers associated with the project	Number of developer accounts	[17, 19]
	Number of developers who contributed to release.	Number of unique developers whose contributions were included in the release.	[18, 20, 35, 38-41]
		Controlling for	
Software type	Sample cases will be selected from the same product topic.		[5, 9, 13, 18, 28, 42-46]
Language	Sample cases selected will develop their product using the same programming language.		[18, 19, 38]

The contribution of this study will be to explain if and how certain factors affect quality in an open source setting. Quality is defined twofold, as the static defect density in a release and the evolutionary defect resolution rate.

The model here explained will predict quality for a FLOSS version, allowing for the analysis of quality within a product and among products.

2.3 Role of Modularity

This part of the dissertation will test the role of modularity in predicting FLOSS quality. This study uses a metric of modularity chosen from the literature review performed on part one. It then tests all possible models where modularity could predict quality. These models include direct, causal, moderating, and mediating.

Additional constructs will include enhancements and defect fixes added to a software release (the same ones from part three). If any constructs from part two prove to be significant in predicting quality, they will also be included in the analysis to determine the best relationship among constructs and modularity in predicting quality.

Modularity is used to manage system complexity by reducing the number of working parts [47]. Elements are hidden from the system when they are encapsulated [48] into a smaller number of units or modules. The modules can then be designed independently of each other but will work together as an integrated system when brought together [49].

Modularity can be achieved by implementing an architecture to establish their functions and membership in the system. Interfaces determine how the modules will interact, fit together, and communicate. They will also need to conform to design rules set up by standards, which are also used to assess module performance [49].

The highest level of modularity is realized by systems composed totally out of components. Components are modules that have weak coupling and strong cohesion, meaning that they are independent of each other and have minimal interaction with one another because each component groups functionality with high dependency [50].

A system's modules are said to be true components if they can be combined and reconfigured with other components (even with those from different systems) to create new systems. Components are designed for many uses, even unimagined ones, while modules are designed for a specific use and context [50].

Some FLOSS projects have implemented modularity with plug-in application program interface (API) architectural styles. 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 [51].

FLOSS interest in modularity can be traced back to the Linux project. Modularity in Linux was defined as having a core functionality (the kernel) that was separate from the features of the product, which live in modules that can be configured and compiled separately [20]. This means that the changes to the core would be more infrequent and carefully managed and tested because they would affect all components of the software. But the feature modules could be changed

more easily because they did not affect the core, thus allowing implementers to add them or remove them more easily.

However, studies on open source modularity have not measured it using the Linux definition. They have used object oriented measures such as coupling [14, 17-19, 21, 52], correlation between functions added and functions changed [4], function call dependencies among source files [53, 54], inheritance [13], and other measures which can be seen in table 4.

Table 4. Modularity Metrics

Metric	Explanation	Source
Coupling	Loosely coupled objects (those with low coupling scores) are considered more independent and thus more modular.	[14, 17-19, 21, 52]
Average component size	Smaller component size equals higher modularity.	[3]
Function call dependencies among files	References in a file to classes in other files. Few references equal higher modularity.	[53, 54]
Amount of commits performed by developers that contribute to at least two modules	In highly modular projects, this number will be lower.	[55]
Correlation between growing rate and changing rate	Correlation between functions added and functions modified.	[4]
Number of children (NOC) Depth in inheritance tree (DIT) Number of inherited methods overridden by a subclass (NORM)	Correct use of inheritance makes code more modular.	[13]
Number of directories into which the source code is divided.	One directory, two directory, more than two directory levels.	[56]
Number of subprojects	Number of subprojects with at least one task launched by a project.	[57]
Number of modules that can be separately compiled and configured.	Separately compiled and configured modules are considered to be independent.	[20]

The problem with using code structure metrics, which look at source files is that files within a module will be tightly coupled, and a module could be composed of several files. A module groups related and dependent features, as in the Linux project, into subprojects which are separate from the core functionality.

This separation into subprojects can also be seen in the projects hosted in the SourceForge repository. The subprojects are also built and versioned independently. Looking at the individually versioned and compiled subprojects within a project in

the SourceForge repository will provide a proxy that best captures how modularity is implemented in FLOSS projects.

Three models will be tested to determine modularity's role on open source quality. Modularity will be evaluated in the roles of mediator, moderator, and on its direct effect on quality.

If modularity has a direct effect on quality, the more modular a project, the fewer defects it will have and the faster those few defects will be fixed.

H_1: The greater the modularity of the project at the time of the release, the lower the defect density of the release.

H_2: The greater the modularity of the project at the time of the release, the lower the defect resolution rate of the release.

If modularity improves quality, it will result in a lower number of defect fixes being introduced, and increased growth of the project with more modules with more enhancements being introduced.

H_3: The greater the modularity, the greater the number of enhancements in a release.

H_4: The greater the modularity, the lesser the number of defect fixes in a release.

Modularity could be caused by more enhancements and fewer defect fixes being introduced, thus increasing the quality of the product by reducing the number of defects and the time it takes to fix them.

H_5: Modularity mediates the relationship between the number of enhancements in a release and their effect on the defect density and the defect resolution rate of that release.

H_6: Modularity mediates the relationship between the number of defect fixes in a release and their effect on the defect density and the defect resolution rate of that release.

Modularity could increase the quality of a product by moderating the effect of introducing enhancements and defect fixes into the product.

H_7: Modularity moderates the relationship between the number of enhancements in a release and their effect on the defect density and the defect resolution rate.

H_8: Modularity moderates the relationship between the number of defect fixes in a release and their effect on the defect density and the defect resolution rate.

This study will determine the role of modularity in affecting quality in open source software projects. It will explain if the best fitting model is shows a direct, causal, mediating, or moderating relationship.

Future research projects will look at the quality of the modularity implementation itself and whether certain approaches produce higher quality than others.

2.4 Empirical tests

The predictive models in part three and four have formulated hypotheses that will need to be tested using a factorial design. The factorial design is the best design for

these predictive models because it helps understand the effect of independent variables on dependent variables.

Chapter two uses qualitative grounded theory to understand and interpret how quality is defined and operationalized by the FLOSS research community.

Table 5. Empirical Tests

Chapter	Design	Level of analysis	Methods	Subjects
2	Qualitative text analysis	Research article	Literature review using grounded theory to code definitions and measures of quality.	FLOSS research community
3	Factorial	Software version	Quality measured by counting the number of defects, enhancements introduced in a software version from the time it was released going back to the date the previous version was released.	SourceForge projects from Notre Dame University repository
4	Factorial	Software version	Quality will be measured as above. Modularity will be measured as the number of individually versioned products in the project directory at the time the main software product version was released.	SourceForge projects from Notre Dame University repository

2.5 Data Analysis

Chapter two performed a grounded theory analysis of FLOSS quality literature and did not use any statistical methods. Chapters three and four are looking for the best predictive model and will use SEM (structured equation modeling) to find it. In SEM, the best fitting model will have the lowest chi square score.

The data to be analyzed will be collected from the SourceForge repository at Notre Dame University [31]. This repository contains monthly dumps of data from SourceForge. From this data, a sample will be selected of projects from the same category and written using the same programming language; this is necessary in order to control for complexity that might be inherent to a given programming language or a certain type of software product. The projects chosen will need to have at least two stable versions released in order to provide enough data to analyze.

Table 6. Data Analysis Approach

Chapter	Sample	Statistical Method	Analytical Test
2	40	N/A	N/A
3	500	SEM (structured equation modeling)	Tests whether the constructs can predict quality. The best fitting model will have the lowest Chi square score.
4	500	SEM (structured equation modeling	Tests whether the constructs can predict quality. The best fitting model will have the lowest Chi square score.

2.6 Threats to Validity

The greatest threat to validity in this dissertation is the limited generalizability of the study. The sample is taken from one repository (SourceForge.net) and is controlled for product category and programming language used. By doing this, the internal validity will be high because the cases will be similar to each other and issues such as complexity due to product category and programming language are eliminated. The problem with this approach is that the results will only be generalizable to FLOSS projects of the same category and programming language as the sample.

This dissertation is seen as a starting point in a longer research stream. Whatever is learned in this study will be applied to future research that will include FLOSS projects of different categories and programming languages.

Table 7. Threats to Validity

Type of threat	Effect	Countermeasures
Selection	Differences in cases might be responsible for the effect found. Projects hosted in SourceForge.net might be different from projects hosted by other environments.	Cases will be selected from the same product category and using the same programming language. SourceForge.net hosts the largest number of projects and is the most popular hosting environment.
Measurement	Statistics and metrics may not be reliable reflections of the phenomenon.	Project quality measures are operationalizations of the concepts theorized and taken from literature and previous research.
Mortality	The data source is archival so there is no risk of participants dropping out.	Not necessary.
External validity	May only be generalizable to FLOSS projects of a certain category and written in a certain programming language, hosted by	This is a limitation of this research and will need to be addressed by future research extending what is learned here

SourceForge.net.	into other project categories, programming languages, and repositories.

3 Contributions to Knowledge and Practice

This research aims to develop a theory of FLOSS quality by identifying the factors that predict the quality of a FLOSS product. The theory will also contribute a measurable definition of FLOSS product quality and modularity.

These contributions are significant because there is single definition of FLOSS quality, there is no model to predict FLOSS quality, and the actual role of modularity in determining quality has not been addressed.

Understanding FLOSS quality in a way that can be measured will help practitioners to evaluate different FLOSS projects in order to decide which ones to integrate into their environment.

4 References

[1] E. Raymond, "The cathedral and the bazaar," *Knowledge, Technology, and Policy,* vol. 12, pp. 23-49, 1999.

[2] A. Bonaccorsi and C. Rossi, "Comparing Motivations of Individual Programmers and Firms to Take Part in the Open Source Movement," *Knowledge, Technology and Policy,* vol. 18, pp. 40-64, 2006.

[3] I. Stamelos, L. Angelis, A. Oikonomou, and G. L. Bleris, "Code Quality Analysis in Open Source Software Development," *Information Systems Journal,* vol. 12, pp. 43-60, 2002.

[4] J. W. Paulson, G. Succi, and A. Eberlein, "An empirical study of open-source and closed-source software products," *Software Engineering, IEEE Transactions on,* vol. 30, pp. 246-256, 2004.

[5] J. Kuan, "Open Source Software as Lead-User's Make or Buy Decision: A Study of Open and Closed Source Quality," presented at the Second Conference on The Economics of the Software and Internet Industries, 2003.

[6] S. Raghunathan, A. Prasad, B. K. Mishra, and H. Chang;, "Open source versus closed source: software quality in monopoly and competitive markets," *IEEE Transactions on Systems, Man and Cybernetics, Part A, ,* vol. 35, pp. 903-918, November 2005.

[7] K. Crowston, K. Wei, J. Howison, and A. Wiggins, "Free/Libre Open Source Software Development: What We Know and What We Do Not Know," *ACM Computing Surveys,* vol. 44, 2012.

[8] 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.

[9] K. Crowston and B. Scozzi, "Bug fixing practices within free/libre open source software development teams," *Journal of Database Management,* vol. 19, pp. 1-30, 2008.

[10] C. L. Huntley, "Organizational learning in open-source software projects: an analysis of debugging data," *Engineering Management, IEEE Transactions on,* vol. 50, pp. 485-493, 2003.

[11] T. J. Halloran and W. L. Scherlis, "High Quality and Open Source Software Practices," presented at the Proceedings of the 2nd Workshop on Open Source Software Engineering (ICSE 2002), Orlando, FL, USA, 2002.

[12] Y. Kidane and P. Gloor, "Correlating temporal communication patterns of the Eclipse open source community with performance and creativity," *Computational & Mathematical Organization Theory,* vol. 13, pp. 17-27, 2007.

[13] S. Koch and C. Neumann, "Exploring the Effects of Process Characteristics on Product Quality in Open Source Software Development," *Journal of Database Management,* vol. 19, pp. 31-57, 2008.

[14] T. Gyimothy, R. Ferenc, and I. Siket, "Empirical validation of object-oriented metrics on open source software for fault prediction," *Software Engineering, IEEE Transactions on,* vol. 31, pp. 897-910, 2005.

[15] D. G. Glance, "Release Criteria for the Linux Kernel," *First Monday,* vol. 9, 5 April 2004.

[16] L. Zhao and S. Elbaum, "Quality assurance under the open source development model," *Journal of Systems and Software,* vol. 66, pp. 65-75, 2003.

[17] C. A. Conley, "Design for quality: The case of Open Source Software Development," PhD, Stern Graduate School of Business Administration, New York University, New York, NY, USA, 2008.

[18] E. Capra, C. Francalanci, and F. Merlo, "An Empirical Study on the Relationship among Software Design Quality, Development Effort, and Governance in Open Source Projects," *IEEE Transactions on Software Engineering,* vol. 34, pp. 765-782, 2008.

[19] D. Barbagallo, C. Francalenei, and F. Merlo, "The Impact of Social Networking on Software Design Quality and Development Effort in Open Source Projects," presented at the Proceedings of the International Conference on Information Systems, 2008.

[20] A. Mockus, R. T. Fielding, and J. Herbsleb, "A Case Study of Open Source Software Development: The Apache Server," presented at the Proceedings of the 22nd International Conference on Software Engineering (ICSE), 2000.

[21] I. Samoladas, G. Gousios, D. Spinellis, and I. Stamelos, "The SQO-OSS Quality Model: Measurement Based Open Source Software Evaluation," presented at the 4th International Conference on Open Source Systems (OSS2008), Milan, Italy, 2008.

[22] N. Tsantalis and A. Chatzigeorgiou, "Identification of Move Method Refactoring Opportunities," *Software Engineering, IEEE Transactions on,* vol. 35, pp. 347-367, 2009.

[23] K. Crowston, J. Howison, and H. Annabi, "Information Systems Success in Free and Open Source Software Development: Theory and Measures," *Software Process: Improvement and Practice,* vol. 11, pp. 123-148, 2006.

[24] A. H. Ghapanchi and A. Aurum, "Measuring the Effectiveness of the Defect-Fixing Process in Open Source Software Projects," presented at the Proceedings of the 44th Hawaii International Conference on System Sciences, Hawaii, USA, 2011.

[25] M. Michlmayr, F. Hunt, and D. Probert, "Quality Practices and Problems in Free Software Projects," presented at the Proceedings of the First International Conference on Open Source Systems, Genova, Italy, 2005.

[26] P. C. Rigby, D. M. German, and M.-A. Storey, "Open source software peer review practices: a case study of the apache server," presented at the 0th International Conference on Software Engineering (ICSE2008), Leipzig, Germany, 2008.

[27] M. Aberdour, "Achieving Quality in Open Source Software," *IEEE Software,* vol. 24, pp. 58-64, 2007.

[28] A. G. Koru and J. Tian, "Defect handling in medium and large open source projects," *Software, IEEE,* vol. 21, pp. 54-61, 2004.

[29] L. Torvalds, "The Linux Edge," in *Open-Sources: Voices from the Open-Source Revolution*, C. DiBona, S. Ockman, and M. Stone, Eds., ed Sebastopol, CA: O'Reilly & Associates, Inc., 1999, pp. 101-111.

[30] C. M. Schweik, R. C. English, M. Kitsing, and S. Haire, "Brooks' Versus Linus' Law: An Empirical Test of Open Source Projects," presented at the Proceedings of the 2008 international conference on Digital government research, Montreal, Canada, 2008.

[31] Y. Gao, M. Van Antwerp, S. Christley, and G. Madey, "A Research Collaboratory for Open Source Software Research," in *Proceedings of the 29th International Conference on Software Enginering + Workshops (ICSE-ICSE Workshops 2007), International Workshop on Emerging Trends in FLOSS Research and Development (FLOSS 2007)*, Minneapolis, MN, USA, 2007.

[32] M. Cohn, *Agile Estimation and Planning*: Prentice Hall, 2006.

[33] D. Hartmann and R. Dymond, "Appropriate Agile Measurement: Using Metrics and Diagnostics to Deliver Business Value," in *Proceedings of the conference on AGILE*, Washington, DC, USA, 2006, pp. 126-131.

[34] P. Adams, A. Capiluppi, and A. de Groot, "Detecting Agility of Open Source Projects Through Developer Engagement," in *Open Source*

Development, Communities and Quality. vol. 275, B. Russo, E. Damiani, S. Hissam, B. r. Lundell, and G. Succi, Eds., ed: Springer Boston, 2008, pp. 333-341.

[35] E. Petrinja, A. Sillitti, and G. Succi, "Comparing OpenBRR, QSOS, and OMM Assessment Models," in *6th International Conference on Open Source Systems (OSS2010)*, Notre Dame, IN, USA, 2010, pp. 224-238.

[36] V. del Bianco, L. Lavazza, S. Morasca, D. Taibi, and D. Tosi, "The QualiSPo approach to OSS product quality evaluation," presented at the Proceedings of the 3rd International Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development (FLOSS '10), Cape Town, South Africa, 2010.

[37] C. M. Schweik and R. English, "Identifying Success and Abandonment of Free/Libre and Open Source (FLOSS) Commons: A Preliminary Classification of Sourceforge.net projects," *Upgrade,* vol. 8, December 2007.

[38] I. Chengalur-Smith, A. Sidorova, and S. L. Daniel, "Sustainability of Free/Libre Open Source Projects: A Longitudinal Study," *Journal of the Association for Information Systems,* vol. 11, pp. 657-683, 2010.

[39] J.-c. Deprez, F. F. Monfils, M. Ciolkowski, and M. Soto, "Defining Software Evolvability from a Free/Open-Source Software Perspective," presented at the Third International IEEE Workshop on Software Evolvability, Paris, France, 2007.

[40] J.-C. Deprez and S. Alexandre, "Comparing Assessment Methodologies for Free/Open Source Software: OpenBRR and QSOS," presented at the PROFES 2008, 2008.

[41] R. Glott, A.-K. Groven, K. Haaland, and A. Tannenberg, "Quality Models for Free/Libre Open Source Software--Towards the "Silver Bullet"?," presented at the 36th EUROMICRO Conference on Software Engineering and Advanced Applications, Lille, France, 2010.

[42] Y. A. Au, D. Carpenter, X. Chen, and J. G. Clark, "Virtual organizational learning in open source software development projects," *Information & Management,* vol. 46, pp. 9-15, 2009.

[43] B. Wray and R. Mathieu, "Evaluating the performance of open source software projects using data envelopment analysis," *Information Management & Computer Security,* vol. 16, p. 449, 2008.

[44] A. G. Koru and H. Liu, "Identifying and characterizing change-prone classes in two large-scale open-source products," *Journal of Systems and Software,* vol. 80, pp. 63-73, 2007.

[45] L. Yu, S. R. Schach, K. Chen, G. Z. Heller, and J. Offutt, "Maintainability of the kernels of open-source operating systems: A comparison of Linux with FreeBSD, NetBSD, and OpenBSD," *Journal of Systems and Software,* vol. 79, pp. 807-815, 2006.

[46] I. Samoladas, I. Stamelos, L. Angelis, and A. Oikonomou, "Open source software development should strive for even greater code maintainability," *Commun. ACM,* vol. 47, pp. 83-87, 2004.

[47] R. N. Langlois, "Modularity in technology and organization," *Journal of Economic Behavior & Organization,* vol. 49, pp. 19-37, 2002.

[48] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Commun. ACM,* vol. 15, pp. 1053-1058, 1972.

[49] C. Y. Baldwin and K. B. Clark, "Managing in an age of modularity," *Harvard Business Review,* vol. 75, pp. 84-93, 1997.

[50] D. Messerschmitt, "Rethinking Components: From Hardware and Software to Systems," *Proceedings of the IEEE,* vol. 95, pp. 1473-1496, 2007.

[51] W. Scacchi and C. Jensen, "Governance in Open Source Software Development Projects: Towards a Model for Network-Centric Edge Organizations," in *Proceedings of the 13th International Command and control Research and Technology Symposia (ICCRTS 2008)*, Seattle, WA, USA, 2008.

[52] A. G. Koru and J. Tian, "Comparing high-change modules and modules with the highest measurement values in two large-scale open-source products," *Software Engineering, IEEE Transactions on,* vol. 31, pp. 625-642, 2005.

[53] D. Spinellis, "A Tale of Four Kernels," presented at the 30th International Conference on Software Engineering, 2008. ICSE '08, Leipzig, Germany, 2008.

[54] A. MacCormack, J. Rusnak, and C. Y. Baldwin, "Exploring the structure of complex software designs: An empirical study of open source and proprietary code," *Management Science,* vol. 52, p. 1015, 2006.

[55] G. Robles Martínez, "Empirical Software Engineering Research on Libre Software: Data Sources, Methodologies and Results," PhD, Universidad Rey Juan Carlos, Madrid, Spain, 2005.

[56] A. Capiluppi, P. Lago, and M. Morisio, "Evidences in the evolution of OS projects through Changelog Analyses," in *3rd Workshop on Open Source Software Engineering at the International Conference on Software Engineering (ICSE)*, Portland, OR, USA, 2003, pp. 19-24.

[57] P. Giuri, F. Rullani, and S. Torrisi, "Explaining leadership in virtual teams: The case of open source software," *Information Economics and Policy,* vol. 20, pp. 305-315, 2008.

Tampereen teknillinen yliopisto
PL 527
33101 Tampere

Tampere University of Technology
P.O.B. 527
FI-33101 Tampere, Finland