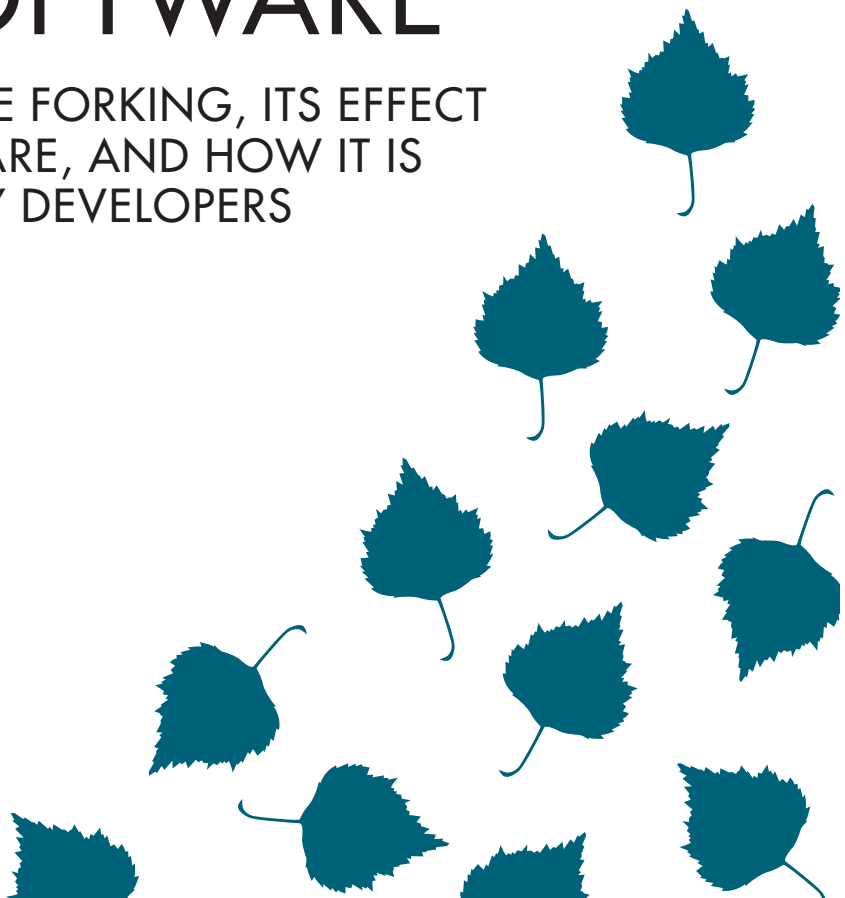
EKONOMI OCH SAMHÄLLE
ECONOMICS AND SOCIETY

HANKEN

UNDERSTANDING CODE FORKING IN OPEN SOURCE SOFTWARE

AN EXAMINATION OF CODE FORKING, ITS EFFECT ON OPEN SOURCE SOFTWARE, AND HOW IT IS VIEWED AND PRACTICED BY DEVELOPERS

LINUS NYMAN



Ekonomi och samhälle
Economics and Society

Skrifter utgivna vid Svenska handelshögskolan
Publications of the Hanken School of Economics

Nr 287

Linus Nyman

Understanding Code Forking in Open Source Software

An examination of code forking, its effect on open source software, and how it is viewed and practiced by developers

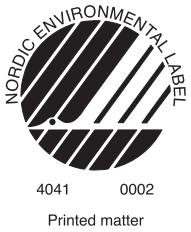
Helsinki 2015

Understanding Code Forking in Open Source Software: An examination of code forking, its effect on open source software, and how it is viewed and practiced by developers

Key words: Code forking, fork, open source software, free software

© Hanken School of Economics & Linus Nyman, 2015

Linus Nyman
Hanken School of Economics
Information Systems Science, Department of Management and Organisation
P.O.Box 479, 00101 Helsinki, Finland



Hanken School of Economics

ISBN 978-952-232-274-6 (printed)
ISBN 978-952-232-275-3 (PDF)
ISSN-L 0424-7256
ISSN 0424-7256 (printed)
ISSN 2242-699X (PDF)

Edita Prima Ltd, Helsinki 2015

ACKNOWLEDGEMENTS

There are many people who either helped make this book possible, or at the very least much more enjoyable to write. Firstly I would like to thank my pre-examiners Imed Hammouda and Björn Lundell for their insightful suggestions and remarks. Furthermore, I am grateful to Imed for also serving as my opponent.

I would also like to express my sincere gratitude to Liikesivistysrahasto, the Hanken Foundation, the Wallenberg Foundation, and the Finnish Unix User Group. They gave me the financial freedom to spend years scratching my stubble while pondering a truly fascinating topic.

The time spent at work was made enjoyable by a great bunch of people. First off, many thanks to Catarina and Sofia for their excellent company during our shared first year of PhD coursework and confusion. We were fortunate to catch the tail end of the "Third Floor", a marvelous group of people one could not but enjoy spending time with – yes, I'm looking at you Jennie, Sanne, Hertta, Markus, and Matte. (*The Third Floor is dead, long live the Third Floor!*) I was a wanna-be geek hoping to go native during the course of writing about computer-related stuff. It was Mikael who helped me take step after faltering step toward geekdom. Our countless hours in the Nerd Cave discussing Humble Bundles and daily deals on Steam, assembling a new computer (or, as my wife usually put it, "*another* new computer?"), or debating the correct coleslaw moisture ratio made even the mundane enjoyable. Many thanks also go to the rest of the department, past and present, the so-called FLOsters: Pekka, Dmitri, Eric, Mikko, Olga, Jonnu, Fille, Wille, Inkku, Patrik, Mats, Martin von W. and Martin F., Jan(ne), Sari, Tom, Henrika, Kari, Paulina, and all of the others. Last but not least, Chris deserves thanks, not just for being who he is, but also for making me appear to be considerably more eloquent than I ever will be.

My aversion to getting a real job deserves only part of the credit for inspiring me to write a dissertation. The majority of the credit goes to Saku Mantere, who supervised me through the writing of my Master's thesis, and encouraged me to take on something more challenging. Further thanks are owed to Saku and Eero Vaara for serving as my supervisors for the first year of my dissertation. A big debt of gratitude is owed to both Bo-Christer Björk (aka Bobi) and Turid Hedlund, who took me in at the Information Systems Science department once it became clear that my interests were driving the book in that direction. Bobi and Juho Lindman served as my supervisors, granting me my request to freely stumble around in a little-researched field until I smacked my head into something interesting. I am grateful to both of them for offering the perfect mix of freedom, guidance, and feedback. Further thanks go to Juho for being an insightful and knowledgeable co-author.

I owe a great debt of gratitude to the many researchers and industry professionals who helped with ideas and feedback along the way. In addition to co-authoring one of the papers of this dissertation, Tommi Mikkonen offered help and guidance whenever needed, even back when I had only a vague notion of what I was doing. Tommi and the other open source researchers at the Tampere University of Technology, most notably Imed and Terhi, always made me feel like part of the gang. Walt Scacchi deserves a special mention for sharing his insights in general, but also in particular for suggesting that I look into the history of the term fork. Susanna Taimitarha was my go-to stats wizard, and I could imagine no one more pleasant or knowledgeable.

Few things are as valuable to the confused PhD student as an expert willing to answer emails. I thank Glyn Moody, Jesper Nevalainen, and Heather Meeker for being such experts. Few things are as valuable to the procrastination-prone PhD student as a journal editor who spurs article submissions beyond even what was promised. I thank Chris McPhee for being such an editor.

This book would have amounted to little more than dubious speculation, were it not for the many programmers and open source experts who shared their views and insights with me over the years. In addition to those already mentioned, I am very grateful to Lars Wirzenius, Artem Goutsoul, Niklas von Hertzen, Stefano Campadello, Patrik Welling, Jukka Z., Kalle H., Pekka B., Rasmus J., and the many other programmers who shared so freely of their time. Thanks also go to Olivier ("Olli") Beutels for being both knowledgeable and approachable, and to Camilla Z. and Ralf W. for sharing their experiences, opinions, and insights.

Monty Widenius offered so generously of his time and insights, it requires a special mention. From the multiple interviews included in this book, to the later co-authored papers on the inner workings of open source, this dissertation benefitted immensely from both his knowledge and his willingness to share his stories and views. I would also like to thank Monty's assistant Andrea for all her help in making things happen.

Tracking the history of the term fork, both in finding its way into the English language as well as into computing, was only possible due to the help of many people, all deserving of thanks. For sharing freely of their time and recollections of years past, as well as for their help in attempting to track down the original source code from Project Genie, my humble thanks go to: L Peter Deutsch, Butler Lampson, Ken Thompson, Melvin Conway, Jack Dennis, Jerry Saltzer, Tom van Vleck, Brian Randell, Wayne Lichtenberger, and Dag Spicer. Mark Riley and Elizabeth Tucker are to thank for sharing their knowledge of the early history of the term fork, and Anton Granvik has my eternal gratitude both for helping dig up obscure Latin references as well as for his oversight of my lackluster knowledge of Portuguese.

While on the topic of uncovering and understanding yesteryear, Steve Russell and Peter Samson both deserve and have my thanks for shedding light on the early days of *Spacewar!,* and Reino Kurki-Suonio likewise for sharing his insights and stories about the history of computing in Finland.

If it is indeed true that one can judge a man by the company he keeps then I owe an enormous debt of gratitude to my friends outside of academia for their continued bad judgment. Petteri, Anssi, Arno, Stefu, Markku & Hajam (the PhD slippers have been in regular use!) and the rest – thanks for all the great times! Thanks also go to my wife's many wonderful friends (as well as wife's-friend's-brother: I'm looking at you, Andu). I have had the delightful marital duty of every so often spending time with Kata & Osku, Minna & Tomas, and Ann-Sofie & Bertrand (some of whom I already knew, most of whom I did not), and now consider them my friends as well. For their pleasant company, pre-opening Coffee Club, and inventing of the Humanoid, I am grateful to Mika, Ansku (aka Anastasia), and the rest of the staff, past and present, at Shaker.

Heartfelt thanks go to my mother and father, Kate and Ebbe, for their support, encouragement, and especially their babysitting services! Thanks also to my godfather Konschin, who accepted the role of Bullshit Bingo Detector, tasked with keeping fashionable nonsense out of my dissertation. And to my mother-in-law Kisa and father-

in-law Georg, for being not only great grandparents, but also great babysitting grandparents!

Finally, immeasurable thanks go to my wife Yrsa, and my two wonderful sons Sebastian and Benjamin, both born during the writing of this book. Completing my PhD before they learned to say, “Daddy, are you *still* working on your dissertation?” served as a constant source of panicked encouragement. One of the defining characteristics of the Finnish male is his inability to express emotions like love and gratitude when sober. Another defining characteristic is his inability to *stop* expressing them when intoxicated. Were I to have written these acknowledgements while considerably less sober, I would have said something to the effect of: “Without you, there would be nothing. Somewhere beneath the thick layers of cynical skepticism that I define myself by, there is an unquenchable warm and cuddly glow that exists because you all are part of my life.”

Helsinki, 9.1.2015

Linus Nyman

CONTENTS

1 INTRODUCTION....1
1.1. Code forking: what has been written so far....2
1.2. Code forking: some examples from open source history....5
1.3. The research approach and objective....6
1.4. Positioning the articles with the goals of the dissertation....8
1.5. Comparing the article goals with the previous body of knowledge....9
1.6. Structure of the dissertation....9
2 OPEN SOURCE SOFTWARE....10
2.1. What is "open source" software?....10
2.1.1. The nuts and bolts: source code and compiling....10
2.1.2. The briefest of history lessons, part 1: software....11
2.1.3. The briefest of history lessons, part 2: hackers and the hacker ethic....13
2.1.4. Back to square one: What is open source software?....13
2.2. Of purists and pragmatists: free software vs. open source software, and their use in this book....14
2.3. Software copyright and open source licenses....15
2.3.1. Software ownership and licensing....15
2.3.2. Open source software licensing....17
2.3.3. Types of open source licenses....17
2.3.4. License compatibility....18
2.4. The business of open source....20
2.4.1. Business models....21
2.4.2. Architectures of participation....24
2.5. Summary....25
3 CODE FORKING....26
3.1. The history of the term "fork"....26
3.1.1. The fork system call is proposed....26
3.1.2. The fork system call is born....28
3.1.3. Forking takes the fork system call global....30
3.1.4. Forking forks to open source....31
3.2. The definition of fork....32
3.3. How does the right to fork affect open source software?....33

3.3.1. Software level 34
3.3.2. Community level 35
3.3.3. Business ecosystem level 36
3.4. The prevalence and complexity of forking: some (simplified) examples 37
3.5. Code forking defined, part 2: competitive and non-competitive forks 39
3.6. Summary 40

4 RESEARCH OBJECTIVES, METHODS, AND DATA 41
4.1. From ignorance to confusion to insight 41
4.2. The research questions and articles that address them 42
4.3. Research methods, data collection, and analysis 43
4.3.1. Paper 1 43
4.3.2. Paper 2 43
4.3.3. Paper 3 45
4.3.4. Papers 4 and 5 46

5 SUMMARIES OF THE PAPERS 48
5.1. Paper 1: Code Forking, Governance, and Sustainability in Open Source Software 48
5.2. Paper 2: To Fork or Not to Fork: Fork Motivations in SourceForge Projects 49
5.3. Paper 3: Freedom and Forking in Open Source Software: the MariaDB Story 51
5.4. Paper 4: Hackers on Forking 52
5.5. Paper 5: When is a Fork a Fork? On the Definition(s) of Code Forking 54

6 DISCUSSION 56
6.1. Main findings and contribution 56
6.2. The research questions revisited 57
6.2.1. Research question 1: What significance does code forking have to open source software? 57
6.2.2. Research question 2: How do programmers view the right to fork and the practice of code forking? 58
6.2.3. Research question 3: Why do programmers fork code; what are the motivations behind code forking? 59
6.2.3.1. Fork motivations: five categories 59
6.2.3.2. Fork motivations: four circles and an arrow 61
6.2.4. Research question 4: Does “fork” mean different things to different programmers? 63
6.3. Comparing the findings to previous knowledge 64

6.3.1. Academic study 64
6.3.2. Industry insights 66
6.4. Managerial implications 67
6.5. On validity and reliability, limitations, and suggestions for future research 68
6.5.1. Validity and reliability 68
6.5.2. Limitations 69
6.5.3. Suggestions for future research 70
6.6. Summary 72

REFERENCES 73

APPENDICES

Appendix 1 The Open Source Definition 86

Appendix 2 The interview guide for papers 4 and 5 88

Appendix 3 Papers 89

TABLES

Table 1 An overview of previous research and writings on code forking 2

Table 2 Business model selection criteria (source: Widenius and Nyman, 2014) 22

Table 3 Some examples of the coding process 44

Table 4 Key facts for article 1 49

Table 5 Key facts for article 2 51

Table 6 Key facts for article 3 52

Table 7 Key facts for article 4 54

Table 8 Key facts for article 5 55

FIGURES

Figure 1 An original, unlicensed program and differently licensed versions released of it 16

Figure 2 David A. Wheeler's (2007b) license compatibility diagram 19

Figure 3 Example of a corporate ("sponsored") open source project development (from paper 3) 24

Figure 4 The fork system call as originally suggested by Conway in 1963 27

Figure 5 "Fork logic" in Tymshare (derived from Project Genie) 29

Figure 6 The fork system call described in the Unix Programmer's Manual from 1971 29

Figure 7 35 years of forking in the lives of Unix and Unix-like systems 38

Figure 8 The continuum of competitive versus non-competitive forks 39

Figure 9 An overview of the research questions and the articles that address them ... 42

Figure 10 Competitive versus non-competitive fork likelihoods 61

1 INTRODUCTION

Open source software[1] can be found in more places than most people might imagine. Beyond the obvious implementations such as computers and smartphones, open source software can be found in things like TVs, cars, weather stations, and washing machines. Open source software has even left our planet – it can be found in satellites orbiting the earth. It is likely that anyone reading this has within the last week used a program or device that relied on open source to function.

A broader interest in the open source software phenomenon appears to have emerged around the turn of the millennium, with books describing open source in general and Linux in particular (e.g. Raymond, 1999; Moody, 2001), biographies of key figures like Linus Torvalds (e.g. Torvalds and Diamond, 2001) and Richard Stallman (e.g. Williams, 2002), and the business and economics of open source in general and Linux in particular (e.g. Hecker, 1999; Fink, 2003). Academic research into open source began emerging around this same time. A seminal paper by Lerner and Tirole (2002) focused largely on understanding and describing open source and its emergence, as well as theorizing on its motivations. Three of the earliest themes in the study of open source software were: the people and motivations behind it (e.g. Robles, Scheider, Tretkowski and Weber, 2002; Lakhani, Wolf, Bates, and DiBona, 2002; Lakhani and Wolf, 2005); the legal framework guiding, enabling, and enforcing it (e.g. McGowan, 2000 & 2005; Tirole and Lerner, 2005); and aspects related to innovation in open source software (e.g. von Krogh, Spaeth, and Lakhani, 2003), including "the private-collective innovation model" (first presented in von Hippel and von Krogh, 2003) and "open innovation" (e.g. West, 2006; West and Gallagher, 2006).

Several other themes emerged within the study of open source. Von Krogh and von Hippel (2006) reviewed existing research on open source, noting that it can be categorized into three areas of study: the motivations behind open source contributing; governance, organization, and the process of innovation in open source software projects; and competitive dynamics enforced by open source software. Aksulu and Wade (2010) conducted a review of the research in open source software during its first ten plus years, dividing the 1 355 peer-reviewed articles they found into seven categories: conceptual, performance metrics, legal and regulatory, open source software (OSS) production, OSS applications, OSS diffusion, and "beyond software".

The topic of code forking is notable mainly by its absence. While an in-depth discussion of its definition will come later, a very general interpretation of what forking means is copying an existing program and distributing a modified version of it. This is something that one cannot do with proprietary software for numerous reasons, including copyright protection and restricted access to the source code. Conversely, with open source software, one cannot *forbid* anyone from forking the code. Despite the significance of the right to fork, most of the material available in which there is at least some discussion of code forking is of a practitioner-oriented nature. Indeed, as recently as 2012, Robles and González-Barahona note:

> To the knowledge of the authors, no complete and homogeneous research on forking has been done by the software engineering research community.

[1] Open source software is described and explained in Chapter 2, but can briefly be defined as software that is licensed under an Open Source Initiative (OSI) approved license.

A deeper analysis of forking is significant given both the lack of current research into code forking as well as the central role that code forking, and the possibility of forking, plays in open source software. 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 the project can continue to exist and even evolve as long as there is an interested developer community (e.g. Nyman et al., 2011 & 2012). Indeed, the right to fork is not only a large part of what makes much of open source software possible, it is also what has defined the business of open source and determined how it is conducted. The right to fork means that anyone can start their own version of a program, even ones that are the result of millions of euros worth of investment.

1.1. Code forking: what has been written so far

As much of the extant literature on code forking is to be found outside of academia, it would be remiss to exclude it from this study. Existing literature can be divided into 3 general categories, as visualized in Table 1: i) peer-reviewed research papers (papers included in this dissertation are excluded from the table); ii) books and practitioner guides, commonly by people whose work is either in, or related to open source software development; and, iii) other sources, including news websites, blogs, discussion forums, and other sources of a similar nature, and commonly written by developers or other members of an open source software development project.

Table 1 An overview of previous research and writings on code forking

Type of publication	Examples	Focus
Peer-reviewed research papers with a focus on forking	Nyman et al. (2011 & 2012) Robles and González-Barahona (2012) Viseur (2012) Gamalielsson and Lundell (2012 & 2014) Azarbakht and Jensen (2014)	Theoretical papers Fork motivations and outcomes Impacts and motivations The long-term sustainability and evolution of a fork Community contributions and social dynamics
Books, practitioner guides, papers published on personal webpages (not peer-reviewed) that discuss forking	Raymond (e.g. 1998, 1999) Moen (1999) St. Lauren (2004) Weber (2004) Fogel (2006) Wheeler (2007a) Meeker (2008)	Explaining the phenomenon of forking, the licensing that makes it possible, and how it affects open source software
Stories about forking (news sites, industry reports, blogs, hacker folklore, etc.)	Blogs (primarily developers) Community websites (e.g. Slashdot.org, Stackoverflow.com) Industry news-websites (e.g. arstechnica.com, itwire.com, theregister.co.uk, openlife.cc)	Discussions about forks, speculation and discussions of the practical implications of forks that have already occurred or that may occur

While there are some academic papers that mention forking, few have it as their focus. Of those that do, this author has co-authored two theoretical papers on the topic of code forking and sustainability, with a focus on the benefits to sustainability that derive from the right to fork (see Nyman et al., 2011 & 2012). Gamalielsson and Lundell (2012 & 2014) studied the LibreOffice fork from OpenOffice, finding that it is possible for a fork to prosper and achieve sustainability over time, and for a software community to outlive open source software projects. Robles and González-Barahona (2012) conducted an in-depth analysis of 220 forks, revealing that they occur in every software domain, have become more frequent in recent years, rarely merge with the original project, and can be of a friendly as well as competitive nature. Viseur (2012) conducted a study of 26 forks, with a focus on their motivations and impact on the original project, finding that the majority of forks studied (42%) were motivated by a desire for technical specialization (i.e. doing something differently than the original), and that a fork is rarely (19%) followed by the extinction of the original project. In the majority (54%) of the cases studied, the fork and original both survived, achieving cohabitation. Azarbakht and Jensen (2014) note that in addition to forking as a "friendly divide" in order to try out new features, forking may also be a "violent split" resulting from conflict. They propose and test a means of collaboration analysis, using temporal visualization, to detect unhealthy collaboration dynamics and potentially prevent an unwanted fork.

A review of existing literature on code forking shows that while there is much similarity among their definitions, they are not identical. Raymond (1999) and Robles and González-Barahona (2012) offer perhaps the most detailed typography of forking and fork-like phenomenon. Raymond proposes a distinction between *code fragmentation*, *pseudo-forking*, and *code forking*, with the actions of the developer community as well as the compatibility of the new code being central issues in differentiating code forking from code fragmentation. Different distributions of a program are considered pseudo-forks. At first glance they appear to be forks, but in fact are not, since they can benefit enough from each other's' development efforts not to be a waste. Robles and González-Barahona (2012: 3) define forking as occurring when "a part of a development community (or a third party not related to the project) starts a completely independent line of development based on the source code basis of the project." Furthermore, they note that for something to be able to be called a fork it should have: i) a new project name; ii) a branch of the software; iii) a parallel infrastructure (web site, versioning system, mailing lists, etc.); and, iv) a new developer community (disjoint with the original). Moen (1999) quotes an unnamed source in describing forking simply as "the right to start your own mutant version of any open source project". In a similar vein, St. Lauren (2004: 171) notes that forking "occurs whenever a software project splits", and, while they may be compatible at the start, the unique histories of each one's development will eventually and inevitably push them apart. Similarly, Weber (2004: 12) finds that such systems are likely to evolve into incompatible versions. The Jargon file, an online hacker dictionary, defines a fork as "when two (or more) versions of a software package's source code are being developed in parallel which once shared a common code base, and these multiple versions of the source code have irreconcilable differences between them" (Raymond, 2000).

The issue of compatibility is a central issue not only to some definitions of forking, but also to motivating factors relating to *not* wanting to fork. While open source licenses allow for both forking and pseudo-forking, Raymond (1999) notes that only pseudo-forking is common, most likely due to the strong social pressure against forking projects. The sentiment is mirrored by Meeker (2008):

> 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.

Weber (2004) further discusses the issue of reputation, noting that a fork that ran into development problems would take a toll on the reputation of the developer that forked the code. In addition to reputation, Weber underlines the challenge of attracting a sufficient pool of developers to aid in the further development of a fork, particularly if the original already has a significant community. To gain the support of developers of the original, the developers who fork the code must convince the community both that they are better suited to lead the development, and that they will be able to attract a community that will be larger than the community working on the original. Programs under a copyleft license (discussed in greater depth in section 2.3.1) have further disincentives to forking: if a program is forked and then improved, the original program can most likely simply incorporate those improvements, thus hampering the benefit of forking both from a development viewpoint (e.g. Moen, 1999), as well as a business-oriented viewpoint (e.g. Wheeler, 2007a).

Even given the many reasons why forking is noted to be actively avoided, the importance of the right to fork is still underlined by many. Fogel (2006: 88) considers the potential to fork a program to be "the indispensable ingredient that binds developers together", noting that forks are often seen as being bad for everyone, so developers are generally willing to compromise in order to avoid a serious threat of forking. Thus, the potential to fork is also a strong element in the governance of open source programs (e.g. Weber, 2004; Moody, 2009a), as no one has a "magical hold" (Fogel, 2006) over any project, i.e., anyone can fork any project at any time. Moen (1999) expressed similar sentiments, calling the right to fork "a vital safety valve" in case the existing developers stop working on a project or decide to stand in the way of progress. The same is true of corporate actions and acquisitions. Should a company try to "hijack" (Lerner and Tirole, 2002) the source code by making it proprietary, or otherwise not act in the best interests of the program and its community, forking provides the developer community a way to spin off their own version of the project (e.g. Moody, 2009a). The right to fork is a vital right in open source (e.g. Weber, 2004) and a software license that does not grant that right is not genuinely an open source license (e.g. Moen, 1999; see also the Open Source Definition[2] and the Free Software Definition[3]).

Both Weber (2004) and Fogel (2006) 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. Importantly, it is not the existence of a fork, *per se*, that hurts a project, but rather the loss of developers and users (e.g. Raymond, 1999; Fogel, 2006; Moody, 2011). Wheeler (2007a) notes that there are four possible outcomes in the event of a fork:

1. The death of the fork – the most common outcome

[2] Maintained by the OSI, available at: http://opensource.org/osd, accessed 13 May, 2014.

[3] Maintained by the Free Software Foundation, available at: https://www.gnu.org/philosophy/free-sw.html, accessed 13 May, 2014.

2. The re-merging of the fork – both projects re-join each other (though one or the other may be the dominant source of the combined effort)
3. The death of the original development
4. Successful branching – both succeed, typically catering to different communities

To these four, Robles and González-Barahona (2012) add a fifth outcome: that both projects are discontinued. The notion that both forks succeeding is typically achieved through serving different communities and needs was explained, e.g., by Moen (1999) and Weber (2004) who noted that ideas for extensions to a project that are outside of the focus of the project, but that manage to generate sufficient community interest, can result in a fork that will concentrate on an important niche that the original cannot fill. In such a case, the development teams may well work together, sharing code and information to prevent redundant effort. St. Lauren (2004) shares this sentiment, positing further that the likelihood of a fork developing into a non-competitive version of the original is greater the less mature the original program is when the fork occurs.

Having mapped out previous views on forking, it is significant to conclude by noting that forks, in the traditional sense of the word (section 3.2 and paper 5 broaden the meaning of the term), are quite uncommon. In fact, Raymond (1998) notes that forks are so rare as to be "remembered in the hacker folklore." Thus, in addition to extant writings on forks, it is useful to cover as least some of the forks from said folklore. A discussion of some example forks may furthermore aid the less open source savvy reader to position the theme and area of focus of this dissertation.

1.2. Code forking: some examples from open source history

The story of the GCC-EGCS fork is among the most well known and most notable forks given that the fork became the *de facto* standard, eventually replacing the original it was forked from (outcome two of Wheeler's list). The story begins in 1987 with the release of a program called the GNU Compiler Collection (GCC).[4] Developers at that time considered GCC to be a significant advancement to the emerging field of programming. However, many of those developers did not share the views of the project leaders regarding the future and direction of the program. This led to the birth of a number of GCC forks. In 1997, an experimental fork called EGCS (an acronym for Experimental/Enhanced GNU Compiler System) was started with the goal of unifying several of these forks (e.g. Henkel-Wallace, 1997). This new fork received considerable community support and interest and rapidly evolved into a program poised to overtake the original development. Indeed, a mere two years after EGCS was started, the original developers of GCC gave up their project and even allowed EGCS to use the name GCC. Thus, when one speaks of the GCC, one is actually speaking of a fork of the original GCC that became so successful that it was accepted by the original as the new standard.

Differences of opinion between a project's leaders and community have spawned many, if not most, of the forks in hacker folklore. However, a fork's becoming the new *de facto* standard is no assured outcome. Forks may thrive by serving a different purpose or they may simply wither away into abandonment. Two further stories illustrate these points: the story of the BSD family of forks, and the story of a fork called Drizzle.

[4] Compiling will be covered in section 2.1.1.

Drizzle is a fork of MySQL (a so-called relational database management system). It differs from MySQL in that it is a slimmed down, lighter version of it, with a focus on speedier implementation of code commits considered critical by the community (see, e.g., blog posts by Aker, 2008; Widenius, 2008). Drizzle has enjoyed a significant community, having had just shy of 100 contributors and over 13,000 commits by 2010 (Aker, 2010). However, despite these impressive numbers, by late 2014 Drizzle appeared to no longer be in active development.[5] The final example, from the BSD family of forks, illustrates a further interesting and significant aspect of forking: there are many programs that are no longer actively developed, but that live on in the form of numerous different forks of the original. Furthermore, as the BSD story also illustrates, programs may see more than just one level of fork. The picture can become rather complicated, with forks spawning forks of their own that, in turn, may be forked, and forked again. But let us digress no further and steer back to the promised story.

Once upon a time (1977), an operating system called Berkley Software Distribution (BSD) was released. Though active development of BSD ended in the 1990s, there are currently dozens of forks of BSD in active development. One such fork is FreeBSD. FreeBSD was forked from 386BSD, which was a fork of Network Release 2 (Net/2), which was a fork from (version 4.3 of) BSD. Each of these forks stems from the original BSD, and each had their own purposes and goals. The Net/2 fork of BSD prioritized open source licensing over proprietary licensing, rewriting or removing all proprietary code from BSD. The 386BSD fork of Net/2 was rewritten to make the operating system work on different computer hardware. And, finally, the FreeBSD fork of 386BSD was initially to be an updated version of 386BSD, but due to disagreements between developers and uncertainty about 386BSD's future, it was released as its own fork instead. There are many more forks in the BSD family than can be covered in so brief an introduction. While a somewhat more in-depth view of BSD's history will be provided in section 3.4, a final point worth noting is that FreeBSD itself has been forked numerous times. FreeBSD can be found in dozens of programs, both open source and proprietary, either in part (e.g. Mac OS X has significant amounts of FreeBSD code) or in its entirety (e.g. Netflix uses FreeBSD[6]).

1.3. The research approach and objective

In the fall of 2009 when I began working on this book, I could not find any studies with a specific focus on code forking. Due to this scarcity of information, I decided to adopt an exploratory approach. This approach is useful when the research questions are vague or when there is little theory available to guide the development of a hypothesis (Hair, Money, Samouel, and Page, 2007: 154). In some ways this approach mirrored the early studies on open source itself, examining questions like how the basic economics of open source worked (Lerner and Tirole, 2002), what the underlying programmer motivations were (Lakhani and Wolf, 2005), and if the open source phenomenon could be considered sustainable (Bonaccorsi and Rossi, 2003). Furthermore, I decided to adopt an article-based approach, as I believed such an approach would offer the opportunity to explore the topic broadly while still focusing

[5] In the summer of 2013, Drizzle still participated in Google's Summer of Code. However, the Drizzle DB Project's Twitter feed (https://twitter.com/DrizzleDB) has had no new posts since 13 August, 2013, and the latest stable release dates back to 23 September, 2013 (see, e.g., https://launchpad.net/drizzle/7.2). Furthermore, when discussing Drizzle with two separate MariaDB employees, one in early 2014 and one in late 2014, both believed common opinion to be that Drizzle no longer was in active development.

[6] See, e.g., https://openconnect.itp.netflix.com/software/index.html, accessed 5 December, 2014.

on specific areas of interest within the topic. The methodology adopted was chosen individually for each article, based on the goals of each particular article. The varied nature of the research questions led to a variety of methods used. These are described in greater depth in section 4.3.

Among practitioner literature there has been a tendency to offer somewhat limited space to the concept of forking and to place somewhat more focus on the potential negative aspects of a fork occurring. I could not find any papers that had focused on a theoretical analysis of forking, attempting to chart its positive relevance to open source software development in general, and open source software's sustainability in particular. Highlighting the contemporary relevance of this information, Gamalielsson and Lundell (2012 & 2104) explored a very similar topic during the writing of this dissertation. Thus, one question that came to interest me was what positive aspects one could note about the right to fork.

During the writing of this book, other research on the topic of forking was published. This research has offered us a better understanding of forking. Beyond Robles and González-Barahona (2012) and Viseur (2012), there is little focus among extant studies into the motivations behind the inception of a fork, i.e., what factors help lead to a fork's Big Bang moment. This seems a critical moment, most obviously because any question of how a fork performs or affects its environment is contingent upon its first coming into existence. Developer motivations for starting a fork seemed a significant and fascinating area of study.

In order to gain a comprehensive understanding of the underlying motivations behind forking – an understanding that went beyond just one specific instance of forking – it seemed that one needed to understand how developers view forking as a concept. This was another area that had not yet been the explicit focus of academic study. And yet its significance seems undisputable. Understanding what developers consider valid motivations for a fork can offer significant insight regarding both predicting forks and helping to avoid them. Furthermore, the more developers who share the views that resulted in a fork, the greater the likelihood that the fork can attract a significant developer community. This, in turn, is significant to the fork's ability to remain relevant through continued development and updates.

Based on the points of interest above, the objective of this dissertation can be summarized as follows: the overall aim of this dissertation is *to provide a greater understanding of the practice of code forking in open source software, with particular emphasis placed on understanding developer views and motivations.* The specific research questions stemming from this aim are:

> ***Research question 1:*** *What significance does code forking have to open source software?*
>
> ***Research question 2:*** *How do programmers view the right to fork and the practice of code forking?*
>
> ***Research question 3:*** *Why do programmers fork code; what are the motivations behind code forking?*
>
> ***Research question 4:*** *Does "fork" mean different things to different programmers?*

Here it should be noted that in this book I use the terms programmer, hacker, and developer interchangeably.

1.4. Positioning the articles with the goals of the dissertation

Paper 1, **Code Forking, Governance, and Sustainability in Open Source Software**, analyses the concept of forking from a theoretical perspective, discussing its implications on a general level. This paper primarily addresses research question 1.

Paper 2, **To Fork or Not to Fork: Fork Motivations in SourceForge Projects**, studies forks hosted on the SourceForge repository, with a focus on the motivations developers state as the reasons behind the forks. The article offers insight primarily into forks of a non-competitive nature – instances in which a program was copied and used as the starting point for a new project, but that did not directly compete for the same users, or at least one that attempted to serve a different need (the concepts of competitive and non-competitive forks are discussed further in section 3.5). This paper primarily addresses research question 3, but also speaks to research question 1.

Paper 3, **Freedom and Forking in Open Source Software: the MariaDB Story**, analyses a competitive fork. In this instance a competing version of the original program was started and the original developers split into two groups, those that remained with the original version and those that joined the fork. This is forking in what can be considered its original meaning, and a fairly rare occurrence in open source software. This paper primarily addresses research question 3, but also speaks to research question 1.

Paper 4, **Hackers on Forking**, is based on interviews with open source programmers about their views on forking. It attempts to chart programmers' views on the positives and negatives of the right to fork, what kinds of forks they consider acceptable, and document their views on developer responsibilities in regards to the act of forking. This paper primarily addresses research question 2, but is also of considerable significance to research questions 1 and 3.

Paper 5, **When is a Fork a Fork? On the Definition(s) of Code Forking**, describes the results of a study in which programmers were interviewed about their definition of the term fork and what the specific requirements are for something to be defined as a fork. This paper addresses research question 4. This research question was a late addition to the research objectives. It was born from the realization that there appeared to be significant differences in interpretation of the term, an insight acquired during the researching and writing of the other articles of this dissertation. The significance of this understanding coming during the researching and writing of this book, and how that was addressed, is discussed in section 6.5.1.

The reasoning behind the order of the papers is the following. Paper 1 gives a succinct description of the effect that the right to fork has on several aspects of open source software development. Furthermore, for those unfamiliar with the topic, it also serves as an introductory paper to the concept of forking. Papers 2 and 3 then go on to explore the motivations behind forks, both a competitive fork (MariaDB, arguably one of the most significant forks in recent open source history) as well as forks of a primarily non-competitive nature. Finally, we hear from the hackers themselves, regarding both how they view forking (paper 4) and how they define a fork (paper 5).

1.5. Comparing the article goals with the previous body of knowledge

Paper 1. There are some noteworthy differences between the focus of the papers in this dissertation and the focus of previous studies. Though the previous body of work cannot be said to have overlooked the potential positive aspects of forking, such aspects had nonetheless received fairly little attention and were the focus of paper 1 in particular.

Paper 2. To the best of my knowledge, paper 2 was the first paper to address the motivations behind forks at the time of its publication. Two further papers on the topic were published a year later (see Robles and González-Barahona [2012] and Viseur [2012]), and it is still a subject area ripe for further investigation.

Paper 3. Gamalielsson and Lundell (2012 & 2014) analyse a fork (the LibreOffice fork of OpenOffice), but with a slightly different focus from paper 3, where the spotlight is on the motivations behind the decision to start a fork.

Paper 4. This paper goes beyond a focus on motivations to examine more broadly how programmers view forking. While views about forking have been stated by some academics as well as practitioners, I believe this paper is the first time a deeper understanding of programmers' views (both testing and broadening previous insights) has been sought through interviews on the grassroots level specifically targeting the topic.

Paper 5. The issue of the changing nature of how programmers use the word fork is a phenomenon that I have not seen brought up in previous works, and is significant in particular in designing research on the topic of forking.

1.6. Structure of the dissertation

American folk musician Woody Guthrie is noted to have said, "Any fool can make something complicated. It takes a genius to make it simple." While I cannot make the complexities of open source in general and code forking in particular simple, it is nonetheless one of the goals of this dissertation to make them approachable enough so as to enable anyone interested to understand both the general topic of this book as well as the findings presented in it. To achieve this end, it is necessary to include some background information about open source software. Chapter 2 is dedicated to this end, and is intended primarily for those unfamiliar with the topic. Code forking is covered in chapter 3. Chapter 4 discusses issues related to research methods, data collection, and its analysis. Chapter 5, in turn, summarizes the papers of this dissertation. Chapter 6 concludes with a discussion of findings, contributions, implications, validity and reliability, as well as some suggested avenues for future research.

2 OPEN SOURCE SOFTWARE

The purpose of this chapter is to offer the uninitiated reader a basic understanding of open source software. The aim is not to cover the topic completely, but rather to focus on elements helpful to understanding code forking, covered in the next chapter. Those already familiar with open source software may prefer to skip ahead to chapter 3.

This chapter is structured as follows: first, we cover the question of what open source is. This is done through both a technical and historical approach. Second, we cover open source licenses and license compatibility. Finally, we cover the business of open source, including business models and architectures of participation.

2.1. What is "open source" software?

While it would be easier to answer this question by simply quoting the official Open Source Definition (OSD), I feel it more helpful to take the scenic route (the OSD can be found in Appendix 1). To understand *what* open source software is we should first understand a bit about the nuts and bolts of software. To understand *why* open source software is, and why open source software is more than just "open source" software, we should first know a little bit about the history of software, and a thing or two about hackers. Let us begin with two basic concepts of software: source code and compiling.

2.1.1. The nuts and bolts: source code and compiling

The "source" in open source software refers to the software's source code. The source code is what tells a computer what to do. If you are like me, you may every now and then be given a list by your spouse that reads something along the lines of "milk, bread, tomatoes, food for Sat + Sun." Allowing for a generous stretch of the metaphor, we can compare that list to source code. The list, in short, tells me what to do: firstly, that I need to go to the store, and secondly that I need to buy milk, bread, tomatoes, and "food" for the weekend. While computers require much more specific instructions than you or I would need in order to accomplish a task, the concept is the same. Source code is the list that tells a computer what to do: "If the user pushes this button, do this; if the user pushes that button, do that."

When I was growing up, my first computer was a Commodore 64. The programming language it used was called BASIC.[7] A very simple computer program written in BASIC could look like this:

```
10 Print "Hello world. This is my dissertation. (So many words,
so few readers...)"
```

[7] Or "Commodore BASIC", to be exact. Commodore BASIC is a variation of the BASIC (Beginner's All-purpose Symbolic Instructional Code) programming language.

This program, quite simply, prints the message “Hello world. This is my dissertation. (So many words, so few readers…)” onto the computer screen. There are many programming languages. Like with spoken languages, they differ in appearance but can be used to achieve the same results – to communicate a message to the computer. The same program written in a programming language called Java could look like this:

```
public class linusDissertationExample {

  public void main(String[] args) {

       System.out.print("Hello world. This is my dissertation.
(So many words, so few readers...)");

    } }
```

While the result would look identical when seen on a computer screen, the way it is achieved – the source code – differs. The important aspect for this context is that source code, like any language, can be read and understood by users who have learned the language. And furthermore, that source code as a concept should be no more intimidating than a list of groceries. It is simply a way of telling the computer what to do.

Before a program is used, or is distributed for general use, it is translated into a language that the computer itself can read. This process is called compiling. A computer program called a compiler takes the human-readable source code we have written and translates it into executable machine code, commonly called “binary” (from the Latin *Bini* meaning “twofold” or “two together”, as it consists solely of 1’s and 0’s). However, consisting of row upon row of nothing more than various combinations of the numbers 1 and 0, machine code is much more difficult – if not near-impossible – for humans to interpret.

2.1.2. The briefest of history lessons, part 1: software

Looking at the array of software for sale today, it is perhaps interesting to learn that software has not always been a business in its own right. During the first several decades of computing, when computers were massive in both size and cost, software was not a stand-alone business but rather a product that hardware suppliers knew needed to exist in order to sell or rent their hardware.[8] The software needed to run a computer was commonly bundled with the computer, meaning anyone who bought or rented a computer received software for it free of additional charge. Software enthusiasts modified and improved hardware suppliers’ software, as well as wrote their own programs, and distributed these freely to other interested users. Hardware suppliers were positive towards this practice, as it added to the usefulness of their product – the computers themselves. Having a wealth of software available for a

[8] Renting was initially more common than buying; in fact, IBM initially only allowed customers to rent their computers. A lawsuit settlement in 1956 resulted in the requirement that IBM also allow customers to buy their computers, not just rent them (Pugh, 2002).

particular piece of hardware was a significant selling point (Grad, 2002; see also Levy, 1974).

Spacewar!, one of the first computer games ever written, offers an example of how software evolved and was shared in the early days of computing. *Spacewar!* Was first written by Steve Russell in 1962 on a PDP-1 computer. Upon completion, he willingly redistributed the source code, which came to be copied and modified by numerous people. On the spread of the *Spacewar!* code, Russell (personal communication, 21 June, 2013) notes:

> I gave out several copies of the code for people to study, and after May of 1962 the sources were in the DECUS library. I don't know how many requests they processed. I believe that Bob Saunders did a PDP-7 version using the PDP-1 sources as a guide.

As it was long before the Internet (and even several years before the ARPANET), sharing was done in a more laborious, primitive manner. When asked about the process of sharing code, Russell (ibid.) notes that he would send paper tape sources:

> Printing was time-consuming on Flexowriters (10 characters a second) and copying was time-consuming. Copying paper tape was faster and easier.

The "paper tape" in question was a thin paper strip, like ticker tape, with the source code imprinted using holes punched into the paper. Anyone interested in the program could then borrow the tape from the DECUS library to study or install the program. However, Russell (ibid.) believes that the game spread more by programmers hearing about the concept and coding it themselves than by borrowing existing source code:

> I'm pretty sure that the majority of Spacewar! copies were done by seeing or hearing about Spacewar! and then figuring out how to do it on the hardware that they had at hand.

The game spread and grew in various ways: through direct copies of the code as well as through implementing the same idea from scratch, and being ported to a number of other computers. These various versions of *Spacewar!* then also evolved in different directions, with people coding their own features into their versions (or, rather, the version on their university's computer, as this was before the era of the personal computer). *Spacewar!* was significant because it helped showcase what computers were capable of and also became the inspiration behind the first coin-operated arcade game (e.g. MacDonald, 2014).

A pivotal time in the history of software came in 1969, when IBM decided to unbundle its hardware and software,[9] making the software for their computers a market of its own (Grad, 2002; IBM archives; see also Välimäki, 2005). The concept of software licenses and proprietary licensing was not IBM's invention alone, but rather something that was becoming increasingly popular as the second half of the twentieth century unfolded. By the early 80s, AT&T had launched a commercial version of the operating system Unix. Indeed, the rise in popularity of restricting access to source code was the impetus for the birth of what was to become open source software (e.g. Williams, 2002). The rise of proprietary licensing stands in stark contrast with the rise of the subculture from which open source software stems: a group that came to be known simply as "hackers".

---

[9] Largely as a pre-emptive move to avoid a monopoly lawsuit by the Department of Justice, notes Grad (2002).

2.1.3. The briefest of history lessons, part 2: hackers and the hacker ethic

A group of people who liked to tinker, examine, solve, and improve coalesced around the new phenomenon of computing. One large gathering of such likeminded people formed at the Massachusetts Institute of Technology (MIT). This group in particular, but also others with similar interests and world-views came to be called hackers. The hacker ethic (originally documented by Levy [1974], and later also covered by Raymond [1997 & 1998], and Himanen [2001]) valued ideals such as freedom of information, sharing, life-long learning (including, but not restricted to, hands-on learning at a computer), and community[10] (see Levy, 1974: chapter 2).

The rise of proprietary licensing clashed with the hacker ethic in several respects, not least when compared with the hacker sentiment that "no problem should ever have to be solved twice" (Raymond, 1997). Proprietary software was seen as erecting "artificial technical, legal, or institutional barriers [...] that prevent a good solution from being re-used and *force* people to re-invent wheels" (ibid.; emphasis in original). Problems, once solved, would have to be solved again by others not using the same software, as the solution was locked away behind a proprietary license. With proprietary licensing, information was no longer universally free to share. Among the more notable (and oft-told) stories regarding this time is Richard Stallman's attempts, in 1980, to address a problem with printer jamming. He was ultimately unsuccessful, as he did not have access to the printer's source code.

It was the birth of proprietary licensing that sparked the birth of free software, from which open source software was born (as will be discussed further in section 2.2). It was born not to introduce a new concept, but rather to maintain the freedom of development and sharing that, until then, had been the norm. Richard Stallman, who had been at MIT since 1971 and thus been a part of (the tail end of) the MIT hacker group at its prime, became concerned with the rise of the proprietary license. Stallman (1999) likened it to an enforced collapse of community: "[with proprietary software,] the first step in using a computer [is] to promise not to help your neighbour". In 1984 he left MIT and started the GNU Project,[11] whose goal was to create a free operating system called GNU.[12] In 1985 he founded the Free Software Foundation to help support the free software movement with the goal of promoting a new software sharing community (ibid.).[13] During the early years of the GNU project, Stallman wrote unique licenses for each program (e.g. Tai, 2001). In 1989 he wrote the GNU General Public License (GPL), a license still in widespread use today, which was designed to enforce the openness of a program's source code (open source licenses are discussed further in section 2.3).

2.1.4. Back to square one: What is open source software?

Open source software can be seen as a return to how software was developed before the widespread adoption of proprietary software licensing. While open source software is perhaps best known for being available free of charge, what makes it unique are the

[10] As well as practical jokes, at MIT in particular: something that came to be called "hacks" (see, e.g., http://hacks.mit.edu/, accessed 27 September, 2013).

[11] A project he had announced the previous year (27 September, 1983) in a Usenet newsgroup (see Stallman, 1983).

[12] GNU is a recursive acronym for "GNU's not Unix" (Unix is another operating system).

[13] The GNU Project and the free software philosophy are detailed in the GNU Manifesto (Stallman, 1985).

freedoms it offers beyond that of price. Among the central rights of open source software are the rights to access the source code, modify the program, and redistribute it, either in its original or modified form. Though there are different kinds of open source licenses, it is these basic rights that are the foundation upon which open source is built. These rights also enshrine both the challenges[14] and successes of open source. Indeed, open source software has been found to be of higher quality than proprietary software (Coverity, 2013).

For those interested in a more technical, detailed definition of what open source software is, one must refer to the Open Source Initiative. They maintain a list of approved open source licenses, as well as the Open Source Definition (included in Appendix 1).

2.2. Of purists and pragmatists: free software vs. open source software, and their use in this book

I have so far spoken mainly of open source software, with only a brief mention of free software. In actuality, these two related concepts coexist.[15] While both have their own definitions and requirements as to what constitutes "free" or "open source" software, the practical difference between the two is negligible as far as licensing is concerned.[16] The main dividing factor is in the view of why access to the source code, as well as the other rights guaranteed by such licensing, are important. To continue in this chapter's tradition of painting somewhat simplified pictures of a whole, free software enthusiasts see the openness of the code to be of primary significance, while open source software enthusiasts see open source as a superior development model: a way of achieving the best possible code. Author and technology writer Glyn Moody (2001) sums up the differences in categorizing the approaches, or camps, as *purists* and *pragmatists*, with the purists favouring the definition free software and the pragmatists open source software.

Proponents of the term "free software" commonly value, and underline, the freedom guaranteed by the license. This freedom does not refer to the program being free of charge – indeed, the right to charge for a program is seen as an important means of raising funds for free software development.[17] Rather, "free" refers to the freedoms guaranteed the user – the central rights noted in the previous section (2.1.4.).

The term "open source" was coined in February of 1998, and the Open Source Initiative was founded later that same month, with the goal of educating and advocating for the superiority of an open development process.[18] The founders believed that companies

---

[14] Here I am referring in particular to the challenges of financing an open source software development, a topic discussed further in section 2.4.

[15] There is actually a third term: "*libre*" software. This is sometimes used to avoid the ambiguity of the word "free". Additionally, there are geographical areas in which *libre* is more commonly used than free.

[16] See, e.g., http://www.gnu.org/philosophy/categories.html, accessed 7 February, 2014.

[17] From "the GNU Project", by Stallman: *"Since 'free' refers to freedom, not to price, there is no contradiction between selling copies and free software. In fact, the freedom to sell copies is crucial: collections of free software sold on CD-ROMs are important for the community, and selling them is an important way to raise funds for free software development. Therefore, a program which people are not free to include on these collections is not free software."* Source: http://www.gnu.org/gnu/thegnuproject.html, accessed 14 April, 2014.

[18] See: http://opensource.org/history, accessed 10 February, 2014.

participating with the open source community could further improve open source software development. Thus, we see that already at the time of founding, the idea of collaboration between the open source community and companies was not only present, but also pursued. While the Free Software Foundation is not against making a profit on software or its development (in fact, they encourage people to charge "as much as they wish or can"[19]), businesses and entrepreneurial programmers seem to more commonly associate themselves with the term open source than free software. This is perhaps due to the free software enthusiasts' dedication to freedom, and in particular their fondness of using licenses like the GPL that strongly enforce these freedoms, something that some developers find deterring (see, e.g., Aslett, 2011a & 2011b).

While we may consider free and open source enthusiasts separate groups, their efforts are often combined. One example is the GNU/Linux operating system. At the time of writing, the GNU kernel, part of the GNU operating system, has yet to be completed. However, the other elements of the GNU operating system are commonly combined with the Linux kernel. The resulting operating system owes much to the Free Software Foundation's GNU project, but is commonly known simply as Linux (to Stallman's chagrin[20]).

In this book, I use the term open source to include free software as well. Alternative solutions would have been to use terms like "free and open source software", "free/libre and open source software", "F/LOSS", or "FOSS". In the interest of reader friendliness, and with apologies to free software enthusiasts, the term open source is used as a synonym for all of the above.[21]

2.3. Software copyright and open source licenses

To understand the complex world of open source software and its licensing as they relate to the business side of open source software, we must also understand two significant aspects of software copyright more generally, namely ownership of code and the right to choose a license for the code one owns. Thus, we will begin this section with a short detour into software copyright more generally. This will be followed by brief discussions about open source software licensing, types of open source licenses, and the significance of license compatibility, i.e., questions relating to the combining of programs under different licenses.

2.3.1. Software ownership and licensing

In theory, whoever writes a computer program owns the code. In practice, a programmer may be employed to write the code, in which case its ownership can go straight to his or her employer. Code ownership brings with it two important rights: i) the right to decide how to license the program (i.e. what license to choose); and, ii) in addition to the right to sell copies of the program, the right to sell ownership of the code to someone else. These issues are explained below (and their significance is discussed further in section 2.4).

---

[19] See: https://www.gnu.org/philosophy/selling.html, accessed 9 February, 2014

[20] See, e.g., http://www.gnu.org/gnu/linux-and-gnu.html, accessed 14 April, 2014.

[21] This is not a choice unique to this book. E.g. Kelty (2008) begins the book *Two Bits* by stating: "This is a book about Free Software, also known as Open Source Software".

The right to choose a license

The owner of a program can choose what license to release it under. However, the owner of the original code, in this example the programmer, continues to own the rights to the original copy. One can think of this original copy as unlicensed: it is not altered or bound by the licenses that copies of the program have been released under. Whoever owns the code owns a copy that exists without license, or outside of license boundaries, and can make new releases as they see fit. Thus, the programmer is free to distribute multiple versions of the program under different licenses, all the while retaining the rights to the original, unlicensed version from which new copies can be distributed. The concept is visualized in Figure 1.

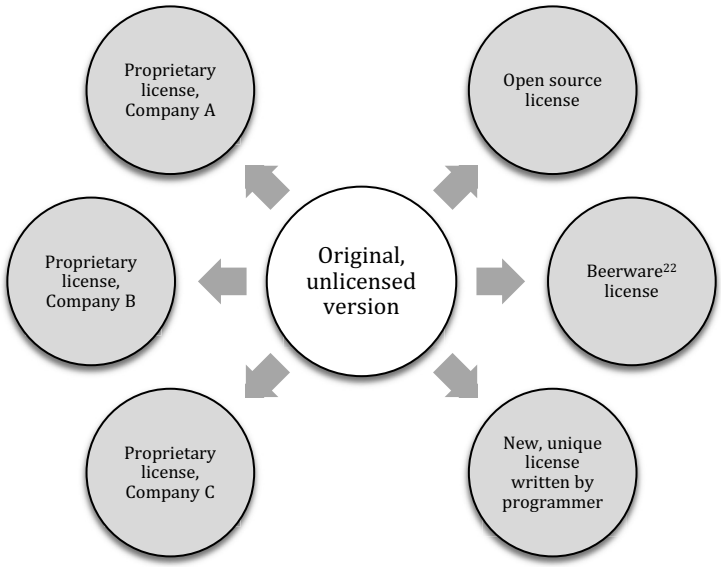


Figure 1 An original, unlicensed program and differently licensed versions released of it

The right to sell the program

The right to sell the source code is quite self-explanatory: in addition to the right to sell or give away copies of the program, one can sell or give away the rights one has to the original, unlicensed code. In such a case, the new owner is then the holder of the unlicensed version of the code, and can decide about its licensing. However, this does not affect previously released versions of the software. In other words, even if someone buys all the rights to a program, including its source code, they cannot alter the

[22] "Beerware" is a term sometimes used for very relaxed, permissive licenses. For instance, "THE BEER-WARE LICENSE", written by Poul-Henning Kamp, states (revision 42, in its entirety):
"<phk@FreeBSD.ORG> wrote this file. As long as you retain this notice you can do whatever you want with this stuff. If we meet some day, and you think this stuff is worth it, you can buy me a beer in return." (Source: http://web.archive.org/web/20060421154730/http://people.freebsd.org/~phk/, accessed 5 January, 2015)

licensing of previously released versions. What they can do, should they choose to, is to change the licensing of future versions. Thus, going back to Figure 1, the differently released versions of the program would continue to exist under the stated licenses regardless of any change of ownership of the original code.

2.3.2. Open source software licensing

The basic rights guaranteed by all open source programs (covered in section 2.1.4) are guaranteed with the use of an open source license. Where proprietary licensing is used to limit or restrict the ways in which software can be used (including modification and redistribution), open source licenses use licensing as a means of limiting the kinds of restrictions that can be put on the software. This is most evident in the so-called copyleft licenses, discussed below. In other words, the license is used to ensure the freedom of the code. Allowing the modification and redistribution of works is a prerequisite for both free and open source software licenses. Meaning, one is allowed to copy either part or all of an existing program and then redistribute it, either as is, or with changes one has made to it. Thus, all free and open source licenses guarantee the right to fork the code.

While there are hundreds, if not thousands, of different licenses that could be considered open source licenses,[23] at the time of writing the five most commonly used licenses made up just over three quarters of all license use, and the ten most common licenses over 90% of all license use.[24] Licensing has been noted to be a significant factor in open source adoption decisions (e.g. Daffara, 2011) as well as an important factor for the community (e.g. Gamalielsson and Lundell, 2014).

2.3.3. Types of open source licenses

Open source licenses can be divided into two main groups: permissive and copyleft (sometimes also called hereditary [Meeker, 2008], viral[25] [e.g. Mundie, 2001], reciprocal, or restrictive). The copyleft license can further be divided into two subgroups: strong copyleft and weak copyleft.

The ***permissive*** license, as the name implies, is the least restrictive of the license groups. It allows the code to be combined with any other type of license, either open source or proprietary. The permissive licenses also allow forked versions of the program to be relicensed under either a different open source license, or even a proprietary license.[26]

The ***weak copyleft*** is located in-between the permissive and the strong copyleft. The weak copyleft license allows combining with both strong copyleft and proprietary licenses. It does not, however, allow the re-licensing of the work under a proprietary license. Any modifications to the code must also be licensed under the same license (i.e.

---

[23] In that they comply with the Open Source Initiative's definition of open source. However, at the time of writing there were only 70 such licenses that were approved by the OSI. Source: http://opensource.org/licenses, accessed 24 March 2014.

[24] Source: the Black Duck Software knowledge base, available at: http://www.blackducksoftware.com/resources/data/top-20-licenses, accessed 5 September, 2013.

[25] For a discussion on why this term can be considered inaccurate, see Meeker (2013).

[26] In which case one can decide whether one chooses to distribute the source code with the binaries.

weak copyleft). The weak copyleft license is perhaps most commonly used for libraries, allowing non-copyleft programs access to, and use of, the information in the weak copyleft licensed program. The GNU Lesser General Public License (LGPL) is the most common of the weak copyleft license; the Mozilla Public License (MPL) is another.

The main unifying feature of the ***strong copyleft*** licenses is that certain obligations must "run with" the license, much like an easement would run with the rights in land (H. Meeker, personal communication, 2 September, 2010). Copyleft licenses stipulate that any derivative works must be licensed under the same license. In practice this means that if a program is licensed under a strong copyleft license, while – as is the case with all open source licenses – one is guaranteed the right to fork the program, one cannot relicenses it under either a permissive or proprietary license. One can also not combine a strong copyleft program with a permissively licensed program. At the time of writing, the GNU General Public License (GPL), a strong copyleft license, was the most common open source license (Black Duck Software, s.a.). Much like the LGPL is synonymous with the weak copyleft license group, the GPL is synonymous with the strong copyleft license group.

Copyleft licenses are triggered upon distribution. Unless and until one distributes an open source program, the license stipulations need not be followed. This means that one can download and modify open source code freely without being required to share one's modifications. One is furthermore not required to inform anyone that one has downloaded a program, that one has made modifications to it, or that one uses said program. If one is asked to share one's code, there is no legal obligation to do so. It is the act of distribution that triggers the requirements. Once a program is distributed, the stipulations of the license must be followed (e.g. distributing, or making available, the source code, license compatibility issues, etc.).

A variation of the GPL worth noting is the Affero GNU General Public License (AGPL). It differs from other open source licenses regarding the question of what constitutes distribution. The AGPL considers online use of a program to be distribution. In practice, this means that if one has an AGPL licensed program running online where users can access and use it, the license requirements are triggered and must be followed.

2.3.4. License compatibility

Software reuse is common in open source software (Haefliger, von Krogh, and Spaeth, 2008). However, while one is free to take (or fork) any open source program, what other programs one is allowed to combine it with will depend on the software licenses used by said programs. Thus, the issue of license compatibility – which licenses may and may not be combined with one another – is significant. Indeed, successfully managing this challenge, in particular in cases where software under both open source and proprietary licenses are sought to be combined, is seen as a prerequisite for success (e.g. Lokhman, Mikkonen, and Hammouda, 2013).

A piece of software may appear to the user as a single program, but can in fact be made up of several unique parts, licensed under different licenses. These need to be compatible with one another in order to be legally combined. Furthermore, there are many connections between different software programs that are necessary for, e.g., an application to function. When discussing these necessary pieces of software (e.g. operating system, libraries, middleware, etc.), it is common to speak of a software

stack. All of the software in a stack needs to have licenses that are compatible with the other software it is connected to. When programs under separate licenses are combined, the new whole effectively has the license of the least permissive of all the licenses used.

The right to combine one type of license with another is often directional, or one way compatible: while one license may allow combining with any other license, not all licenses will allow combining with the license in question. For instance, in the case of permissive and strong copyleft licenses, the permissive license allows combining with the strong copyleft license, but the strong copyleft does not allow combining with the permissive license. Furthermore, how the programs in question are combined and interact with one another can also affect compatibility (as is the case with dynamic vs. static linking[27]).

The rather complex nature of license compatibility is illustrated in David A. Wheeler's (2007b) diagram, presented in Figure 2. However, it must be noted that the open source license landscape continues to evolve as new licenses, and in particular new versions of popular licenses, are released. These can add new complexity to license compatibility. Specifically, regarding Wheeler's diagram, there are some compatibility claims for licenses that have not yet been released at the time of its publication (where a plus ["+"] sign is used to imply "or later"). For instance, while it can be presumed to be the case that future versions of the GPL will be compatible with AGPLv3, this is nonetheless not a certainty at the time of writing.

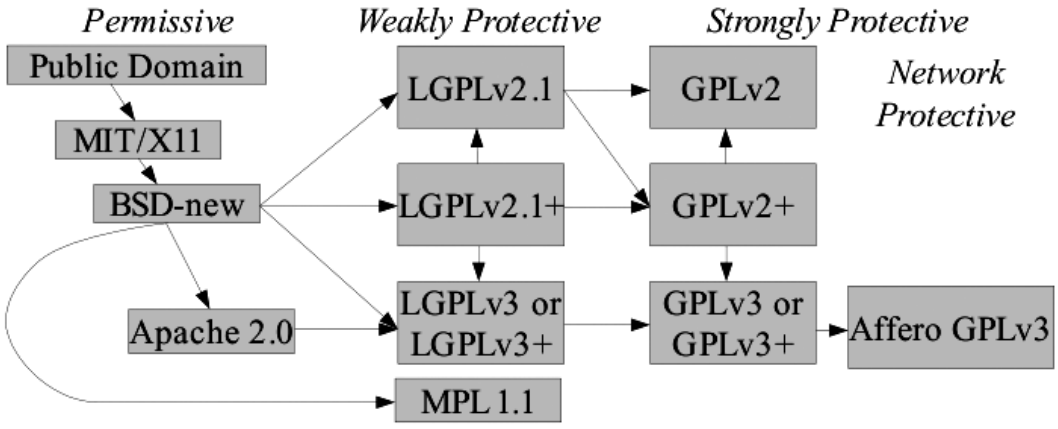


Figure 2 David A. Wheeler's (2007b) license compatibility diagram

Practices have evolved to circumvent license compatibility issues. Among these practices are that software components with different licenses are isolated from one another through architectural design. This can be done for instance by adding an additional program, that is compliant with both licenses, in-between the non-compliant

[27] Somewhere I must draw a line as to what is beyond the scope of this explanatory chapter. However arbitrarily chosen it may appear, I have drawn that line at explaining the difference between dynamic and static linking. Interested readers can find concise explanations e.g. on stackoverflow.com.

licenses,[28] allowing the communication of information between the two (Hammouda, Mikkonen, Oksanen and Jaaksi, 2010; see also Lokhman, Mikkonen, and Hammouda, 2013).

In any somewhat more in-depth discussion of open source licensing, it would be remiss not to include a disclaimer. Many open source licenses were written by programmers rather than lawyers, and can at times be somewhat ambiguous regarding specifics. This has led to some discussion regarding how these licenses should be interpreted, much of which would ultimately be decided through court cases. However, such court cases are rare. Thus, the legal rights and obligations of open source licenses, and the GPL in particular, may be considered a matter of interpretation. What I have reported on here are the commonly accepted norms that have evolved regarding the interpretation of various licenses. Furthermore, it must be emphasised that copyright and open source licensing are complex topics and only the most superficial of coverage has been afforded space here. Välimäki (2005) examines the rise of open source licensing, offering a valuable starting point for those seeking a deeper understanding of the topic.

2.4. The business of open source

While it may have been Stallman's desire for a free operating system that sparked the movement from which open source emerged, the phenomenon today has grown to embrace a broad spectrum of actors, among them an increasing number of corporations. Today, open source is increasingly commercial, as well as commercially developed (Wheeler, 2009). Furthermore, it has been noted that a majority of open source software development is done by companies (Weiss, 2011), and that open source has proven itself to be a powerful, new, competitive tool that can be leveraged to appropriate value (Carbone, 2007). Peters[29] (2007) notes that most companies have gone from asking themselves if they should use open source, to asking themselves how and where they best can take advantage of open source solutions.

There are many different kinds of corporate involvement in open source: some companies base their business models around open source software and its development (these will be discussed in the next section, 2.4.1); other companies may use open source software as well as contribute to its development; others still may merely use open source software without contributing (Hauge, Ayala, and Conradi, 2010; see also, e.g., Riehle 2009). Interestingly, even competitors may cooperate in developing a commonly used open source program. Plumbing is a common simile used in explaining why competing companies can work together to develop an open source program they both use and that their competitors can fork. Neither company positions itself as being unique due to its superior plumbing, but both companies need it to work as flawlessly as possible. Similarly, in software, working together to improve a common program (that is important for several companies but that is not in itself a unique selling point for any of the companies involved) benefits all parties and allows them to target more funds and manpower on the elements of their programs or services that make them unique.

[28] Or licensing a connector situated in-between the programs in question in such a manner as to allow for combining, for instance an LGPL connector in-between a GPL and a proprietary program.

[29] Executive Director of GNOME, an open source desktop environment and user interface; also, part of the GNU project.

There is a broad spectrum of motivations regarding why corporations are interested in participating in the development of open source software. Where developer motivations include many social motivations, Bonaccorsi and Rossi (2003) found that firms tend to emphasize economic and technological reasons for entering and contributing to open source. Technological reasons may include a wider adoption of code (West, 2003), faster identification and resolving of bugs (Schindler, 2007), standardization (Lindman, Juutilainen, and Rossi, 2009), and a superior product to proprietary counterparts (Wheeler, 2007a). Furthermore, Lundell and Gamalielsson (2011) identified open source as a viable strategy for implementing long-term sustainable software systems. There are significant economic gains to harnessing community efforts. Furthermore, it has been noted that proprietary software may, in the long run, be hard pressed to compete successfully in the same market with a complementary open source product (Lindman and Rajala, 2012). Indeed, releasing an open source version of a competitor's closed source product can be a competitive strategy (Fitzgerald, 2006).

The fact that open source programs are both free of charge[30] and free to be forked has a tremendous impact on the business of open source. The business models and approaches that have evolved around open source software have had to accommodate these rights, and thus differ in many respects from the business models found around proprietary software. A further area in which open source software tends to differ from proprietary software is in its architectures of participation – specifically, the harnessing of community contributions in development of the software. Though businesses are involved, the developer community is still considered vital to success. The remainder of this chapter will cover these two topics: business models and architectures of participation.

2.4.1. Business models

Business models can be seen as something more complex than merely a revenue source (e.g. West 2007; Bailetti 2009). At the heart of the issue of business models, and what I shall limit the approach in this dissertation to, is the question of how a firm can create value for their customers while simultaneously extracting some of that value for itself (West, 2006).[31] In open source, the choice of business model is closely tied to both license choice and business goals (e.g. Lindman, Rossi, and Puustell, 2011). There are a variety of different pieces that make up the puzzle that is a company's possible means of revenue (extracting value, in the words of West [2006]) through open source. These pieces need not be mutually exclusive. Indeed, many can be simultaneously used. In the following discussion, we shall limit ourselves to business models that generate income based on open source software and its development.

For a company that wants to create a business model around open source software, two central issues decide what business models are possible: i) ownership of code, and ii) the program's location in the software stack (see Widenius and Nyman, 2014). Ownership of code, as discussed in section 2.3.1, is significant in that it grants the right

[30] One is allowed to charge money for (both free and) open source programs. However, one cannot forbid one's customers from redistributing the same program either at a discounted price, or entirely free of charge. Thus, they are rarely not free of charge. An example of an exception, mainly from the time before high-speed Internet access became commonplace, would be buying a Linux distribution on CD or DVD.
[31] Thus, while some may consider the business models covered here incomplete descriptions of a "business model", they cover central elements of how value is extracted.

to choose what license (or licenses) to release the program under. Location in the software stack is significant in that the more programs in the stack that rely on the program to function, the more potentially valuable the program is.

If one owns the code then one can freely decide what license (or licenses) to release the program under, and all business models are available to choose from. However, if one does not own the code, then a support and services role is the only available approach. If one both owns the code and it is embedded, then dual licensing is an option. These business model selection criteria are depicted in Table 2.

Table 2 Business model selection criteria (source: Widenius and Nyman, 2014)

Criteria	Support and services	Open core	Business source	Dual licensing
Does this business model require ownership of the code?	No	Yes*	Yes	Yes
Does this business model require the program to be embedded?	No	No	No	Yes

*Ownership is required for the closed source extensions

A further important aspect to note regarding business models is that one's choice of license will not only affect customers' views and options, it will also have an effect on the community. As will be discussed in more depth in the coming section on architectures of participation, choosing a license at the most permissive end of the spectrum of license options may have a negative effect on community growth, while choosing a license at the strong copyleft end of the spectrum may have a negative effect on business growth (see, e.g., Widenius and Nyman, 2013).

Support and services

In this business model, the product is given away for free but income is generated by offering support, services, training, features development, etc. around the product. While this approach is challenging regarding scalability, it nonetheless has among its ranks one of the more oft-told open source success stories: for the past few years, the biggest provider of support and services in open source, Red Hat, has had a turnover of over a billion dollars.[32] (For more on Red Hat's business model and approach, see e.g. Young, 2002; Suehle, 2012.)

Open core, a.k.a. Commercial extensions

In this approach, part of the program – the "core" – is open source, with other parts of the program provided for a fee.[33] The approach attempts to deal with the challenge of

[32] E.g. USD 1.33 Billion in 2012-2013 (Red Hat, 2013a).

[33] There can be some ambiguity to the term, e.g. Apache gives away the entirety of their program for free, but third parties can provide proprietary add-ons to it. For the purposes of this dissertation, a company or

profitability in open source software development. However, as the approach involves closing off some of the source code, it raises strong emotions among some developers. Purists oppose it on the grounds of it not being open (or free), while pragmatists may oppose the limitation of not being able to see and access all of the source code, as well as the resulting vendor lock-in.

Business Source

Business source, originally put forth by Michael "Monty" Widenius (see Widenius and Nyman, 2013), implements two different licenses with a time-based, automatic license change. The source code is made visible and editable to all from the start. However, for a set amount of time a segment of users (defined by the developer) has to pay to use it. After this initial time period, the license automatically changes to an open source license. As long as the software continues to be developed and improved upon, the developers will maintain a steady income. Furthermore, corporate as well as community contributors alike can know that, with a delay of a few years, new and improved open source software will continue to be generated.

On the surface this license shares some similarities with a concept called delayed open sourcing, or time-limited hybrid licensing (e.g. Sprewell, 2010). In this model a program is initially licensed under a proprietary license, and later re-licensed as open source. However, Business source differs significantly in that it allows access to and modification of the source code even prior to the license change to open source.

There are some companies that have publicly noted their intention to switch to Business source.[34] Furthermore, Widenius has noted in personal correspondence with this author, that at the investment company Open Ocean Capital, there are ventures in their early development stages that plan to implement the license. However, it is still too early to tell if the Business source license will achieve noteworthy adoption or have a significant effect on the open source landscape as a whole.

Dual licensing

Dual licensing means offering a program under two separate licenses. Commonly one version of the program is licensed under a copyleft, GPL-style license and another under a proprietary license. A proprietary license is necessary, for instance, when one needs to embed the code with proprietary code. Embedding GPL licensed code with proprietary code would require the releasing of the proprietary code under a GPL-style license as well, whereas obtaining a proprietary version would allow the original code to remain proprietary. Thus, the primary customers for a company that dual licenses a piece of software are companies that need to be able to combine the program in question with their proprietary software. In practice, dual licensing is quite rare, since it requires both ownership of the code and a customer with a program that requires embedding said program. Furthermore, license compatibility challenges may also be addressed, e.g., by using LGPL connectors, thus arguably negating the need for dual licensing (as discussed in section 2.3.4).

---

foundation that provides all of the code they produce as open source is not considered to adopt an open core model, even if closed source extensions exist that are developed by others.

[34] Among them Baasbox (www.baasbox.com) has tweeted this intent.

2.4.2. Architectures of participation

While there are many different open source projects, they can be categorized broadly into two groups: those in which a community serves as the owner and driving force behind the project, and those in which a corporation serves that purpose (e.g. West and O'Mahoney, 2005; Markus, 2007; O'Mahoney, 2007; Riehle, 2007). A community-developed ("autonomous", West and O'Mahoney, 2008) project shares governance and control among its members, in some cases with a non-profit foundation created to support the project and delineate ownership. In a sponsored open source project, the project is controlled by one or more corporate entities (*ibid.*).

Figure 3 depicts a development model for a sponsored, or "corporate", open source project. While the corporation is the primary driving force behind the project, the project also receives ancillary support from both the open source community and the corporate community. The corporate community is commonly made up of companies for whom the development of the software is significant, typically because they use the software, but it is not a unique selling point in their service offering.

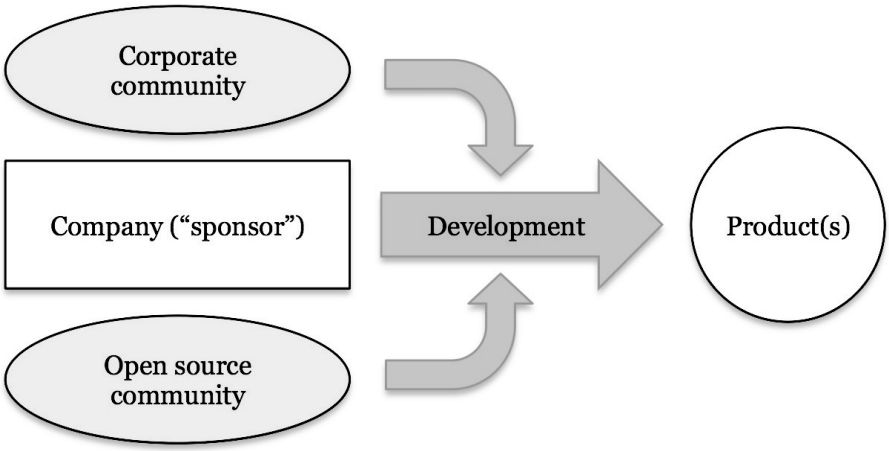


Figure 3 Example of a corporate ("sponsored") open source project development (from paper 3)

Members of the corporate community may contribute either money or developer time to aid the project. The means of participating in development may furthermore vary between projects, and are decided largely by how much control the sponsor company is willing to relinquish, and how open they choose to be regarding the project and its development. While both open source and corporate community members can participate in the development (e.g. through bug reports, bugfixes or new features), it is ultimately the sponsor company that decides what is included in the end product.

Sponsored and community projects differ both in their developer communities and in their architectures of participation (West and O'Mahoney, 2008). Among the tensions that need to be managed are those between sponsor and participant goals – between community desires and profit. When these goals come into conflict, a fork may ensue. Bacon (2009) notes that the governance body should be tasked with maintaining and defending the primary values of the community, as well as opposing any improper

requests that may result from commercial sponsors. As Lindman, Rossi, and Puustell (2011: 34) noted in their study of the interplay between business, developer community, and license: if a company has chosen open source in order to achieve competitive advantage, it must also address community developer motivations. Tirole and Lerner (2005) studied the determinants of license choice, finding that projects geared towards end users are more likely to have a restrictive license than those geared towards developers. In their study targeting the developer community of the LibreOffice fork of OpenOffice, Gamalielsson and Lundell (2014: 138) found licensing to play a significant role. During the time leading up to the fork, several contributors of the OpenOffice community explained that choosing a copyleft license rather than a permissive license was an important prerequisite for their joining the project.

While a community-led project may not have the same challenges regarding aligning goals and profit, the concern of a common goal is still very much a relevant point. Should a developer's project generate community interest, and the community by and large come to be of the opinion that the project should be taken in a different direction, the risk of a fork is great unless the original developer is able to incorporate these goals. A further challenge particularly relevant to a community-led project is that of securing enough funding to be able to employ full-time workers to help guarantee its continued development.

2.5. Summary

Open source software is software licensed under an open source license. Where proprietary software comes compiled and commonly without the source code, open source software provides users with the source code while guaranteeing the right to run, modify, and redistribute the program. The roots of open source can be traced back to the free software movement, and earlier still to the common practice of sharing code that was prevalent before the rise of proprietary licensing.

Open source licenses are commonly divided into three groups: permissive, weak copyleft, and strong copyleft. The permissive licenses, as the name suggests, allow for more freedom in the use and reuse of the code, as well as combining it with other programs (i.e. licenses). The strong copyleft lies at the other end of the spectrum and stipulates that, while the program can be modified, it must remain under a strong copyleft license. The weak copyleft lays somewhere in the middle of these two extremes.

Open source began as a community-driven phenomenon that sought to also attract corporate participation, and firms have since become a large part of open source software development. Open source licensing (and the right to fork) has necessitated business models that differ somewhat from those of proprietary software. Open source software projects are commonly maintained by a firm, a non-profit organization, or a community of developers.

3 CODE FORKING

This chapter covers the concept of code forking from various angles. The nuances in meaning of the term "fork" will be discussed throughout the chapter. A broad description of a fork is when a developer copies an entire existing program and then releases a new, modified version of it. The right to fork code is guaranteed by all free and open source licenses: if it is an open source program, you are allowed to fork it.

Due to the limited scope of extant research into the topic of forking at the time of writing, some significant points from practitioner literature (that are also covered in the articles of this dissertation) are brought up in this introductory chapter. While somewhat unconventional, this is done out of necessity in order to better illustrate key concepts of code forking.

3.1. The history of the term "fork"[35]

The word fork stems from the Latin word *furca*, meaning "a fork or similarly shaped instrument" (de Vaan, 2008). The earliest surviving use of the word is from *De Agri Cultura* ("On Agriculture"), which was written by Cato the Elder around 160 BC, and is the oldest piece of Latin prose to survive.[36] Unfortunately, there is no commonly agreed upon hypothesis among researchers of the field regarding the origin of the Latin word *furca* (E. Tucker, Faculty of Oriental Studies at the University of Oxford, personal communication, 5 December, 2012). However, for the purposes of this book we will concern ourselves with much more recent history. Specifically, how the word fork entered the world of computer programming, from which it later found its way (albeit in an altered meaning) to the world of open source software.

3.1.1. The fork system call is proposed

The year was 1962. Melvin Conway, later to become known for "Conway's Law",[37] was troubled by what seemed an unnecessary inefficiency in computing. Conway (personal communication, 2 November, 2012) notes:

> I was in the US Air Force at the time involved with computer procurement, and I noticed that no vendor was distinguishing between "processor" and "process". That is, when a proposal involved a multiprocessor design, each processor was committed to a single process.

The concept of a processor only being able to work on one process, particularly in the case of a single-processor computer, was problematic for reasons that those who have been introduced to computing in more recent years (with immensely more powerful computers) have never experienced. For instance, it meant that the computer could not do any two things at once. Any given process had to be finished for the computer to be able to do anything else. For example, at any given time, you could either print a document or write one, but not both.

---

[35] Special thanks go to Walt Scacchi for suggesting the idea of tracking the roots of the term fork.

[36] See, e.g., The Oxford Companion to Classical Literature (3 ed.), available at: www.oxfordreference.com.

[37] "Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure." The thesis (or law) was originally part of an article published in Datamation (Conway, 1968), but was popularized, and given its name, by Brooks (1975).

In early 1963, as "an intellectual exercise" (ibid.), Conway wrote an article suggesting a means of implementing parallel processing in a multiprocessor system design, using what he called "fork" and "join" system calls (Conway, 1963 & personal communication, 2 November, 2012). Ideas of a similar nature had been put forth before,[38] and Conway himself (personal communication, 2 November, 2012) notes:

> I do not claim to have invented that idea, since it must be present in all implementations in some form or other. The use of associative memory for address mapping is almost certainly not original with me either; I think it was in the air at the time. Vendors were pushing associative hardware memory on the military for database search (which was impractical), and I adopted the idea for address mapping. But it may have already been in use elsewhere.

However, to the best of this author's knowledge, Conway was the first to suggest the term "fork". Furthermore, Conway's 1963 conference paper was the first instance found of "fork" in print in this context.[39] The inspiration behind the choice of the term came from the expression of a "fork in the road" (M. Conway, personal communication, 2 November, 2012). The original figure picturing the fork is shown in Figure 4, as it appeared in the article "A multiprocessor system design" (Conway, 1963).

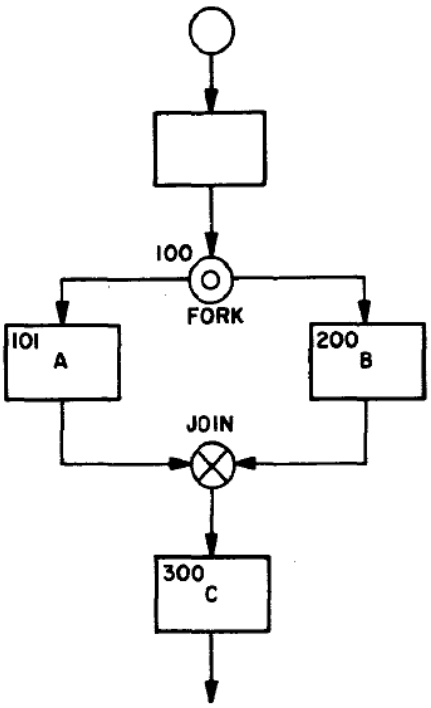


Figure 1. Conventions for drawing fork and join.

Figure 4 The fork system call as originally suggested by Conway in 1963

[38] Conway (1963) cites the use of the term "BRANCH" for a similar notion by Cheatham and Leonard (1963) and Thompson and Wilkinson (1963); the term SIMU by Dreyfus (1961); and BRT (Branch Transfer) by Richards (1960).

[39] It is not my intention to decide where the credit for the idea of the fork system call is due. Rather, to identify (and give credit for) the first use of the term fork. On the use of fork, Conway (personal communication, 2 November, 2012) also notes: "As far as I know, I was the first to apply the word in this technical context."

The basic idea, simplified, was that the program would make a copy of itself, allowing two separate functions to be done at once. When completed, they would be joined together again, hence "fork" and "join". Conway presented the article later that fall at the American Federation of Information Processing Societies (AFIPS) fall joint computer conference and it was published in the conference proceedings. Conway (personal communication, 2 November, 2012) notes that he didn't recall the paper causing much stir or excitement at the conference. However, a few short years later, the article was to have a profound impact on computing.

3.1.2. The fork system call is born

One of the people who read and was inspired by Conway's article was Melvin Pirtle, co-leader of a computer research project called Project Genie (W. Lichtenberger, personal communication, 1 November, 2012). Project Genie was started in 1964 at the University of California, Berkeley. Project Genie and the SDS 940 computer that it produced (a modified SDS 930) are perhaps best known for being the first to implement time-sharing (see, e.g., Spinrad and Meagher, 2007). However, the legacy that came to pervade computing was the fork system call, the first ever implementation of which was in Project Genie's SDS 940 (ibid.).

Conway's article found its way to Project Genie,[40] and the fork system call was implemented in Project Genie by L Peter Deutsch (B. Lampson, personal communication, 27 September, 2012; L P. Deutsch, personal communication, 2 November, 2012). However, Deutsch noted in his communication to this author, he used a somewhat different technical means than originally suggested by Conway. It was from Project Genie that the fork system call found its way to the operating system Unix (Ritchie and Thompson, 1974; K. Thompson, personal communication, 27 September, 2012).

Despite several attempts, I was not able to obtain a copy of the original source code from Project Genie.[41] However, the source code from Tymshare is available online. Dag Spencer, Curator of the Computer History Museum, notes (personal communication, 24 November, 2012) that Tymshare's code was derived from the Project Genie source code. Butler Lampson kindly took a brief look at the Tymshare code and noted: "it looks to me as though most of it is untouched from the Berkeley system code" (B. Lampson, personal communication, 27 September, 2012). Like Project Genie, Tymshare was written in assembler, a low-level programming language. The earliest, if not only, publicly available copy of the source code is from a Tymshare document from July of 1967,[42] a section of which is shown in Figure 5.

---

[40] Though the exact route is uncertain: Wayne Lichtenberger (personal communication, 1 November, 2012) notes that Pirtle "made several mentions of the Conway paper". L Peter Deutsch (personal communication, 2 November, 2012) notes that the article may have come to his attention from a pre-print he had of an article (the article in question being Dennis and Van Horn [1966]) in which Conway's article was cited.

[41] It is some small consolation that the source code would in any case most likely have been on tape, and thus unreadable by any machine I would have had access to.

[42] The Tymshare Monitor code survives in Bitsavers's archives, and is available at: http://bitsavers.informatik.uni-stuttgart.de/pdf/sds/tymshare/listing/Tymshare_Monitor_Jul67.pdf.

```
*
*       FORK LOGIC
*
* FIND HIGHEST FORK IN STRUCTURE
HFK     ZR0
HFK1    LDA PPTR,2; SKA PRMSK; BRU *+2; BRR HFK
        MRG PLMSK; CAX; BRU HFK1
* GET FORK ENTRY. PUT PPB INTO PIM.
GFK     ZR0; LDA FPLST; SKG =0; BRR GFK
        SUB =PPTR; COPY AX,A
        XMA PPTR,2; STA FPLST; MIN GFK; BRR GFK
* PUT NEW FORK ON QIO
* SET PDOWN(OLD)=NEW, PDOWN(NEW)=0
*     PFORK(NEW)=OLD, PPAR(NEW)=PDOWN(OLD)
STFK    ZR0; STX FK04; SKR NFORK; BRU *+2; BRM MONCR
        LDX PPB; LDB PB,2; STB PPB; CXB; STB STFK2
        LSH 3; LDX FK04; STB PIM,2; CXA
        LDX =QIO; BRM QPUT; LDX PACPTR; LSH 12
        ETR PLMSK; XMA PPTR,2; CAB; ETR PRMSK; ADM PPTR,2
        LDA FK04; XXA; ETR PRMSK; STA PPTR,2; LDA PACDMB; STA PTEST,2
        LDX PACPTR; CLA; RSH 3; LDA QUTAB; LSH 15; BRR STFK
STFK2   ZR0 0     XPB FOR NEW FORK
```

Figure 5 "Fork logic" in Tymshare (derived from Project Genie)

The process was called a fork, but due to the linguistics of the programming language the actual word fork only existed in the comments section of the code as well as in the user manual. Similarly, in Unix, the word fork first appeared in the comments rather than the actual source code. Figure 6 shows the fork system call as presented and described in the Unix Programmer's Manual, version 1, dated November 3, 1971.[43]

```
11/3/71                                                SYS FORK (II)

NAME            fork  --  spawn new process

SYNOPSIS        sys     fork    / fork = 2.
                (new process return)
                (old process return)

DESCRIPTION     fork is the only way new processes are created.
                The new process's core image is a copy of that of
                the caller of fork; the only distinction is the
                return location and the fact that r0 in the old
                process contains the process ID of the new pro-
                cess.  This process ID is used by wait.

FILES           --

SEE ALSO        sys wait, sys exec
```

Figure 6 The fork system call described in the Unix Programmer's Manual from 1971

For the first implementation of the word fork in the actual source code, "fork()", one must jump ahead to 1973, the year Unix was rewritten in the programming language C (see, e.g., Ritchie and Thompson, 1974).[44]

---

[43] Source: http://cm.bell-labs.com/cm/cs/who/dmr/pdfs/man21.pdf, accessed 20 November, 2012.

[44] Establishing an exact date for when the fork system call would have first existed in C is challenging. Regarding Research Unix, C was introduced in the third edition and edition 4 was rewritten in C.

3.1.3. Forking takes the fork system call global

In addition to popularizing the fork system call, Unix was also the first program to be forked on a large scale. Unix was first created in 1969 by Kenneth Thompson and Dennis Ritchie[45] (Ritchie, 1979; see also Ritchie and Thompson, 1974), computer scientists at Bell Laboratories (also known as Bell Labs). At the time, Bell Labs was a division of the American Telephone & Telegraph (AT&T). AT&T had been granted a government-sanctioned monopoly of the telephone business, but was forbidden from entering the computer business.[46] Thus, AT&T could not charge money for Unix. As a result, copies of the operating system were distributed freely upon request.

This was a time before the personal computer. Large, powerful (for their time) mainframe computers were expensive and rare; gaining access to somewhat smaller, less powerful minicomputers was easier. Unix was unique in that it combined the aspects of requiring less of the computer than other operating systems, with offering a greater level of freedom for the programmer. Raymond (2003) notes:

> Unix development rapidly took on a countercultural air. It was the early 1970s; the pioneering Unix programmers were shaggy hippies and hippie-wannabes. They delighted in playing with an operating system that not only offered them fascinating challenges at the leading edge of computer science, but also subverted all the technical assumptions and business practices that went with Big Computing. Card punches, COBOL, business suits, and batch IBM mainframes were the despised old wave; Unix hackers reveled in the sense that they were simultaneously building the future and flipping a finger at the system.

Unix was further aided by the publicity that came with Ritchie and Thompson's 1974 paper and conference presentation about the operating system. Many universities began contributing to Unix, chief among them the Berkley campus of the University of California. They released their own version (or fork) of Unix, Berkley Software Distribution (BSD, sometimes called Berkley Unix), in 1977.

In 1983, the U.S. Department of Justice won an anti-trust lawsuit against AT&T (put into effect on 1 January, 1984), freeing AT&T from its ban on entering the computer industry. AT&T commercialized Unix, in a version known as UNIX System V. Thus (at the latest) began what came to be known as the Unix Wars. Initially, System V and BSD vied for attention and code commits. However, the commercialization of Unix also led to a decrease in willingness to take part in Unix's development due to fears of litigation, as AT&T began clamping down on source-code distribution (Raymond, 2003). Eventually, through both settlement in court and the re-writing of much of BSDs code to remove proprietary code owned by AT&T, an open source BSD distribution of Unix was completed.[47] In the early 90s, the operating system Linux was first introduced, and soon (due in no small part to in-fighting, lawsuits and legal concerns related to Unix, as well as the Internet explosion) gained the attention of many Unix developers. Linux, like Unix, also incorporated the fork system call. To date, both Unix and Linux have

---

[45] The name "Unix" was suggested by Brian Kernighan in 1970, and was a pun on the operating system Multics (Ritchie, 1979).

[46] In a 1913 agreement known as the Kingsbury Commitment, the United States government accepted the granting of a government regulated monopoly position to AT&T on the grounds that the telephone, by the nature of its technology, would operate most efficiently as a monopoly providing universal service. In a 1956 consent decree (to end an anti-trust suit brought against AT&T), AT&T agreed to restrict its activities to the regulated business of the national telephone system and government work (AT&T website; see also Thierer, 1994).

[47] The BSD family of forks is covered briefly in sections 1.2 and 3.4.

spawned, at the very least, hundreds of forks (see, e.g., Lévénez, 2013; Lundqvist, 2012).

This summary does not attempt, in any meaningful detail, to cover either the history of Unix or the birth of Linux. Rather, it aims to show that it was the increasing popularity of Unix, as well as Unix-like operating systems and their many flavours, that helped make the fork system call a household feature of operating systems worldwide, thereby also popularizing the term "fork" among programmers.

3.1.4. Forking forks to open source

Pinpointing when the word "fork" first started showing up in open source is challenging. As early as 1980, the term fork was used by Eric Allman in describing the creation of branches of the revision control system Source Code Control System (SCCS). In section 12 of "An Introduction to the Source Code Control System", entitled "Maintaining Different Versions (Branches)", Allman (1980) notes:

> Creating a branch "forks off" a version of the program.

Here "branch" refers to a process in revision control in which a copy (i.e. branch) of a program is made, which can then be modified in parallel with the original. In today's open source vernacular, such a branch is sometimes also called an experimental fork. Allman (ibid.) explains the practice in this way:

> Sometimes it is convenient to maintain an experimental version of a program for an extended period while normal maintenance continues on the version in production.

Thus, the process of creating what is today commonly called a fork had been called forking or to fork off ("fork off a version") by at least 1980. But once created, the project was called a version or branch (rather than a fork), at least when it was of the experimental kind described by Allman. (Allman was contacted[48] to see if he could shed some light on the matter, or had any recollections of the early use of the term fork, but did not reply.)

At least as early as 1983, the term "fork off" was used for the process of creating a new discussion group on Usenet, as seen in a post by John Gilmore titled "Can somebody fork off a 'net.philosophy'?" (Gilmore, 1983). However, in the 1993 post "Shattering – good or bad" by Russell Nelson (1993), Gilmore is credited for coining the term "shatter" for what would later come to be called a fork:

> We've all seen it. You distribute a system in source form, and multiple revisions result, each with a different feature set. John Gilmore coined the term "shattering" for this phenomenon.

Nelson was contacted to see if he had any recollection of the terminology regarding forks, and when the term began appearing, but he did not.[49] Gilmore was also contacted[50] to see if he could recall any specifics regarding the use of the term fork at the time, but no reply was received.

---

[48] Email sent 30 November, 2012.

[49] His reply, *"No, I don't, sorry."*, was received 5 November, 2012.

[50] Email sent 29 January, 2014.

By 1995, the term "fork" can be found in what was to become the common meaning for the term. In a discussion thread entitled "Hey Franz: 32K Windows SUCK!!!!!", Bill Dubuque's reply[51] includes the following:

> Win-Emacs is a MS-Win port of XEmacs 19.6 (a cutting-edge fork of GNU Emacs version 19).

Furthermore, in a 1996 essay entitled "Linux and the GNU System", Stallman (1996) used the term fork in the same vein:

> But people who think of themselves as «Linux users» are more likely to release a forked «Linux-only» version of the GNU program, and consider the job done. We want each and every GNU program to work «out of the box» on Linux-based systems; but if the users do not help, that goal becomes much harder to achieve.

Additionally, in 1997, when announcing the start of the EGCS project (covered in section 1.2), Henkel-Wallace (1997) noted that its goal was "to merge the existing GCC forks". Thus, we know that by the mid-1990s the term had, at least by some, been adopted in what was to become its original common meaning of a (more or less) competing version split off from the original. That Stallman was one of those who used the term in such a meaning is likely to have further strengthened its adoption.[52]

3.2. The definition of fork

The practice of defining a fork is more problematic then it might appear. Included here is a brief overview. The issue is explored in greater depth in paper 5.

At its most brief, a fork has been defined simply as being a new version of an existing open source project (e.g. Moen, 1999). A fork is sometimes defined as the result of irreconcilable differences among a developer team, leading to two different versions – implying that part of the definition is a requirement that the new version be started by some of the original developers.[53] Incompatibility between the original program and the new program is sometimes included in the definition of a fork and often noted to be a largely unavoidable result of forking (e.g. Raymond, 1999; St. Lauren, 2004; Weber, 2004; Meeker, 2008). Robles and González-Barahona (2012) offer the most detailed definition of a fork among existing literature, defining a fork as occurring when "a part of a development community (or a third party not related to the project) starts a completely independent line of development based on the source code basis of the project." They include four further criteria for something to be called a fork. It should have: i) a new project name; ii) a branch of the software; iii) a parallel infrastructure;[54] and, iv) a new developer community, disjoint with the original.

For the purposes of this dissertation, a fork is taken to mean *any instance in which the code of a program is copied, modified, and reused to start or develop another program*. A project that is based on another project but whose code has been entirely rewritten can also be considered a fork. The term "competitive fork" is used to define instances in which a fork is created with the intention (either explicit or inescapable) of

---

[51] Available at: https://groups.google.com/forum/#!topic/cu.cs.macl.info/7Z5f-cshw1k, accessed 22 May, 2014.

[52] As an example, Stallman's use of the term fork was also included in a quote included in a post by Russ Allbery (1996) in a discussion thread on the question of a naming policy for GNU and Linux.

[53] See the Jargon File: http://www.catb.org/jargon/html/F/fork.html.

[54] Web site, versioning system, mailing lists, etc.

competing with the original for developers or users or both. The notion of competitive and non-competitive forks is discussed further in section 3.5.

Having stated the definition used, one must follow up with some of the many challenges of defining a fork. The term fork is undergoing what is known as a semantic change: a change over time of a word's use and meaning. Semantic change is a common occurrence in the history of a language's evolution, however, it makes the job of unifying the definitions and use of a term challenging. In the case of the term fork, this semantic change appears to have been noteworthy in both alacrity and scope. As a result, there is overlap between what is categorized here as a fork, and other terms that are (or have been) more or less commonly used for similar (or identical) actions. For instance, the definition used in this dissertation shares many similarities with what has commonly been called a "branch." Furthermore, in its broader interpretations, fork becomes largely synonymous with large-scale code reuse.

In addition to the term fork broadening to encompass events that are also known under different terms, there are an increasing number of prefix-based variations on the term fork. By way of example, the term "community fork" or "community-developed fork" can refer to a fork of an open source program that was done, and is maintained, by members of the community of the program from which it was forked (e.g. Drizzle from MySQL). However, the uses of prefixes to the term fork are not always clearly defined. For instance, Fung, Aurum and Tang (2012) introduce the term "social fork" as a phenomenon that includes community, software, as well as other artefacts. However, how this differs from other forks such as a community fork is not clear, nor is it clear what the specific requirements are for the term to be used.[55]

Looking at terminology from a purely technical perspective, specifically the fork system call, there are terms in common and undisputed use. A system call initiates the duplication of a process (i.e. an instance of a computer program that is being executed). This new copy, created by a fork system call, is called the "child". Conversely, the original is called the parent. While this author has not come across the widespread use of parent and child in discussing the forking of programs, their use is nonetheless also possible.

3.3. How does the right to fork affect open source software?

Forking, and indeed even the mere existence of the right to fork, affects open source software profoundly. While forking can be seen as the invisible hand of sustainability within open source, forking is also paradoxical in that it has at once the potential to both save and destroy an open source project (Nyman et al. 2011 & 2012). The effect of code forking on open source software will be analysed on three different levels: the software level, the community level, and the business ecosystem level (from Nyman and Lindman [2013], which is paper 2 of this dissertation).

[55] This author has also come across the terms "internal" and "external" fork, used to distinguish between forks done by members of a project versus those done by outside developers. However, attempts to track down the source of this prefix, or indeed uncover a widespread use of it, were unsuccessful.

3.3.1. Software level

The first of Lehman's laws of software evolution (see, e.g., Lehman, 1980; Lehman, Ramil, Wernick, Perry, and Turski, 1997) is that of continuing change: that any program written to perform some real-world activity must be continually adapted or it will become progressively less satisfactory. Furthermore, Lehman notes that such systems must also continue to grow in functional content (Law 6) and be rigorously maintained (Law 7), or user satisfaction with the program will decrease. Regarding the evolution of a software program, Brooks (1975) notes that successful software adapts to outlive the hardware it was originally written for. Software evolution and change, then, can be considered vital to a program's usefulness over time.

Factors like advances and new inventions in technology may necessitate software updates and improvements. However, the need for a new or improved version of a product may also be intentionally implemented into its design. Such a phenomenon is commonly called planned obsolescence, a term popularized in the 1950s by industrial designer Brooks Stevens (The Economist, 2009). Stevens's approach to planned obsolescence was to increase sales by making consumers want a new product sooner than was necessary. One example of this is fashion, where something becomes "last season" and therefore worthy of replacing despite the fact that it has not outlived its usefulness. However, there are many examples of products being designed to last for a shorter duration than the maximum that would be possible.

Such a phenomenon has been documented at least as early as the 1920s, when the light bulb industry came together to generate more business by intentionally decreasing the average lifespan of a light bulb (Dannoritzer, 2010). Among more recent examples, also noted by Dannoritzer (ibid.), are things like the Apple iPod battery design, and printer "kill switches." Apple's iPod battery was considered faulty in design by so many consumers that it led to five class action lawsuits (see also BBC, 2004), and printers have been found to include a component, dubbed the kill switch, that makes it stop working after a certain amount of prints have been made, even if the printer itself would still be entirely functional.

In computing, related issues that might arise due to planned obsolescence are that newer versions of programs may not be compatible with earlier operating systems or versions, and that new versions of a program may introduce new file formats that cause compatibility issues with earlier versions of the program. Additionally, both end-of-life announcements and end-of-support deadlines for earlier versions need not be related to a decreased usefulness of the program in question, but rather with a desire to focus on a newer version; thereby urging users in need of support to upgrade from a product they were still satisfied with.

Open source software is rather unique regarding planned obsolescence in that all open source licenses guard against its implementation. Specifically, the right to fork makes planned obsolescence impracticable, at least on technical (software-related) grounds. As long as there is a large enough community interest in a project, it will persist. Even in the case of an abandoned project, a renewed community interest can once again spark its evolution and development (back into relevance). Indeed, among the almost 400 forks studied in paper 2 of this book, 7% were done in order to restart a project that had been abandoned.

The right to improve a program, the right to combine many programs, and the right to make a program compatible with other programs and versions, are all fundamental

rights built in to both the definition of open source, as well as the requirements for being able to classify a license as an open source license. And research has shown these rights are often exercised (Fitzgerald, 2006).

3.3.2. Community level

Fogel (2006) has called the right to fork the glue that binds project participants together, as no one has a magical hold over the project – anyone can fork it at any time. This is indeed commonly the case: developers try to avoid having members split into disparate groups working on separate forks, as such an event would lead to redundant efforts and less progress overall. However, in cases in which the leader of a project is seen by the members to not be acting in the best interests of the group as a whole, forking provides an opportunity for the developer community to ensure that the project can continue evolving in accordance with the majority's goals and wishes. This, also, is a kind of fork that developers consider entirely acceptable (as discussed further in section 6.2.2 and paper 4 of this book).

Tensions between project leadership and developer community are perhaps more easily identified in corporate projects, in which the issue of a project's ability to generate income – in addition to its development – need be included. A concern brought up by Lerner and Tirole (2002) and Ciffolilli (2004) is that of corporate "hijacking" of code: changing a project's source code from an open source to a closed source license. As was discussed in the sections on licensing, there are restrictions as to when this is possible, depending on the original license as well as ownership of the code. Purchasing ownership of a codebase gives the option to change the licensing of future releases. Otherwise, such concerns are primarily relevant for projects licensed under a permissive license. Here, forking is an option for retaining community control over a version of the codebase. Ownership of the unlicensed code will go to the purchasing company, but the community can fork any version licensed under an open source license, and are free to do whatever the license in question allows. However, in this case dual licensing is no longer an option, since that requires ownership of the original.

Moody (2009a) notes that open source companies and the open source community differ substantially in that companies can be bought and sold but the community cannot. He goes on to note that 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 take the fork in whichever direction they choose. Moody's comments were made in discussing one of the most interesting series of events in recent open source history: the changing ownership of the MySQL codebase and the subsequent MariaDB fork. Here, developer reactions to corporate actions lie at the heart of the fork. From a legal perspective, forking even a massive program with widespread adoption is entirely possible. However, it must be noted that forking a program is the easy part. It is what is required after forking that poses the real challenges; issues like building a community, user-base, a stable income, and establishing the availability of support and services for the program.

Should a competitive fork nonetheless occur, then even in the case of an actual split of the developer and user base, a fork can still potentially be beneficial. For instance, a fork could solve problems relating to a situation in which a programmer would be interested in working on a program, but reluctant to work with a specific person or team working on the same project.

3.3.3. Business ecosystem level

The right to fork is arguably the most significant of all factors behind shaping the open source business model landscape into what it is today. This is in part due to what companies *can* do because of the right to fork, but equally due to what they *cannot* do because of the right to fork. As Wheeler (2007*a*) notes:

> [C]opylefting licenses (such as the GPL and LGPL) permit forks, but greatly reduce any monetary incentive to create a fork. Thus, the project's software licensing approach impacts the likelihood of its forking.

A fork is bound by its license, and lucrative approaches like dual licensing are not available to those who work on a fork rather than the original, unlicensed version. By way of example, the MariaDB fork (discussed in paper 3) faces this very challenge. When MariaDB was forked from MySQL, MariaDB was licensed under (and thus bound by) the open source license MySQL was offered under – the GPL (version 2). This means that the MariaDB Foundation, charged with overseeing MariaDB and its development,[56] cannot use the same monetization strategy as Oracle, since the MariaDB Foundation is bound by the GPL, whereas Oracle (the current owner of MySQL[57]), is not. The most significant difference is thus that, whereas Oracle can offer dual licensing of MySQL, the MariaDB Foundation cannot.

Any distributed open source code can be modified and redistributed as a new program. However, upon distribution, the original company cannot withhold from others the right to also modify and redistribute the same code, regardless of how much it cost or how long it took to develop. Indeed, one key aspect to the effect of forking on the business ecosystem level hinges upon the question of distribution. Since open source license requirements are triggered upon distribution, if a company does not distribute a program, it can use open source code in the creation of its software, without having to share the source code to the new software in question. So, in the case of a program that is only used internally in a company, the right to fork provides a large body of code from which to select programs that can be modified to fit the company's needs. The downside to such internal modification and use is that the company will then not gain the potential benefit of having a large number of developers outside of the company aiding in the testing and debugging of the software. Furthermore, with each new update to the program or programs that the company based its software on, it may need to update its own code as well, for instance to avoid problems with compatibility with other software.

Cooperation is a common feature in the business of open source (see, e.g., Skerrett, 2011; Smith, Alshaikh, Bojan, Kak, and Manesh, 2014). Companies that try to monetize open source software directly, regardless of the business model, are the exception rather than the rule. Generally, open source is used in co-creating superior software needed by several firms, and that can be identical for them all. In other words, companies can all contribute to non-unique software that the companies in question need to support whatever their unique product or service offering is. For instance, several game developer studios could cooperate on programming a game engine. While all resulting games would use the same engine, the games themselves would be unique. A better engine benefits all companies, thus offering an incentive for cooperation. Open

---

[56] See https://mariadb.org/en/foundation/, accessed 10 February, 2014.

[57] MySQL was first acquired by Sun (see, e.g., MySQL, 2008), and Sun was then acquired by Oracle (see, e.g., Oracle, 2009)

source software is commonly developed to *support* what is unique about the business, rather than to *be* what is unique about the business.

An interesting example of an open source cooperation is Red Hat and CentOS. CentOS, an operating system first released in 2004, is a fork of Red Hat's operating system Red Hat Enterprise Linux (RHEL). CentOS offers a 100% binary compatible fork of RHEL, meaning, in effect, that they provide a free version of one of Red Hat's most central programs.[58] Red Hat CEO Jim Whitehurst has noted that they consider proprietary operating systems their competitors, not open source operating systems – even going as far as to say that Red Hat "embraces" CentOS (see Assay, 2013), since having an important open source operating system be based on a fork of RHEL is a means to further establish RHEL as the default Linux. In January of 2014, some ten years after the first CentOS fork was released, Red Hat announced that they would be joining forces with CentOS, contributing both resources and expertise (Red Hat, 2014). This is in addition to Red Hat's involvement with the Linux-based open source operating system Fedora. Red Hat merged with Fedora in 2003, splitting its distributions into the community-focused Fedora and the industry-oriented RHEL. Among other things, Fedora has since served as a proving ground for features that might be implemented in RHEL (Red Hat, 2013b).

In addition to its effect on the dynamics of cooperation, the right to fork also allows businesses to pick up where others left off. After Nokia abandoned the MeeGo project in 2011 (Nokia, 2011), Jolla announced that it would create a business around Meego, made possible by forking the original code (Jolla, 2012). The first Jolla phones with the new Meego-based operating system, Sailfish, were launched in November of 2013 (see, e.g., Tech News Central, 2013).

Franke and von Hippel (2003) have called open source programs "innovation toolkits", as they offer users the source code to improve upon. Similarly, the right to combine programs means that the plethora of online source code repositories, offering hundreds of thousands of programs, make it possible for creative minds to imagine new ways of combining existing programs, thus offering something new through the right to fork (see, e.g., Nyman, Mikkonen, Lindman, and Fougère, 2012).

3.4. The prevalence and complexity of forking: some (simplified) examples

In section 1.2, we briefly covered the operating system BSD as well as a few of its forks. In order to better convey the scope of the phenomenon, a more in-depth view of the history of BSD is shown in Figure 7. As the figure shows (see also section 3.1.3), the BSD source code itself was forked from the operating system Unix. Borrowing from the terminology of the fork system call, one could describe the multiple relationships between the programs covered as parent-child relationships, as well as siblings, i.e., versions that share a common antecedent. Indeed, Unix has spawned so many versions and variations that the term "Unix-like operating system" has been coined to cover these systems.

[58] A mitigating point to note regarding Red Hat's potential loss of income from support and services regarding RHEL is that CentOS users are primarily considered to be very familiar with Linux and thus not need the support and services offered by Red Hat (see, e.g., Vaughan-Nichols, 2009; Clark, 2014).

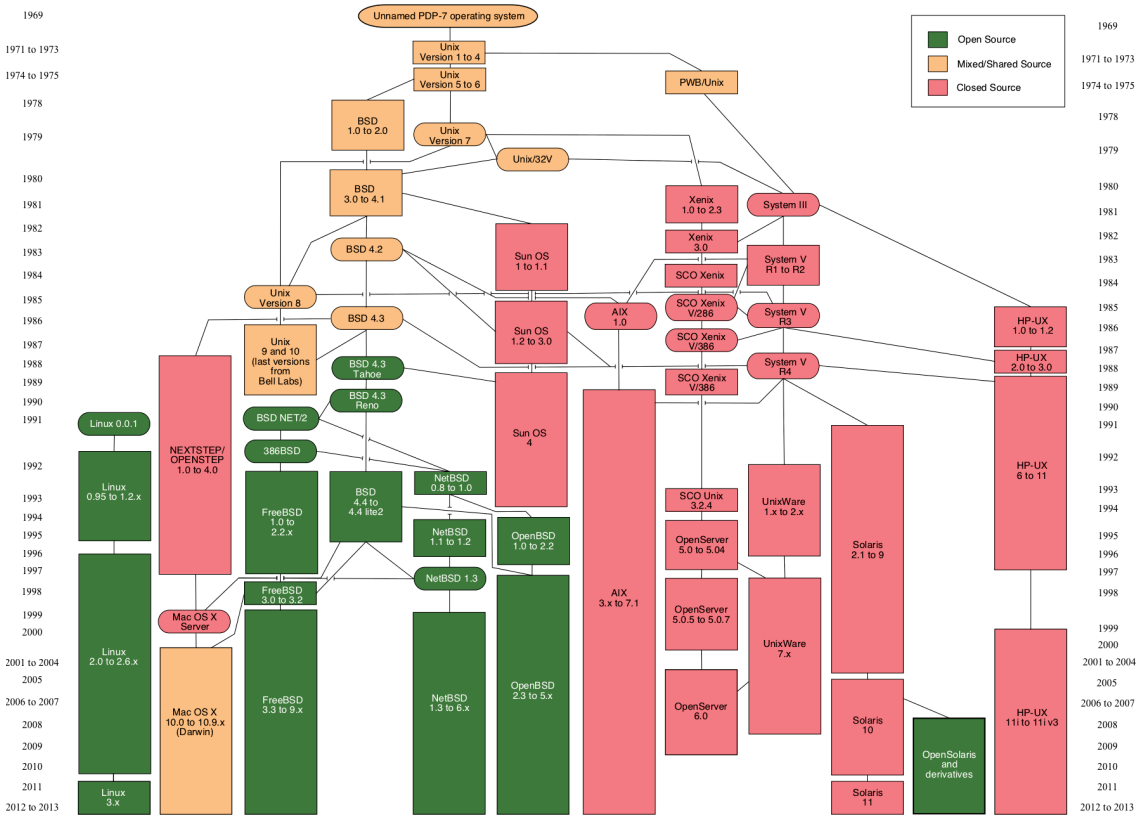


Figure 7 35 years of forking in the lives of Unix and Unix-like systems[59]

While the figure may seem complex, it is actually a simplified view of the flow of source code among Unix-like operating systems. Operating systems are collections of several programs, some of which are used by multiple operating systems. One example of such a program is the GCC compiler system (discussed briefly in section 1.2), which is used by several Unix-Like operating systems including both Linux as well as members of the BSD family. Thus, while the original source code for Linux is not descended from Unix, both Linux and others in the Unix-like family of operating systems still share code in the form of system (and application) software. These may have been forked from a common ancestry and then modified to fit the needs of each specific operating system, or may be included more or less unmodified. Furthermore, the attentive reader will remember that the Linux operating system itself consists of a collection of programs, many of which were originally developed as part of the GNU project. At the time that Linus Torvalds developed the Linux kernel, the GNU project was largely complete, save for a kernel.[60] Thus, Linux itself was enabled by the right to combine the Linux kernel with pre-existing code from the GNU project.

---

[59] The figure is adapted from user KommissärMatthäi's version of the original, available at: http://upload.wikimedia.org/wikipedia/commons/7/77/Unix_history-simple.svg, accessed 27 October, 2014. It is licensed under a Creative Commons Attribution-Share Alike 3.0 Unported license.

[60] A kernel is the central component of any operating system. One of its core tasks is handling the software's interfacing with the hardware of the computer. In addition to being enormously important, a kernel can also be enormous in size: Linux kernel version 3.2 contains over 15 million lines of code (Corbet, Kroah-Hartman, and McPherson, 2012).

The complexity of forks related to the Unix family of operating systems could be extended further still by including all the known implementations and variations of the programs included in the figure. However, involving such complexity would be beyond the scope of a simple figure, given that Linux alone has been forked into hundreds of different variations (called distributions, or distros).

3.5. Code forking defined, part 2: competitive and non-competitive forks

Having taken a closer look at a number of examples, we can now broaden our view of the term fork by introducing the concepts of competitive and non-competitive forks. Forks in which there are no practical changes to the program itself are quite rare. Forks are pragmatically motivated, started as a means of achieving a specific goal. This goal is rarely to compete with the original. However, forks with such a goal are significant in both their motivations and their potential effect. Thus, based on the data gathered for this dissertation, it seems useful to categorize forks into two overarching groups of outcomes: competitive forks and non-competitive forks.

Competitive forks are forks whose primary goal is to serve needs identical to those of the original program, often by producing a near-identical product to that from which it was forked. There will be some difference to the original, but it may be unrelated to the actual source code. For instance, it may instead deal with the licensing of the product or the ability of the community to contribute. Non-competitive forks are born out of a desire for something sufficiently different to the original that it necessitates a new project. A competitive fork will by default compete with the original for its users and developers; a non-competitive fork may or may not do so.

This categorization is a simplistic one, and few forks will fit seamlessly into either category. Even a fork that is born out of seemingly non-competitive motivations, e.g. to alter the direction of the development significantly, may still affect the original as there may be those among both users and developers who switch to the fork. However, it can be seen as a continuum, as depicted in Figure 8. This categorization, placing a program somewhere along the continuum, helps inform of the extent to which its creation may (purposefully or not) affect the project from which it was forked, as well as speak to the primary motivation of the fork.

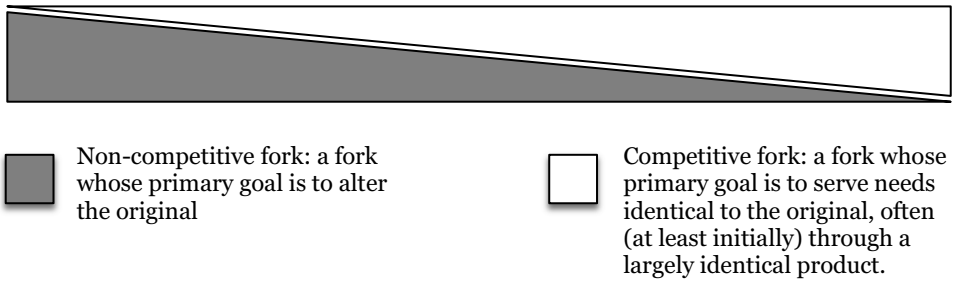


Figure 8 The continuum of competitive versus non-competitive forks

Non-competitive forks are forks whose primary goal is to alter the original, e.g. through implementing different features, changing the focus of development efforts, reviving an

abandoned fork, etc. While some are arguably entirely non-competitive, as would likely be the case when reviving an abandoned project, most will have some element of competitiveness about them. However, this does not necessarily mean that there need be any animosity between the projects. It simply means that, though different in some respects, enough similarities still remain that the fork could sufficiently meet the needs of users of the original project.

Competitive forks are forks whose primary goal is entirely or virtually identical to that of the original. By default, it will compete for both users and developers, seeking to replace the original. While initially producing either an identical or near-identical product, over time the competitive fork may try to gain a competitive advantage through features or solutions that differ from the original. Still, significant similarities between the two are likely to remain. For instance, meeting the technical requirements for the competitive fork to remain a drop-in replacement for the original project. Should the fork achieve a sufficiently strong or even dominant position, such requirements may be given less weight. Competitive forks are the kind of fork that is likely to produce the most redundant effort, something that is viewed unfavourably. Yet at the same time, they address other issues developers feel very strongly about.

3.6. Summary

The term fork was introduced into computing by Melvin Conway in a 1963 conference paper. The fork system call was implemented in Project Genie a few years later, by L Peter Deutsch. After Project Genie, Deutsch moved to work on Unix, and subsequently implemented the fork system call in Unix. With the proliferation of both Unix and Unix-like systems, the fork system call became a household concept in operating systems.

Pinpointing exactly when forking entered the open source vocabulary is difficult, or perhaps even impossible. What is known is that it gradually adapted its meaning, from existing only as a verb, to "fork off" a branch, to eventually, at least by 1996, existing as a noun as well.

Forking affects open source software development, governance, and the business of open source software in many significant ways. Among them are that it guarantees that a project can be kept alive as long as there is a community interested in its development, that project leaders or corporate sponsors cannot act entirely outside of the scope of developers' interests and goals, and that developers and companies alike are offered a plethora of existing programs based on which they can offer new combinations or modifications.

Examining the histories of programs, as is the case with Unix-like operating systems, one may find complex relationships of multiple forks, each taking the original project in slightly different directions.

In discussing forks, one means of concisely conveying significant meaning about a fork is to use the terms competitive or non-competitive fork. Competitive forks seek to replace the original from which they stem, while non-competitive forks seek to take a pre-existing program in a new direction. In practice, the distinction is not necessarily an easy one to make as far as non-competitive forks go, as forks may be of a competitive nature, not by intention, but simply because they are similar enough to the original that competition is inescapable.

4 RESEARCH OBJECTIVES, METHODS, AND DATA

This chapter covers topics related to the writing of the included articles. It begins by detailing the mental journey required when tackling a new topic, followed by the research questions that evolved from this newfound knowledge. The chapter continues by detailing the methods used to seek answers to these questions, the data gathered, and its analysis.

4.1. From ignorance to confusion to insight

I began work on this book by gathering what information I could find about code forking. I found only a few textbooks that made even some brief mention of it. I then came across SourceForge (http://sourceforge.net/), a source code repository. A source code repository can be thought of as an online library of programs that developers can upload their programs to, and interested users can download programs from. SourceForge allows users to search their database using search words. My first set of data was a collection of all the programs on SourceForge that stated that they were forks of another program. I gathered all the data available about the hundreds of such programs: what programming language they used, which licenses, what kind of program it was, how the developers described it, what year it was entered into SourceForge, when it was last updated, and any other potentially relevant information they provided. Then I spent weeks coding the data, making tables and pie charts, analysing and comparing, imploring the massive document and spreadsheet files to tell me something – anything – interesting.

The data eventually resulted in two articles. The first was a study in what motivations developers stated lay behind their decision to fork. The second was an analysis of license use over time, comparing license use among forked programs with license use among open source programs in general. However, while the paper received a 'revise and resubmit', I chose to abandon it due to an uncomfortable insight that resulted from the data: of the hundreds of forks I had gathered information on, there were precious few which seemed to fit snugly within the definition of a fork as stated in the literature. This raised the question of what programmers mean when they call something a fork. The significance of this question – and, in particular, the insight that programmers may be speaking about different things when talking about forking – led me to pursue this avenue of research. However, it was far from obvious what would be the best way to address, or study, the issue.

While considering approaches to the definition issue, I began putting together a questionnaire for a pilot study on how programmers defined forking. Concurrently, I focused on topics I felt were both relevant and unproblematic. Firstly, I co-authored some conceptual papers examining the effects that the right to fork has on open source software in general. While these papers did not perhaps offer much in the way of new insights into forking, they were at least among the first texts I had seen that attempted to highlight that the right to fork also had a positive side, and indeed that forking was one of the integral building blocks of the open source phenomenon.[61] I then turned to a case study, specifically that of the MariaDB fork from MySQL. While I felt there was an

[61] Moen (1999) (among others) deserves credited for doing this long before I did, albeit through the less formal channels of an office mailing list.

ambiguity about the term fork, this was a case that I felt was unequivocally definable as a fork. I was fortunate enough that Michael "Monty" Widenius, founder of both MySQL and the fork MariaDB, was generous with his time, agreeing to a number of interviews in order to make the writing of such a case study possible. Given how rare that kind of fork is, being able to interview the very person behind the decision to fork was a unique and valuable opportunity.

I had now written a few papers that addressed the question of why programmers forked code. However, I was also interested in the question of how they viewed forking as a concept. With accounts of forking being both praised and condemned, it seemed fertile ground for further examining how hackers themselves viewed this right. I conducted a series of interviews aimed at addressing both the question of how programmers defined a fork, as well as how they viewed the right to fork – topics like: what are the positive and negative sides to the right to fork, what are the "unwritten" rules of forking, does one have any moral obligations when forking, and if one does decide to fork, how should one go about it?

4.2. The research questions and articles that address them

The overall aim of this dissertation is *to provide a greater understanding of the practice of code forking in open source software, with particular emphasis placed on understanding developer views and motivations*. Figure 9 presents the specific research questions of the dissertation, coupled with the papers that address them. While most articles are relevant to some degree to all questions, such a grouping would be unhelpfully broad. Therefore, a distinction is made between primary and secondary relevance, with what could be considered minor relevance omitted from the figure. Black arrows connect the research questions to the papers that primarily address them, while grey arrows indicate the research question as a secondary area of focus.

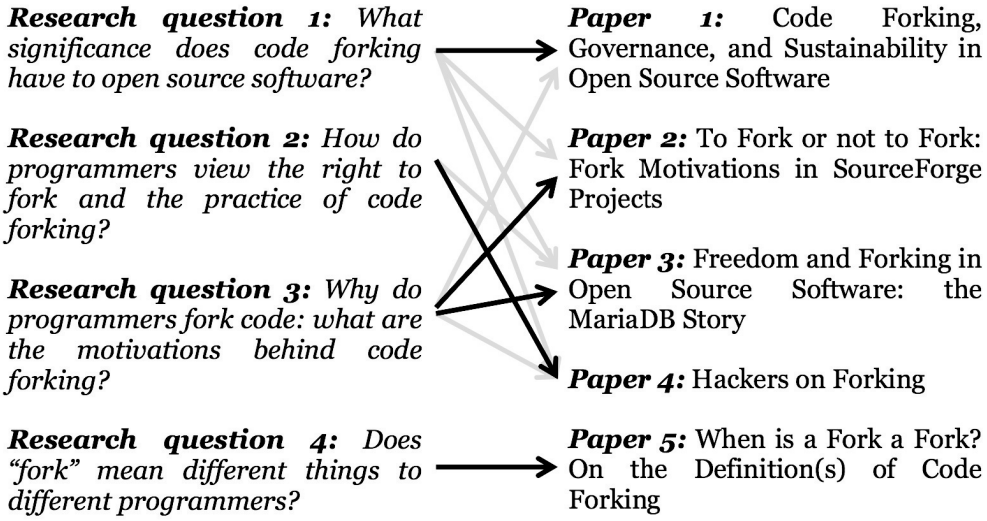


Figure 9 An overview of the research questions and the articles that address them

The time spent in study that did not yield articles – the cost, if you will, of an exploratory approach – was not entirely wasted: I have included in the kappa insights that have not been included in articles.[62] Furthermore, I have co-authored and published additional articles that benefited from the research done for this book, but that are not sufficiently relevant to the topic as to be included.

4.3. Research methods, data collection, and analysis

This dissertation contains one conceptual paper (paper 1) and four empirical papers (papers 2 through 5). The empirical papers are based on three separate sets of data. As will become clear from the summaries of the papers, they contain a mix of different methods. The reasoning behind this was that the method for each paper was chosen based on what approach would best answer the research questions of that paper.

4.3.1. Paper 1

Paper 1 is a conceptual paper analysing and describing effects that the right to fork has on open source software. Being a conceptual paper, it relied on an in-depth literature review rather than a particular research method. The paper sought to highlight benefits inherent in the right to fork, and was based on a review of the literature, including academic studies as well as other sources (news sources, press releases, blogs, etc.), covering current and past events related to open source software, its development, and the business that surrounds it. The paper also benefited from practical examples as well as theoretical supposition based on open source licensing. The paper builds on previous work by Nyman, Mikkonen, Lindman, and Fougère (2011 & 2102).

4.3.2. Paper 2

Paper 2 analyses developer-stated motivations behind forking. It utilizes elements of both quantitative and qualitative approaches.

Data collection

My exploratory approach led me to gather what data I could find related to code forking. As noted in section 4.1, my data came from SourceForge. At the time of the study, SourceForge stated that it contained over 260,000 open source projects, created by over 2.7 million developers, with more than 2,000,000 downloads daily.[63] In addition to storing the source code for projects, SourceForge also allows users to add information related to the projects. Among the additional information developers were able to include was a brief description of the project.

I compiled a list of all of the programs on SourceForge that contained the word “fork”. I also did searches on dozens of misspelled variations of the word, none of which produced any hits. This analysis covered all projects registered on SourceForge from its founding in late 1999 through 31 December, 2010, thus covering a timespan of slightly over 11 years. I then analysed each of the descriptions individually to eliminate false

[62] Chief among these being the story of how the term “fork” came to computing, as well as insights into the business of open source – something greatly affected, if not entirely defined, by the right to fork.
[63] Source: http://sourceforge.net/about, accessed March 9, 2011.

positives, i.e. projects that included the word fork for other reasons, e.g. a program that noted that it "can be used to avoid common security problems when a process forks or is forked." Projects that were impossible to categorize based on the data given (e.g. due to a complete lack of user-stated information) were similarly discarded. This process yielded a total of 566 programs that developers reported to be forks. Then the developer-stated motivations were analysed again, this time with the goal of identifying those forks for which it was possible to identify the motivation behind it. Forks with insufficient or unclear information regarding their underlying motivation were discarded. This process yielded a total of 381 forks for which the motivations behind them could be identified.

Data analysis and coding

The coding process, conducted with my co-author, was done in three phases. First we went through all of the descriptions, summarizing their motivations into as brief a description as possible while still conveying its original meaning. Additional information available on SourceForge, e.g., the project name and its stated features, could also be used to aid and confirm the analysis. We then analysed all of the condensed descriptions to identify common themes, or subcategories, of motivations. In cases where a fork included more than one motivation, we identified what appeared to be the most central of motivations behind the fork. Finally, we examined each subcategory to identify overarching groups of themes. Table 3 offers some examples of the phases of the coding process.

Table 3 Some examples of the coding process

Project description	**Phase 1: highlight relevant sections**	**Phase 2: create sub-categories**	**Phase 3: create main categories**
[Project name] is a fork of [original project name]. [The] purpose of [project name] is to add many new features like globule reproduction, text to speech, and much more.	Add features: globule reproduction, text to speech, and more	Add content	Content modifications
[Name] is just a fork of [...] with better UI and much suitable for catering services, online order management in hospitality domain. Suitable for restaurants, caterers, tent houses etc. Plan to integrate HR , CRM and POS.	Better UI, particularly suitable for restaurant/ catering industry	Focus content	
The continuation (aka fork) of [original]. [Project name] allows anonymous, distributed, encrypted messaging and file transfers. This project aims to improve the overall usability and security of the [original].	Improve overall usability and security	Technical: improvement	Technical modifications
[Project name] is a fork of [name of original] library, that is should be redistributable. [The fork] depends on OpenCFLite and Bonjour. It is intended to port Mac Applications to Windows and other Unix systems.	Port Mac Applications to Windows and other Unix systems.	Technical: porting	

4.3.3. Paper 3

The primary goal of paper 3 was to discover and describe how and why the MariaDB fork of MySQL came about. A secondary goal was to understand why it did not come about sooner given that the right to fork is a constant in open source software. When attempting to illuminate the motivations behind a particular decision, as well as in seeking answers to how and why a decision or an event came about, conducting a case study is a particularly suitable approach (Yin, 2003). There were many unique elements to the MariaDB fork, among those were: that it was a fork of a very common open source program (MySQL), that it was done by the original developer of its predecessor (Widenius), and that it sought to compete with the original as a drop-in replacement. Extreme or unique cases are suitable for single-case case studies (Yin, 2003 & 2009). Archival research was used to both inform the interview questions as well as for triangulation (Yin, 2009).

Data collection

The data consisted of both interviews with MySQL and MariaDB founder Michael Widenius, as well as archival data consisting of:

- The entirety of Widenius's blog
- Press releases and other relevant information available on the MariaDB, MySQL, MariaDB Foundation, Monty Program, and Oracle websites
- Industry articles about MariaDB, including several interviews with Widenius, reported in the following online journals: ArsTechnica, Computer World, Forbes, H-Online, Info World, ITWire, Linux Journal, Techradar, and ZD Net
- Additional relevant articles uncovered by searching the community website Slashdot (Slashdot.org) using the search terms: MariaDB, Widenius, Sun, and Oracle

The majority of the archival data was gathered and examined prior to the interviews. I conducted two face-to-face interviews with Widenius, averaging 45 minutes in length each. The interviews were recorded and transcribed. The first interview was used to clarify the motivations for the fork and the overall picture of events derived from analysing the archival data. The second interview was used both for clarifying follow-up questions, as well as to ensure that the representation and interpretation of Widenius's motivations behind the fork were accurate and complete. Email and phone conversations were used for additional follow-up questions and points needing further clarification. A third interview, conducted some months later for a different project, was also used to gain some additional details and insight about the MariaDB case.

Analysis

In order to gain as broad an understanding as possible about the decision to fork, I created a timeline of significant events leading up to the fork. This timeline covered one decade, from when MySQL took on external funding until the time of writing (early 2013). The timeline included categories for Sun, Oracle, Widenius, MariaDB, MySQL/MariaDB Developers/workers, MariaDB adoption by third parties who had previously used MySQL, and significant actions related to investors and investments in the MySQL codebase. The timeline included in the final paper is sparser, both in content and length. The more extensive timeline put together for the analysis was in order to help ensure the accuracy of the findings related to the fork and its motivations.

The timeline also helped answer the paper's secondary research question of why the fork didn't happen sooner.

Upon completion of the paper, it was sent to Widenius to ensure that the paper accurately portrayed his motivations and view of the events included, as well as to allow him to comment. Widenius noted that the motivations and views stated were accurate, suggesting only minor changes. These were predominantly regarding adding additional clarity to the timeline presented in Table 1 of the article.

I did not interview additional sources for this paper, a decision backed up by the goal of the paper itself: to answer the question of why Widenius decided to fork. Thus it was Widenius's own thoughts and motivations that were key to answering the research question. However, I later interviewed other members of MariaDB for a related paper. During those interviews I found nothing to dispute the findings of paper 3.

4.3.4. Papers 4 and 5

The goals of papers four and five were to document programmers' views, thoughts, and opinions about the practice of forking, as well as their interpretations of the term fork. While the goals of the papers were different, they both relied on documenting programmers' views. Thus, I conducted interviews that included questions for both articles.

Data gathering

I first familiarized myself with the writings and opinions about forking found in academic literature, practitioner texts, and other sources. Based on the knowledge gained from the literature review, as well as work done on my previous articles, I put together a semi-structured interview guide that included questions targeting the research questions of both papers 4 and 5. I then conducted a pilot study of face-to-face interviews with four programmers. After each interview I modified the interview guide, if needed. The main changes done to the interview guide were to add additional clarifying questions for paper 5 regarding specific interpretations of what a fork is and under which circumstances the term is appropriate.

After the pilot study, I conducted nine additional interviews. The interview guide used is included in Appendix 2. All interviews, including the pilot study, were recorded and transcribed to aid in the data analysis. The interview subjects were chosen through snowballing, initially of people I was familiar with, and then through interviewees recommending others who might be interested, or contacting their programmer acquaintances to suggest they get in touch with me if they were interested. The criteria for inclusion as an interviewee were that: i) they needed to have been involved in at least one open source software project; and ii) they needed to currently either make a living from programming, be studying programming, or to have at some point previously had made their living from programming. The programmer with the least experience was involved in their first open source project at the time of the interview. The most experienced programmers had several decades of experience, and included a developer who had worked primarily on the Linux kernel, a developer employed by the Apache Foundation, a developer who was a founding member of both MySQL and MariaDB, and a Linux pioneer who has both authored as well as contributed to numerous other open source programs.

The interviews were conducted in Finnish, Swedish, or English, depending on the preference of the interviewee. The interviews were conducted either face-to-face or using Skype, with a fairly even split between the two. As with the pilot study, the interviews were semi-structured, allowing for additional questions as needed, as well as giving the respondents the opportunity to express all thoughts and insights they felt relevant. A majority of the programmers were from Finland, the rest were from Italy, Russia, and the Ukraine. The interviewees ranged in age from their early twenties to their early fifties. One interviewee was female, the rest were male.

Data analysis

The goal of *paper 4* was primarily to identify unique viewpoints held by the interviewees. Coding was a rather straightforward matter of, firstly, identifying the individual notions stated regarding each individual research question; and, secondly, comparing the answers to see which answers were similar enough in nature that they should logically be grouped together, particularly in cases where there were a significant amount of individual points offered. The questions targeted primarily at paper 5 (question three and its sub-questions) were used to identify developers' views regarding what constitutes a fork. This was of particular significance given the broad range of actions that may (or may not) be included in that term. One developer's view regarding what constitutes a fork was so unique that his answers were excluded from the dataset of paper 4. Thus, the final data consisted of twelve interviews. Upon completion of the paper, it was sent to a subset of interviewees for commenting.

The goal of *paper 5* was to address the seeming inconsistencies regarding how the term fork can be used among developers. The goal was to identify what developers considered to be a fork, including the key defining characteristics or criteria for inclusion in that category. A further goal was to contrast and compare developer answers, both amongst each other as well as with previous literature, to judge the level of similarity among definitions. As the goal was to document views and examine points of parity among them, the analysis was quite straightforward, and a question of comparing the transcribed answers relevant to each question individually.

5 SUMMARIES OF THE PAPERS

This section offers a brief overview of the papers included in this book. The full papers can be found in Appendix 3.

5.1. Paper 1: Code Forking, Governance, and Sustainability in Open Source Software

The term "code forking" has traditionally had negative connotations, bringing to mind issues like redundant efforts and interpersonal strife. This paper examines code forking from the viewpoint of the positive affects the practice of code forking can have, primarily when viewed through the aspects of sustainability and governance. The paper thus builds, in part, on previous work by Nyman, Lindman, Mikkonen, and Fougère (2011 & 2012).

The concept of sustainability and what exactly it means and encompasses is under debate (e.g. Connelly, 2007; Davidson, 2001). For the purposes of this paper, sustainability is defined as the possibility of an open source program to continue to serve the needs of its developers and users.

Software, if left unmodified, is bound to become less satisfactory to its users over time (Lehman, 1980). Additionally, truly successful software is adapted and modified to the extent that it outlives the hardware for which it was originally written (Brooks, 1975). Thus, a key element in software sustainability is the possibility to constantly modify, change, and adopt the software to its users' current needs. The right to fork ensures that a piece of open source software cannot be hindered from evolving to meet such needs: even in the case of planned obsolescence, where products are designed (either through premature failure, technical restrictions, or any other reasons) to be out-dated sooner than necessary, forking allows developers to ensure that an up-to-date, relevant version can be maintained and developed further.

Moving from software as our level of analysis to the level of the developer community, code forking, and even the mere existence of the right to fork, has significant impact. Fogel (2006) notes that the right to fork helps bind developers together, so as to avoid the loss in productivity that could ensue in cases where developers split to work on an additional fork. Furthermore, concerns regarding corporations "hijacking" the code (see, e.g., Lerner and Tirole, 2002) by making the main development branch proprietary can be swayed by the right to fork. However, this is not meant to imply that maintaining a fork is an easy task. During the past decade, there have been some notable forks that have stemmed from community dissatisfaction with governance, or concern with the future openness of the code, or both, e.g., the LibreOffice fork from OpenOffice, and the MariaDB fork from MySQL.

Even with the right to fork, open source software is increasingly popular among companies, and is seeing more and more companies foot the bill for its development (e.g. Wheeler, 2007a & 2009). Forking allows entrepreneurs the right to take over an abandoned project, as was the case with Jolla and Meego. Furthermore, companies can also fork existing programs based upon which they can develop unique programs for in-house use. Finally, the plethora of open source software available on source code repositories can serve as a source of ideas and building blocks for creating unique

combinations in order to meet customer demands, or offer them something entirely new.

The paper examines forking from a positive light, with a focus on sustainability and governance. Its two central claims are: i) that forking represents the single greatest tool for guaranteeing sustainability in open source software development, and ii) that even the mere existence of the right to fork has a profound effect on the governance of open source software programs. The right to fork makes planned obsolescence, versioning, vendor lock-in, end-of-support issues, and similar initiatives all but impossible to implement. Forking grants the community the power to safeguard against unfavourable actions by corporations or project leaders. Furthermore, forking can serve as a catalyst for innovation while simultaneously promoting better quality software through natural selection. Thus, forking helps keep open source initiatives relevant and presents opportunities for the development and commercialization of current and abandoned programs.

Table 4 Key facts for article 1

Full reference	Nyman and Lindman (2013) Code Forking, Governance, and Sustainability in Open Source Software. *Technology Innovation Management Review*, 3(1): 7-12.
Research questions addressed	(1) What significance does forking have to open source software? (3) What are the motivations behind code forking?
Aim	The aim of the article is to examine the implications of the right to fork, with a focus on how forking affects issues related to governance and sustainability.
Data and research method	Conceptual
Proposed contribution	A broader view of forking, with a greater focus on what could be considered the positive aspects of the implications of the right to fork. Among these are forking as a tool for guaranteeing sustainability, as well as forking as a central element affecting the governance of open source software development.

5.2. Paper 2: To Fork or Not to Fork: Fork Motivations in SourceForge Projects

Despite the significance of the right to fork, as well as high visibility forks such as LibreOffice and MariaDB, the topic of forking has seen little study. The data for this article was gathered from source code repository SourceForge, which at the time of the study noted to have over 260,000 open source projects, created by over 2.7 million developers.[64] The repository was started in 1999, and thus contained forks from a time-

[64] Source: http://sourceforge.net/about, accessed March 9, 2011.

span of slightly over 11 years. Developers are given the option of describing their program in connection with its project page on the repository. The SourceForge database can further be searched using search words. The database was searched using the term “fork” (as well as numerous intentionally misspelled variations), thus identifying all programs that the developer had stated were forks. This search yielded a total of 566 programs. After eliminating false positives, as well as programs for which the information supplied by the developer was insufficient to be able to categorize the motivations behind it, 381 forks remained.

The motivations given were by and large of a very practical and pragmatic nature. By far, the most common motivation was content-related, which was the motivation in roughly half of all forks. This group consisted of two primary motivations, both nearly equal in size: adding content, and focusing content. Adding content is fairly self-explanatory, and consists of additions of many different kinds, including better documentation, helper utilities, and larger maps to a game. Focusing content relates to the modifying of a program to better suit the needs of a specific user group, and could be either content-related or technically motivated. The data included forks modified to suit the needs of a diverse mix of groups, including dance studios, catering companies, and astronomers. By way of example, forks with a technical focus included projects forked to target various screen resolutions.

Roughly a quarter of all forks were motivated by technical issues. Most technical issues were related to either porting the original, or improving it without noticeably changing its focus or goals. Among the forks motivated by porting were several that ported to a different operating system. Among those motivated by improving the original were motivations like bugfixes, optimization, and security improvements.

There were four smaller groups, each accounting for between 4-7% of all forks. Reviving an abandoned project was the motivation behind 7% of all forks. 6% of the forks were motivated by license-related issues. Among the motivations were removing proprietary boot code, and licensing a program fork under the GPL. In 6% of all forks, the motivation was to adapt it to needs specific to a country or language. This group would fit well as a sub-group of the focus content group, but keeping it separate adds descriptive detail to the results. Experimental forks, accounting for 4% of all forks, were motivated by the testing of novel ideas or features without needing to alter the original. Interestingly, of all the forks studied, only four stated motivations having to do with open disagreement or discontent with the original developers: three forks that were started because the original would not incorporate their changes, and one which was started in reaction to a breach of trust.

The paper adds to our knowledge of forking by putting forth a taxonomy of motivations behind code forking. Furthermore, it shows that the vast majority of forks (as defined by the developers whose forks were studied) are of both a pragmatic and non-competitive nature. The paper furthermore notes the perceived inconsistencies between the forks and motivations described in the data, and existing definitions of a fork.

In addition to its publication in the OSS 2011 conference proceedings, it also won a “Best Paper” award at the conference, and was invited for publication in a special issue of the International Journal of Open Source Software and Processes.

Table 5 Key facts for article 2

Full reference	Nyman and Mikkonen (2011) To Fork or Not to Fork: Fork Motivations in SourceForge Projects. Published in: 1. *Proceedings of OSS 2011: Towards Sustainable Open Source* (ISBN 978-952-15-2718-0) 2. *The International Journal of Open Source Software and Processes*. Vol. 3, Issue 3. DOI: 10.4018/jossp.2011070101
Research questions addressed	(1) What significance does forking have to open source software? (3) What are the motivations behind code forking?
Aim	The aim of the article is to determine and categorize the motivations developers state for forking programs.
Data and research method	381 self-proclaimed forks on the source code repository SourceForge (sourceforge.net).
Proposed contribution	The vast majority of forks are benign rather than competitive, for the most part motivated by a desire to modify a program to better suit a user's, or group of users', specific needs.

5.3. Paper 3: Freedom and Forking in Open Source Software: the MariaDB Story

This article seeks to answer a question that is at the same time simple and significant: why was the MariaDB fork started? While significant factors that affect the open source community's interest to participate in a development project have been studied, there has been little focus on the motivating factors that can cause a contributor to become a competitor by utilizing the right to fork. The paper details the events that led Widenius, the founder of the MySQL project, to decide to fork MariaDB from MySQL.

MariaDB began as an experimental fork, with the purpose of developing a replacement engine for InnoDB, which had been purchased by MySQL's then-competitor Oracle. While there had been tensions about different issues during MySQL's development, none of them had resulted in a fork. One of the chief sources of tension was the choice of business model, and specifically, whether to close off some part of the MySQL codebase to aid in generating income. Widenius was strongly opposed to this, which played a role in such a change not being implemented during MySQL and its subsequent ownership by Sun.

After the announcement that Oracle would acquire Sun, Widenius was of the opinion that the two most likely scenarios regarding the MySQL codebase were that Oracle would either try to "kill" it (by no longer improving the code, thereby making it only a matter of time before it would become out-dated), or change the licensing model so that part of the code would be closed source. Since Widenius was not able to secure any guarantees about the future of the code, and both of the scenarios that he considered likely were unacceptable to him, he decided to fork the MySQL project.

The article further validates the existing notion that there is a strong threshold to starting a competing fork. Furthermore, it offers an in-depth analysis of the events and motivations behind the birth of a fork. Finally, it introduces the freedom factor hypothesis: limiting either the freedom for developers to contribute to a project, or the freedom inherent in a project's license increases the likelihood of a fork.

Table 6 Key facts for article 3

Full reference	Nyman (2013) Freedom and forking in open source software: the MariaDB story. *Proceedings of the 22nd NFF/Nordic Academy of Management conference*, Reykjavik, 21-23 August.
Research questions addressed	(1) What significance does forking have to open source software? (2) How do programmers view the right to fork and the practice of code forking? (3) What are the motivations behind code forking?
Aim	An analysis of the motivations behind a competitive fork; an examination of the commonly held belief that competitive forks are sought to be avoided.
Data and research method	Interviews with Michael Widenius, secondary data sources (Widenius' blog, news items, developer/IT blogs).
Proposed contribution	An in-depth analysis of a significant fork, motivated by a desire to ensure the future openness and relevance of the code. Further confirmation of the commonly held notion that developers go to great lengths to avoid (competitive) forking.

5.4. Paper 4: Hackers on Forking

The aim of this article is to explore programmers' views on forking. As a topic, forking raises strong and often conflicting views, yet there are no studies that focus on how hackers themselves view forking. The goal of this article is to address two broad areas of interest: how programmers view the concept of forking and how programmers view the practice of forking. This is divided into three topics:

1. What do programmers consider to be the benefits and drawbacks of the right to fork?

2. What constitutes an acceptable and unacceptable fork?

3. How should one go about forking? I.e. what, if any, are the moral obligations and proper codes of conduct in regards to forking?

The hackers interviewed were all of the opinion that forking is a central right, or freedom, of open source software. Regardless of any details or specifics regarding its

benefits or drawbacks, it is seen as a cornerstone of open source software and its development.

The benefits discussed can broadly be grouped into three categories: *preservation*, *experimentation*, and *customization*. Preservation is the possibility to continue an abandoned project through forking the code. Experimentation is the possibility to try new things, either out of personal interest or with the goal of integrating the changes back into the original program. Customization is the possibility to customize software to better fit one's needs. This category also includes situations in which, due to disagreements or other reasons, the developers choose to go their separate ways, each continuing development of their own version of the program.

The drawbacks noted were: that forking can result in confusion among users; duplicated work among developers; compatibility concerns, which can result in extra work for developers; that it can be used as a negotiation tactic; and that it makes monetization challenging.

The vast majority (all but one) of the developers were of the opinion that a fork was always acceptable when legal. One developer noted that a fork was acceptable if the original developers weren't doing their job, offering as an example a case in which the developers had either neglected or ceased development of a program to focus on a commercial version of it. While acceptable, developers still offered several examples of forks they would not support. Among these were: malicious forks, forks that were in breach of an agreement, and forks that were due to personal differences rather than differences of opinion about how to proceed with the program. Malicious forks were brought up by several developers, and covers forks that are created not to improve or modify the original program, but rather to cause confusion and disruption to the original. However, this type of fork was discussed mainly as a theoretical problem, with few being able to give an example of such a fork having actually taken place. One developer, with a long history of working on a Linux distribution, noted that he had seen a few instances of what he referred to as sabotage attempts, where a lone developer forked the entire program and attempted to create disagreements and disruption among the original developers. However, he also noted that these attempts were short-lived. A further example offered of a malicious fork, one that could perhaps best be described as a "FUD fork" (Fear, Uncertainty, and Doubt), would be a situation in which a commercial company released a fork of an open source program in order to discredit it, with the intention of gaining a stronger market position for their own competing, proprietary version of a similar software.

The majority of programmers noted the importance of crediting previous authors as well as the imperative to follow the dictates of the license. One developer, who stated that he preferred the term free software to the term open source software, also stated the imperative, when forking, to keep the source code open. This same developer was also the only developer to offer viewpoints on what the proper code of conduct is when forking, including points like clearly stating what the purpose is behind the fork, and conducting it in a way that causes as little harm and disruption to the original as possible. While other developers didn't offer lists of actions regarding a proper code of conduct, many did note the importance of communication with the developers of the original before forking, to ensure that a fork is actually necessary.

This study contributes to existing literature by both confirming many previous notions about how forking is viewed, as well as by adding more depth to our understanding of how hackers view forking. While case studies into individual forks shed light on how a

particular fork is viewed, a better understanding of how programmers view forking as a concept can grant insight on a higher, or more general level. This, in turn, can allow for more informed speculation regarding how a particular fork will be viewed by programmers.

Table 7 Key facts for article 4

Full reference	Nyman (2014) Hackers on Forking. *Proceedings of the 10th International Symposium on Open Collaboration* (OpenSym 2014). ACM, 2014. ISBN: 978-1-4503-3016-9. Doi: 10.1145/2641580.2641590.
Research questions addressed	(1) What significance does forking have to open source software? (2) How do programmers view the right to fork and the practice of code forking? (3) What are the motivations behind code forking?
Aim	The aim of the article is to document how programmers view forking, with a focus on what they consider to be good and bad about the right to fork, what kinds of forks they consider acceptable vs. unacceptable, and what the moral requirements and correct code of conduct are when forking.
Data and research method	12 semi-structured interviews with open source programmers.
Proposed contribution	The paper corroborates statements from practitioner literature regarding how forking is viewed among programmers, and offers new insights. It offers an understanding of how forking is viewed overall, which can inform how specific forks may be viewed.

5.5. Paper 5: When is a Fork a Fork? On the Definition(s) of Code Forking

The aim of this article is to explore programmers' views on what "fork" means. Specifically, what kinds of criteria they have for something to be called a fork. The data consists of semi-structured interviews with 13 programmers who are, or have been, involved with open source projects. The paper explores definitions of a fork offered in practitioner and academic literature, and presents how the developers interviewed define a fork, and what they view as the key characteristics of a fork.

The paper finds that the definition of the term fork has been expanding from early interpretations that view a fork as a split in a project, to more recent interpretations that also include actions that can be a part of a normal workflow, like creating a branch of a project to test new features. Adoption of this broader interpretation is likely to have been influenced by the interpretation of fork used by GitHub, though the broadening of use of the term started before GitHub's release.

The article highlights that, while there is much common ground regarding definitions, there can also be significant differences regarding what programmers consider to be a fork. This insight is intended to inform members of academia interested in the study of code forking about the existence of such differences, and thus the importance of designing one's research in a way that takes these differences into account. The article furthermore suggests Robles and González-Barahona's (2012) definition as a starting point for a definition that academia could unite around, and suggests the developing of a set of prefixes, primarily motivation-based, to further help subcategorize forks.

Table 8 Key facts for article 5

Full reference	Nyman (2014) When is a Fork a Fork? On the Definition(s) of Code Forking. Unpublished essay.
Research question addressed	(4) Does "fork" mean different things to different programmers?
Aim	The aim of the article is to document what programmers consider to be a fork and identify key characteristics of their definitions.
Data and research method	13 semi-structured interviews with open source programmers.
Proposed contribution	The paper highlights the shift in meaning of the word fork, noting the coexistence of the traditional, narrower view of forking and the more recent, broader view.

6 DISCUSSION

This chapter covers the main findings and contributions of this dissertation, revisits the research questions, compares the findings to those of previous work in the area, discusses managerial implications, proposes some avenues for future study, and addresses the validity, reliability, and limitations of this work.

6.1. Main findings and contribution

I chose open source as my topic partially because I was perplexed by how it could work quite as well as it appeared to, particularly given the existence of the right to fork. The overarching insight from my time spent researching forking is one that may at first seem counterintuitive to those unfamiliar with open source: open source does not work as well as it does *despite* the right to fork code, but rather *because* of it. The main significance of free and open source software is the freedom it grants the users and developers, rather than its being free in a monetary sense. Similarly, this dissertation finds that freedom is a key feature and benefit of forking, both regarding freedom in development, and as a means of guaranteeing the continued openness of open source software.

Interpreted broadly, forking is at the core of open source. Because of the centrality of forking to open source in general the right to fork is what makes open source an excellent development model. The right to fork, enshrined in open source licensing, protects against developers having to 'reinvent the wheel', and instead allows them to keep going forward, to keep improving existing code as well as writing new. Interpreted more narrowly, forking becomes a much less common phenomenon, but one no less significant. Here, the right to fork becomes both a mediating force as well as an insurance policy. The right to fork is the guardian of freedom and the watchdog of meritocracy.

This dissertation's contribution to the existing body of knowledge is discussed in greater depth in the following subchapters. Five primary contributions worth highlighting are offered below, followed by a more in-depth coverage of the research questions in the subsequent subchapter.

1) **Developers consider the right to fork to be a cornerstone of free and open source software**

As was detailed in paper 4, even though developers do not always agree with the forking of a project, they nonetheless consider the right to fork to be of vital importance.

2) **Forking is commonly done for non-competitive reasons**

Even though the right to fork allows creating identical, competing products, competitive forks are rare. As was discovered in papers 2 and 3, and further highlighted in paper 4, forks commonly have unique goals, features that distinguish them from their parent projects. Even forks that seek to compete with the original for users and developers have pragmatic motivations that set them apart from the original.

3) **Competitive forks are rare but do exist, with one motivating factor for such forks being to ensure the freedom of the code as well as the community's ability to contribute to it**

The case of the MariaDB fork from MySQL (documented in paper 3) illustrates that, while developers will go to great lengths to avoid a competitive fork, motivations can and do exist for such forks to occur. In the case of MariaDB, the motivation was the original developer's fear that the project would either be intentionally handled poorly, or that parts of the program would be changed to closed source.

4) **The term fork is undergoing a broadening of meaning**

Where practitioners have previously had rather narrow definitions of a fork, paper 5 shows that the term now appears to be used much more broadly. Actions that would traditionally have been called a branch, a new distribution, code fragmentation, a pseudo-fork, etc. may all now be called forks by some developers. This appears to be in no insignificant part due to the broad definition and use of the term fork by GitHub.

5) **Forking has drastically affected both the proprietary and open source software landscape**

From the right to fork serving as a cornerstone of open source software, ensuring its contributors that they will always have the ability to pursue their vision for any open source product, to the countless forks that serve diverse needs in both proprietary and open source software alike, to the underdogs that challenge perceived mismanagement in a program through forking it, to the guarantee that even an open source project turned proprietary can live on as open source through forking – it is undeniable that the right to fork has greatly affected the software landscape. Equally indisputable is that it will continue to do so for the foreseeable future.

6.2. The research questions revisited

Several of the papers in this dissertation covered multiple research questions. In the following, each of the four research questions is discussed in turn, highlighting the contributions and findings of the relevant papers.

6.2.1. Research question 1: What significance does code forking have to open source software?

The findings of this dissertation further underline the notion, put forth in earlier texts, that the right to fork significantly affects open source governance and software development. It shapes all aspects of open source software from both a developer and business point of view and is a significant reason why open source has become what it is today. The importance programmers put in the right to fork was particularly shown in papers 3 and 4, and the prevalence of forking and its myriad motivations were detailed in paper 2. Exploring the answer in greater depth, we can draw from the discussion included in paper 1, examining forking's significance for companies, developers, and the software itself.

Businesses can benefit greatly from the right to fork. One way is to develop programs for in-house use that are based on, or include, open source software, thus saving time,

effort, and money. Since the requirement to distribute source code is only triggered upon distribution, the source code of in-house programs need not be disclosed. Open source software, except for programs licensed under the AGPL, is thus also possible to use in so-called Software as a Service, where software is made available for online use only, without the option to download the software. A further benefit is the option to revive an abandoned program, again saving the costs of developing the program from scratch. Furthermore, the right to fork, combined with the hundreds of thousands of open source programs available, allows companies to invent ways of combining existing programs into a new whole, offering novel products to their customers. Moving beyond the individual company, we can note that the right to fork shapes the business models that utilize open source. Indeed, much of the business interest in open source comes in the form of collaboration on projects that are beneficial for multiple companies. The fact that open source can serve business interests is a significant factor in driving investment money into further development of open source software. Somewhat ironically, choosing an open source license can be both a significant asset in developing a piece of software, and be the source of significant financial challenges in financing its development.

While it is open source licensing that gives us the framework for how code forking works, we can better understand the practice of forking by understanding how developers view the right to fork. In considering the significance the right to fork has from the viewpoint of developers, we enter into the area of research questions 2 and 3: how programmers view the right to fork, and why they fork programs. Code forking's significance to a program's community is great as well, with a huge impact on both sustainability and governance. These are among the points covered in answering research questions 2 and 3 in the following subchapters.

Analysing the significance of forking at the level of software programs, we can note that forking helps ensure the survival and development of any program that has a large community interest. An open source program cannot be designed to become obsolete, as any unfavourable actions by those shepherding the program can be countered with a fork intended to safeguard the community's vision for the program. The same is true of vendor lock-in, versioning, and related compatibility problems. In cases where these are necessitated not by technical issues, but rather by the software's overseer intending to act out of personal gain rather than in the best interests of the user community, a fork is always an option. In addition to acting as a counter-measure to planned obsolescence, open source software also need not suffer end-of-life or end-of-service announcements, if the program retains a committed community.

6.2.2. *Research question 2:* How do programmers view the right to fork and the practice of code forking?

Developers consider the right to fork to be central to open source, and a right that must exist regardless of their opinions about specific cases in which it is used. Forking is considered unproblematic and favourable when it is done for the good of the code. This can be the case in a number of different ways, among them to revive an abandoned project, to experiment with new ideas, or to customize a program for a particular need. Furthermore, should the governing body of a program – be it a single programmer, a group, or a corporation – make decisions that go against the interests of one or more of the programmers, the right to fork serves as a means of protecting the software and its community's interests through taking the work done so far and continuing it in the direction of their choosing.

Thus, the right to fork is seen as a mediating force, compelling contributors to find common ground, while at the same time guaranteeing an escape hatch should insufficient common ground be found. This common ground need not be solely about the development of a program. For instance, given that some open source licenses can be changed, it may also be possible for a fork to emerge that focuses primarily on offering similar, if not identical, features under a different license. Conversely, should the owner of a project decide to change the license, those developers that disagree with such a change may establish a fork that remains under the current license. The only kind of fork widely seen as unfavourable by developers is what could be called a malicious fork: a fork whose only purpose is to distract and hinder the work being done on the original. However, such forks are exceedingly rare, with few developers being able to cite even a single example of such a fork.

Regarding the practice of creating a fork, there are little in the way of commonly accepted rules regarding how it should be done. The most commonly held opinion relates to the significance of crediting previous contributors who have created the original program from which the fork is made. A further common view is that one should first contact the developers of the original program to see if a means of achieving the goals of both parties is possible without a fork, thus focusing more development efforts into one piece of software. Should a fork be necessary, developers consider it good practice to attempt to develop the versions in such a way that code can still be shared between the fork and the original.

6.2.3. *Research question 3:* Why do programmers fork code; what are the motivations behind code forking?

In brief, forks are motivated by a desire to improve upon an existing program. A program might be forked if the original is abandoned, or if improvements are not accepted into, or not submitted to, the parent program. However, some additional clarity is sought in the following two subchapters.

6.2.3.1. Fork motivations: five categories

Based on the findings of this book, the overwhelming majority of forks are of a pragmatic nature, done to accomplish a technical or content-related change that could not be implemented in the original. However, there are also forks whose primary motivations are not related to its content or the direction of its development, but instead are related to factors such as governance, licensing, the community's ability to commit code, or a project's being abandoned. In such cases, the fork may not seek to implement any noticeable change in the direction of development. Thus, we can broadly categorize the motivations of forks as follows: direction, licensing, governance, ability to commit code, and abandonment. Multiple categories of motivation can play a part in a developer's decision to start a fork.

Direction: technical and content-related motivations

During the course of any project, there are many decisions that need to be made regarding both content-related and technical issues. Issues like what features to include, in what direction a program should evolve, for what hardware the program should be written, etc. Any of these may plant a seed of discontent that could lead to a fork if left unresolved. Given that the impetus for the fork lies in wanting different

things, such forks may either initially or over time become sufficiently differentiated that they no longer compete for the same user and developer base.

Ability to affect: how easy is it to be heard and to get code commits in?

One point that has been a factor in several of the forks remembered in hacker folklore is the inability to affect a project due to difficulties in getting code committed to it. Of course, this may be because of differing goals, as was discussed in the previous section. However, it may also be a result of governance or other issues. Indeed, in the cases of both NetBSD and Drizzle, frustrations with getting patches into the original helped motivate the eventual decision to fork. In the case of the LibreOffice fork from OpenOffice, it was noted that some developers considered the process of getting patches in too laborious and bureaucratic, and contributing to the project had become less fun (Gamalielsson and Lundell, 2014), thus making the idea of contributing to a fork more interesting.

Governance: how is the community handled, how much is shared?

This category relates to the interaction between project leaders and the community. Issues of particular relevance are those having to do with openness and transparency (regarding decision-making, plans for the future, etc.). However, the interpersonal skills of the project managers and other community members in positions of power may affect developers' decisions to fork.

Licensing: do the developers agree with the license choice? Has there recently been a change in licensing (or is there one on the horizon)?

Here, the primary motivation behind the fork is not to change the program itself, but rather to affect or maintain the laws by which the program is governed. The two main groups of forks found in the data analysed for this dissertation are those that seek to move towards a more strongly copyleft license and those that create a fork with the same licensing as the original. The goal of the latter is to ensure that a copyleft version is maintained should the main version change in part or entirety to a more permissive license, or to include some proprietary elements.[65] This may be coupled with a real or anticipated change in the business model of the original program. In addition to licensing and license change, a related point of contention and challenge that also falls in the legal domain can be that of trademarks (see, e.g., Viseur [2012]; Gamalielsson and Lundell [2012 & 2014]). However, this was not an area of focus.

Abandonment

These forks are motivated by the original program being either abandoned or having unresponsive project leaders, leaving it unclear whether the project is stagnant or abandoned. These types of forks are unique in that there is no disagreement or unease at the basis of this kind of fork. Furthermore, there is only one program in development

[65] Since it is not possible for anyone but the owner of the code to change a copylefted program to a more permissive license, forks (by others than the owner) that change licenses must either change to a license of similar strictness, or one that is stricter still than the previous license. I.e., a permissively licensed program can be changed to LGPL, GPL, or AGPL, but not vice versa. However, a fork that is in fact a rewriting of the original program can of course alter the licensing in either direction.

after the fork: it picks up where previous developers stopped, continuing rather than altering development.

6.2.3.2. Fork motivations: four circles and an arrow

In-depth case studies of fork motivations are rare and more data is needed to establish anything definitive. However, Figure 10 offers a suggestion about the situation based on what has been gathered so far. The figure may give the impression that a fork is likely, however this is not its intent. Rather, taking the potential for a fork as a starting point, it analyses what kind of fork might occur. Though reality is not as clear-cut as the four areas portrayed, Figure 10 is proposed as a useful window through which the potential for, or risk of, a fork might be analysed. On the y-axis is how closely a contributing developer's goals for a project match the actual goals of the project. On the x-axis is how satisfied a developer is with the ease of getting code committed and with issues related to the project's governance and leadership.

Two elements must be highlighted regarding the figure. Firstly, it takes as its starting point that developers are content with the license a project is licensed under. The unique relevance of licensing, and discontent regarding licensing, is discussed separately below. Furthermore, due to their unique nature, abandoned projects are not included in the table. They are inherently lacking in active goals, governance, or the ability to include new code commits from the community.

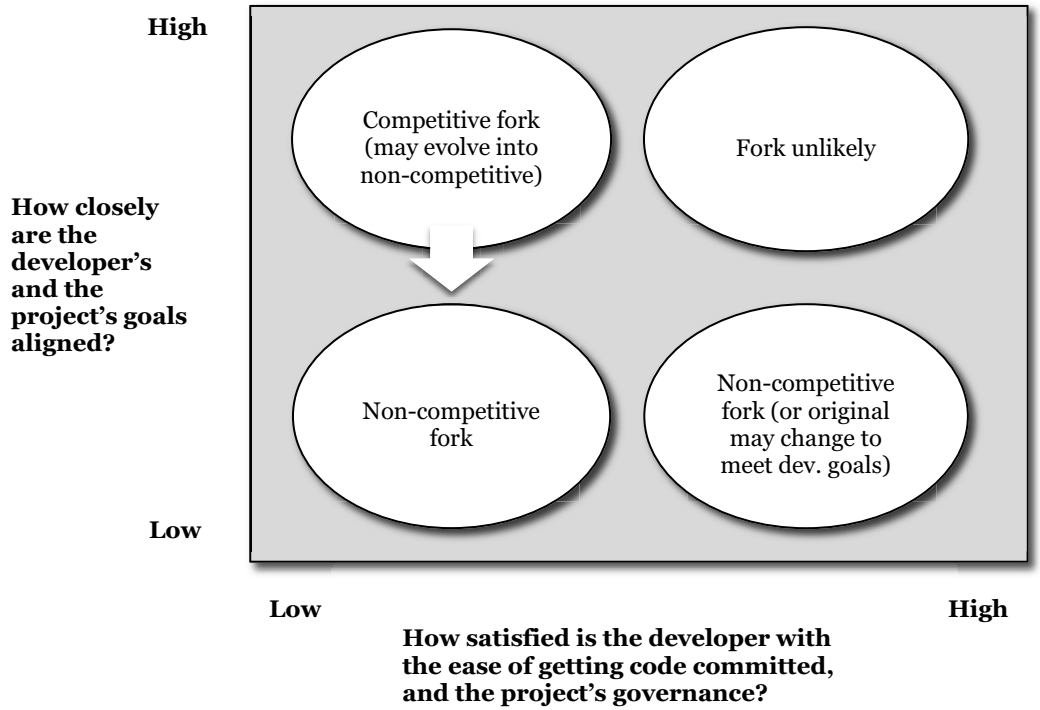


Figure 10 Competitive versus non-competitive fork likelihoods

In the **upper right** is the sweet spot: developer and project goals are aligned, and there are no problems with bureaucracy, governance, or getting code committed.

In the **bottom right** is a situation in which the project is not a good fit for the developer's goals. As such events were not the focus of this dissertation, accordingly, speculation should be restrained. It is possible that a developer might start a fork that better addresses his or her needs. This is likely to primarily be a non-competitive fork. Alternatively, the developer may simply decide to look for a different project that better fits their needs. However, should the developer in question instead commit to the project and find that their patches and additions are readily accepted, the project could over time evolve to meet the developer's needs, changing its goals and thus shifting the project to the upper right of Figure 10.

In the **bottom left** is a situation in which the program does not meet the developer's wishes. Furthermore, getting changes accepted into the project may be difficult or the developer may be dissatisfied with the project's governance, or both. Here, a primarily non-competitive fork is a possibility.

In the **upper left** is a situation in which the developer's goals and the project's goals are closely aligned, but the developer is dissatisfied with how the project is managed or how easy it is to commit code to the project, or both. A fork in such a situation is likely to be of a competitive nature, as the developer of the fork has the same goals as the original development team.

This category is closest to what has traditionally been meant when discussing forking. Consequentially, many of the forks from hacker folklore seem to fit this category, among them such forks as "community forks" and forks due to personal differences. Both the Drizzle and MariaDB forks of MySQL have been called community forks. As was shown in paper 3, MariaDB was born in part from the fear of it becoming impossible to get code committed to MySQL. Drizzle, while it did slim down MySQL into a lighter version and thus was not entirely competitive, was spawned largely by the difficulty of getting code commits accepted into the original. However, the term "community fork" does not generally refer to the motivations behind the fork but rather refers to the origin of the group of developers doing the forking – members of the original program's community. Thus, a community fork may be started for reasons having to do with any of the above categories, or a mixture of them. Forks due to personal differences, rather than differences in project goals, are part of the hacker folklore, and among the few frowned upon types of fork. No clear cases were uncovered during the researching of this book. This is perhaps understandable, as it is unlikely that a developer would choose to highlight such a motivation, even if such forks existed in SourceForge. According to online sources, the OpenBSD fork from NetBSD stemmed from the core developers' lack of acceptance of how a central developer behaved towards NetBSD users and developers. This led to the developer being asked to leave the project (e.g. Glass, 1994), which was followed by the developer in question starting the OpenBSD fork. Thus, personal differences may lie at the heart of a competitive fork. Whatever the specific motivations, over time forks of this type may well evolve to become more non-competitive by altering their focus, thus moving them down to the lower left of the figure.

The license caveat: Even if developers are pleased with the direction, governance, and ability to get code committed to a project, a competitive fork is still possible if developers are displeased with issues regarding licensing. Developers who feel strongly about licensing might even make their decisions regarding joining a project on the basis of licensing. Those who value wide-scale usability of the code, including the ability to use it in combination with proprietary source code, will predominantly favour permissive licensing, while those who predominantly value the guaranteed openness of

the code are likely to favour copyleft licensing. However, as was shown in paper 3, the relevance of licensing can be great in cases where a project changes or is expected to change licenses mid-development.

In conclusion, the findings of this dissertation strongly suggest that the majority of forks are of a predominantly non-competitive nature. They are done to accomplish a technical or content-related change that could not be implemented in the original. Though considerably less common, competitive forks demonstrably also exist. While these are also done to meet a pragmatic need, the need is more often related to the means in which the original is managed, e.g., due to license changes or a fear of a future license change, or due to perceived limitations regarding the community's input and access to the project, and their possibilities to commit to its codebase.

6.2.4. *Research question 4:* Does "fork" mean different things to different programmers?

The final research question addressed the issue of how programmers define forking. The data gathered for this book, including both literature review and interviews, shows that the shortest possible answer to this question is: yes. While there is overlap among the vast majority of definitions, significant differences do exist. It was at one point an ambitious and optimistic goal of this dissertation to offer some kind of resolution on what to do about this problem. However, the complexity and decentralized nature of the problem makes simple solutions not feasible. Instead, the results of the current studies will serve to shed some light on the state of the use of the word fork. Furthermore, this dissertation seeks to offer some ideas as to what might be useful subcategories of forks to provide a more holistic framework for future research and discussion on the topic of forking. This is done briefly below, and in greater depth in paper 5.

All developers interviewed for this book would agree with a very broad categorization of a fork as being when a codebase is copied, in its entirety, and used as the basis for a separate project. However, somewhat differing views can be found regarding the amount of code required for the definition to be accurate. Furthermore, many also considered the term fork appropriate even when done as part of developing a program, without the intent of diverging from it, e.g., in the case of what could be called an experimental fork. Furthermore, some developers noted that something could be considered a fork both if all of the code is reused but left entirely unmodified,[66] as well as even if no code whatsoever is reused.[67] Developers were commonly of the opinion that an internal split is not necessary for a project to be called a fork. No developers highlighted compatibility problems between the original and the fork as a necessary trait for defining something as a fork.

There is a wide array of definitions of the term fork. At one end of the spectrum, forking is used to describe the copying and modifying of a few lines of code. At the other end of the spectrum, a new development is only considered a fork if it copied over the source code along with all available tests, historical data, and other information from the original project. Given the remarkable difference between these extremes, it is clear

[66] If, for instance, the code was unmodified but the software product was externally rebranded.

[67] If, for instance, the entire codebase was rewritten.

that significant variations among programmers' definitions of a fork do exist. This raises at least two questions: *why?*, and *so what?*

Regarding the question of why these differences exist, the data gathered suggests that the differences might stem from a gradual change in definitions, with the younger developers tending to use the more broad definition. The popularity of the version control system GitHub appears to be a significant effector of this change. The term was generally used more broadly among the younger programmers interviewed, who were active users of GitHub, then among the older programmers. The notion that the meaning of the term fork is broadening, and that GitHub is one key reason behind this, was furthermore a hypothesis shared by some of the veteran developers.

Regarding the question of relevance, an awareness among researchers of this spectrum of definitions is important so that they are able to properly frame their research questions and gather the most relevant and interpretable data. When interviewing subjects about forking, or designing surveys to address the topic, researchers must be mindful that the term fork can mean different things to different people. Thus, the interviewee's definition of a fork must also be considered.

6.3. Comparing the findings to previous knowledge

Much of what has been written about forking comes from outside of peer-reviewed academia. Therefore, the findings of this book are compared both with previous academic study as well as with earlier practitioner insights and views.

6.3.1. Academic study

Among the issues studied by Robles and González-Barahona (2012) as well as Viseur (2012) were fork motivations. In their study of 220 forks, Robles and González-Barahona found that technically motivated forks accounted for 27% of the forks studied; discontinuation of an abandoned project motivated 20%; the goal of a "more community-driven development" accounted for 13%; legal issues and "commercial strategy forks" accounted for roughly 10% each; and experimental forks made up just over 2% of the forks. Viseur, in his study of 26 forks, found the majority of forks to be motivated by technical aspects (42%) as well as governance (38%). The end of the original project was the third-largest motivation, accounting for 19% of the forks studied. While the categories of motivations identified in these two studies are not identical in size or specific terminology when compared either with one another or with the findings of paper 2, the overarching motivations as well as the general nature of the findings are very similar: forks predominantly seek to accomplish something novel that is outside the scope of the original program.

Regarding the differences between the findings, two points come to mind. Firstly, the studies had different criteria for defining a fork: Robles and González-Barahona stated specific and quite extensive requirements for inclusion; Nyman and Mikkonen (paper 2 of this dissertation) categorized forks solely based on developers being of the opinion that they are forks; and Viseur gave a fairly broad definition of a fork, but did not report the selection criteria used in the sampling process. This leads us to the second point, regarding the means of data gathering. Robles and González-Barona gathered their forks widely through (among other things) searching the Internet and hacker folklore for forks that fit their definition. Nyman and Mikkonen, on the other hand, gathered

data solely from SourceForge. Viseur did not specify a method of data gathering beyond that the forks were "of popular free and open source projects."

Regarding the relevance of the methodological differences, one central unanswered question is how representative the programs hosted on SourceForge are of programs overall.[68] Similarly, the criteria given for a fork by Robles and González-Barona seem more likely to be met by slightly larger projects, which may in turn have a greater likelihood of being hosted elsewhere. To truly be able to compare the results, the findings of Nyman and Mikkonen would need to be filtered to only leave those forks that fulfil the criteria given for a fork by Robles and González-Barahona. The same is true of Viseur, however given that the article states them to be popular forks, it is more likely that they already meet the selection criteria put forth by Robles and González-Barahona.

Most relevant to the findings of this dissertation is the research done by Gamalielsson and Lundell (2012 & 2014) on the LibreOffice fork from OpenOffice. The data gathered for this book further supports the finding that a fork can achieve long-term sustainability. The findings of this dissertation suggest that this is particularly true in a situation in which the fork is based in whole or in part on a negative reaction to decisions made by the company that manages the original project, features of both the LibreOffice and MariaDB forks. In the event of a program's governing body acting against the interests of the community at large,[69] a fork not only seems likely, but such a fork may even generate more wide-spread support among developers and the broader community than is enjoyed by the original. Thus, a fork could become more sustainable than the project from which it forked. Indeed, this is true of the LibreOffice fork compared to the original (and abandoned[70]) OpenOffice. This dissertation has not studied the sustainability of the MariaDB fork, but as of this writing, in late 2014, it has grown both its contributor and user base and appears to be thriving. When compared with the findings of this book, other findings of Gamalielsson and Lundell's papers can be seen as complementary. Indeed, together the research herein and the research by Gamalielsson and Lundell paint a more complete picture of the motivations behind a competitive fork than either study might on their own.

Some papers that address forking cover a sufficiently different aspect so as not to be directly relevant to the focus of this dissertation. Thus, this book may further speak to both the importance of healthy collaboration and the significance of predicting forks, as was a focus of Azarbakht and Jensen (2014), but not speak directly to their results or findings. In a further paper, Fung, Aurum, and Tang (2012), introduce the concept of social forking, including community, software, as well as other artefacts taken from the original project. While the aforementioned paper was perhaps somewhat lacking in specificity regarding the specifics of the definition of social forking, this dissertation certainly speaks to the importance and usefulness of categorizations for forks. What, exactly, those categories should be is both an important and challenging question.

---

[68] For instance, larger programs may be more likely to handle hosting in-house rather than use a forge, and may thus be underrepresented in the data.

[69] For instance due to issues related to licensing, leadership, or the ability to contribute code.

[70] The LibreOffice fork of OpenOffice was released in January of 2011 (The Document Foundation, 2011). In April of 2011, Oracle, then-owner of OpenOffice, announced their intention to "move OpenOffice.org to a community-based project" (Oracle, 2011a). In June of 2011, Oracle contributed OpenOffice trademarks and code to the Apache Software Foundation (Oracle, 2011b; OpenOffice.org, 2012), forming the basis for what was to become the Apache OpenOffice project. Thus, the original OpenOffice is no longer under development, but two successors, LibreOffice and Apache OpenOfice, remain in active development.

6.3.2. Industry insights

When comparing the findings to the pre-existing body of knowledge from outside of peer-reviewed academia, further similarities are found. Numerous previous authors have underlined the significance of the right to fork (e.g. Moen, 1999; Weber, 2004; Fogel, 2006; Moody, 2009a, and others). This book offers support for this claim. Indeed, one of the primary findings of this dissertation is that programmers consider forking to be a cornerstone of free and open source software development.

Several authors and practitioners have noted that forking is potentially harmful for a project and is hence actively avoided (e.g. Weber, 2004; Fogel, 2006; Meeker, 2008). On the surface, this may seem to contradict the findings of this book. However, when viewed in light of the broadening definition of the term fork over the past decade, the findings of this dissertation can be seen to fully support these previous claims. Indeed, the fork studied in this book that best fits the traditional interpretation of a fork, the MariaDB case, shows that Widenius spent considerable effort during the course of his time with MySQL trying to achieve solutions to problems that were not based on a fork, using the right to fork as a last resort.

Regarding the point that forks are actively discouraged, this book may offer some small clarification to previous statements. Firstly, while developers do consider the right to fork to be potentially harmful to a project, they consider any idea of how to modify or add to a program in a way that makes it accomplish something different from the original, and thus better serving users, to be worthy of development. The first step should be to attempt to get the modifications into the main version. However, in cases where that is not possible, a fork is considered entirely acceptable. In such a case, focus should then be put on trying to share code developments with the original, rather than avoiding the fork entirely. Malicious forks, seeking to accomplish nothing but confusion and delays for the original, were the only type of fork that all developers agreed were discouraged and should be avoided. However, only one developer had actually come across this type of fork.

Secondly, in some readings, the discouragement of forks may be portrayed as an external motivation. While it is true that a fork considered unnecessary and offering of nothing novel may be widely frowned upon, based on the interviews conducted, this dissertation finds the pressure to avoid unnecessary forks to be an intrinsic, pragmatic one. Forks that do not accomplish something that could not otherwise have been accomplished seem to be considered a wasteful use of time and ability.

Finally, a comment could be added regarding the differences between free and open source software enthusiasts. Moody (2001) has put forth what he himself notes to be an oversimplified means of viewing them,[71] a categorization also used in this thesis: purists and pragmatists.[72] I have found such a simplistic categorization to be useful, both initially in myself coming to understand the differences between free and open source software enthusiast, and later in attempting to briefly convey these differences to the uninitiated. What I found when interviewing programmers, among them some

[71] In a blog post for ComputerWorldUK several years later, Moody (2009b) light-heartedly notes: "I dubbed them purists and pragmatists, not because they are perfect labels, but because I am a sucker for alliteration."

[72] One can sometimes hear, among developers and academics alike, the sentiment of being "pragmatic" about software use – meaning that one uses a mix of free and open source programs as well as proprietary programs, based on whatever is the easiest or best way to accomplish one's goals.

free software enthusiasts, was that they were equally motivated by a sense of pragmatism.

Where the majority of those interviewed for this book were driven by pragmatic interests regarding achieving a technically superior program, a smaller group (the free software enthusiasts) were simultaneously very mindful of ensuring the future openness of both the program as well as any forks it may produce. Significantly, the emphasis laid on this goal of openness was in itself based on very pragmatic motivations. For instance having the goal of ensuring that the program may always be kept up-to-date, as well as guaranteeing that any valuable solutions developed remain open for other developers and users to benefit from. While I still agree that purists and pragmatists is a convenient categorization, care might be taken in using it to convey the pragmatic foundation of the purists' approach as well.

6.4. Managerial implications

The right to fork is one major deciding factor regarding open source business models, as this right largely makes the traditional approach used in proprietary software impossible. Through licensing, one can affect what kinds of forks are possible, specifically what kinds of licenses a fork can be licensed under. However, there are no legal means by which one can annul the right to fork, or even prevent a fork from happening. Nevertheless, one can make a fork a laborious process. One means of doing so is by adding trademarks or other similar protected elements to a file, which must first be removed. Still, this makes creating a fork more time-consuming, not impossible.

For a company in charge of an open source program, the right to fork is a risk in that they may lose developers and customers to a competitive fork. However, hackers are pragmatic and don't fork without reason. While starting a fork is easy, keeping it updated and relevant is likely to be both time-consuming and expensive. Furthermore, the business model options available to a fork are significantly more limited than those available to the owner of a project. Therefore, corporate open source ventures must embrace the right to fork, while being designed in such a way that the interests of the community are sufficiently met. If the community considers a company to be a competent and trustworthy shepherd of an open source project, there is little to fear from the right to fork, as it would not be in the community's best interests to do so.

The right to fork can benefit companies looking for a program that can be tweaked to fit their needs. However, submitting modified code back to the original program can be in a company's best interests, as it can decrease the amount of work needed with each new update or release. Additionally, companies should be mindful of the benefits of a large community of developers. Where possible, cooperation with other actors is likely to lead to a better product, developed on more fronts than an in-house fork would.

The basic managerial implications of this dissertation are thus: understanding how forking works and why it is done is of great significance to any company involved with open source software development, specifically those shepherding an open source project. Furthermore, understanding how developers view forking, as well as how they view open source software development in general, is an important step towards being mindful of the kinds of actions that may lead to developers going from contributors to competitors through the starting of a competitive fork. The tension between the need to generate revenue and the desire to maintain the benefits to development that come with the openness of the code is perhaps the biggest challenge in corporate open source

development today. The safest way forward is an approach that manages to generate value for the company without having a negative impact on the product, its openness, and the community's possibilities to affect the product's development. Broadening the interpretation of managerial implications to include project management, i.e. governance, these same issues must be considered vital. Taking one's contributors' views into account and allowing their input to shape the project is a big step towards avoiding a competitive fork.

While forking is always allowed, one is not automatically granted the right to do what one pleases with a fork. Specifically, managers must remember that forked programs cannot always be freely combined with all other programs. License compatibility is a highly significant topic, and while this book has only covered its basics, it is an area that anyone in charge of an open source project, whether a fork or not, should develop expertise in.

6.5. On validity and reliability, limitations, and suggestions for future research

The final questions raised and answered relate to the validity and reliability of the research conducted, its limitations, as well as what related avenues might be of particular interest for future research.

6.5.1. Validity and reliability

Perhaps the greatest concern regarding both the validity (or accuracy) and reliability (or consistency) of the findings relates to the realization, during the writing of this dissertation, of the broadening interpretation of the term fork. Varied interpretations of a term for a phenomenon under study increases the risk that interview subjects are not discussing identical phenomena. Thus, one might get accurate responses to views on forking, but have a different interpretation of what a fork is when compared to one's interviewees, and thus misinterpret the results. Similarly, one may receive answers based on varied or inconsistent interpretations of a fork, without understanding that such differences in interpretation exist, again leading to misleading results.

This challenge, uncovered during the writing of paper 2 (which was chronologically the first paper written of all those included in this book), was addressed in the research design of all later papers. In paper 3, I chose a case study that I considered unproblematic to categorize as a fork: the MariaDB fork from MySQL. In paper 4, addressing programmers' views on forking, I began the interviews with in-depth questions to determine the interviewees' views of what constitutes a fork. This knowledge was used to inform my analysis of their views on forking. Paper 5 addresses this issue specifically, taking some small steps towards outlining the problem and potential first steps towards mitigating it.

That I came to realize the existence of this challenge based largely on the data gathered for paper 2 meant that said data includes forks based on unknown definitions. This was retroactively addressed by labelling them "self-proclaimed" forks. While this does address the definition problem, it still leaves questions regarding the generalizability of the study to some extent unanswerable, as the answer would rely on being able to assess how closely the definitions match, both among the programmers included in the study, and between the programmers included and the general programmer

population. For these same reasons, comparing the results of this study with other large-scale investigations into forks is problematic. While not certain, the findings of paper 5 at least suggest that redoing the study behind paper 2 with a clear-cut means of calibrating definitions of fork would not produce large dissimilarities in the results.

In paper 3, the focus of the case study is targeted at uncovering the motives behind one single decision made by one single person. Though largely unproblematic,[73] triangulation with other data sources was also used, including among other things Widenius's blog and numerous interviews with online news journals that covered related events.

Due in large part to the explorative nature of this book, and the largely un-researched subject area, the dissertation primarily attempts to offer a broad understanding of the phenomenon of forking, why it is done, and how it is perceived. In designing and conducting the research of this book, it seemed appropriate to first document the motivating factors behind forks as well as the perception of forks within the developer community. With this background, some initial insights could then be gained regarding classifying perceptions and motivations by frequency and importance. However, a definitive classification was beyond the scope of this dissertation. Thus, while potentially generalizable, this classification is based on a relatively limited cohort and therefore a wider sampling might reveal slightly different results.

Given the more humble, yet no less significant, goal of uncovering views and motivations, the findings of this book should be considered very generalizable. That is, future research looking into forks not covered in this dissertation is likely to find motivations that are already covered in the pages of this book. Similarly, interviewing other programmers about code forking is likely to reveal similar, if not identical, categories of views.

6.5.2. Limitations

The data for paper 2 was collected using only one forge. Whether a similar study conducted on data from a different forge would give similar distributions of motivations is unknown. Furthermore, the varied interpretations of the term fork would imply that one must be very cautious in generalizing from the findings. Future similar work should take steps to calibrate definitions in the data-gathering phase.

The case study presented in paper 3 is limited in its scope in that I only interviewed Widenius, and not others involved in either the fork or the original program. While I consider this sufficient for the goals of that specific paper – to answer the question why Widenius decided to fork – a greater depth of understanding of the case as a whole, as well as further insight, could have been gained from interviewing additional people.[74] Furthermore, this dissertation has not focused on the likelihood of a fork to succeed. Indeed, forking must be considered the easy part. What comes after – establishing a developer community, gaining customers and an income, technical support staff and a

[73] Barring the rather peculiar potential assumption that the interviewee consistently lied about his motivations.

[74] Since publishing the article, I have interviewed some additional members who were initially a part of MySQL and then moved to MariaDB. These interviews provided a broader picture of significant events both before and after the fork, but did not offer any reasons to doubt the findings of the initial paper.

sales and marketing staff – are the true challenges a competitive fork faces, and the immensity of those challenges must neither be overlooked nor underestimated.

The interviewees in papers 4 and 5 were primarily either from Finland or working in Finland. Furthermore, the sample size is such that it does not inspire generalizing from. Paper 4 in particular should be seen more as an attempt to chart opinions and interpretations in order to offer a starting point for a further, larger study to begin addressing more in-depth, specific questions. The study offers insights into what kinds of views there are of forking, as well as what kinds of opinions programmers have about the practice of forking. It does not definitively prove any connection between these views, nor which groups of programmers are more likely to be of a certain view, or adhere to a certain definition of the term fork.

Paper 5 addresses the challenge of definitions. Here, no concrete definition is put forth as the gold standard, but rather a starting point is suggested based on a previous definition. Also, the paper addressed challenges primarily from a motivational point of view, and does not go into any significant depth regarding the purely technical aspects of forking. Furthermore, the paper does not examine the extent to which further sub-categorizations based on such concerns should be included, or what those categorizations should be.

Code forking as a concept is considered by many to cover a much broader spectrum of actions now than it did when the first texts on forking were written. Whereas forking was originally seen mainly as a split among developers who then went on to develop two separate and independent versions of an initially single project, today many seem to interpret it as encompassing almost any kind of large-scale code reuse. The papers in this book tackle the subject of forking covered under both definitions. This necessarily leads to each definition having been given less depth of study than if the focus had been solely on one. However, both have their own insights to offer.

6.5.3. Suggestions for future research

Overall, this dissertation points in many interesting directions, but leaves much of the exploratory journey they would lead to untraveled. Some ideas for future research include:

Perceptions of forking This book scratches at the surface of developer perceptions of forking. How many developers are of what opinion? How strong are these opinions? A greater insight into the topic could tell us much about how actions within project leadership are perceived and what kinds of actions are more likely to lead to a fork. Moreover, the list of which kinds of behaviours make forking more acceptable among developers should be further investigated. Interviews, surveys, and case studies could offer much depth to this knowledge.

Open core, *or*, Now you see it, now you don't – do you mind? It is not unheard of for open source programs to change to an open core approach in order to improve their possibilities of generating a better income. However, as the MariaDB case shows, this can cause a strong reaction within the community. It would be fascinating to gain a better understanding of developers' views of open core approaches: how many developers consider the approach acceptable and under what circumstances? A particular subgroup of this concept includes programs whose licensing is changed after

already having become established, where code that previously would have been visible no longer is.[75]

Life after a fork Forking is easy, it's what comes after that's hard. Gamalielsson and Lundell (2012 & 2014) show that a fork can be sustainable. Further study examining a project's life after a fork could help identify significant factors in achieving sustainability. Studies should focus on approaches to developer and user acquisition and choice of business models. In cases like the MariaDB fork, dual licensing is no longer an option – how do they go about ensuring an income, and how many actually succeed? Further interesting questions are the practice and frequency of establishing a foundation to aid in the development process. An additional issue that is likely to be relevant to the sustainability of a fork is the motivation behind it. Therefore, studies comparing successful and unsuccessful forks could attempt to chart the motivations behind the forks to test for differences among them.

Why fork unless absolutely necessary? Paper 2 revealed that there were many cases in which the necessity, or drivers, of the fork were unclear. Specifically, those forks that addressed bugfixes, code improvements, security issues, etc. Why did these programs fork rather than have their changes incorporated into the main branch? Were the reasons primarily based on the parent project, e.g. due to them not wanting the changes? If so, why? Or was the original project abandoned? Or did the developers of the fork prefer the idea of creating a new fork rather than have their changes incorporated into the original. Again, if so, why? In cases in which the focus of the program seems unchanged, it would seem to go against a pragmatic best practice to start a new fork rather than submitting changes to the original. Could technical aspects or limitations perhaps have played a part? Further research is needed to clarify if such seemingly unnecessary forks are common and what additional motivations they may include.

Defining a fork This dissertation has included a wide variety of forks, mirroring the varying interpretations of the term fork. At its broadest, the definition of a fork found among developers is perhaps unhelpfully broad. From a researcher's point of view, studying forking from this broad definition of the word involves studying everything to do with any kind of large-scale code reuse. In addition to being a dauntingly broad area of focus, such an approach takes away some of the spotlight from what might be the most fascinating of forks: the competitive forks. The establishment of a commonly agreed-upon view of how one should define a fork – one that would take a stand on concepts like fragmentation, pseudo-forking, distributions, code reuse (etc.) and how they fit into the definition – would be a very practical step that could benefit researchers and potentially the open source community as well. Questions regarding possible definitions, steps, and goals for categories of forks are discussed at length in paper 5 and hence will not be repeated here.

Categorizing a fork In addition to a general definition of a fork, it may be useful to develop a means of further categorization into subgroups. Here, the concept of motivation might be employed. However, technical advances are greatly affecting the ease with which forks can be both implemented and maintained. It might also be interesting to explore categorizations based on technical aspects, either on their own or

[75] Based on interviews as well as informal talks with developers during which the concept of open core has come up, I would venture the guess that, like the definition of fork, the definition of open core is not identical among all developers. Any research into the area should therefore be mindful of this in conducting interviews.

in combination with motivational aspects. Such an approach to categorizing forks was not explored in this book mainly due to this author's limited technical expertise. Nevertheless, the results of such a study would be very interesting.

6.6. Summary

Forks are commonly done for primarily non-competitive reasons, with unique features or goals that distinguish them from their parent projects. Competitive forks are rare but do exist, with some motivating factors being to ensure the freedom of the code and the community's ability to contribute to it.

Even though developers do not always agree with the forking of a project, they nonetheless consider the right to fork to be of vital importance and view it as a cornerstone of free and open source software. The only kind of fork frowned upon by all developers interviewed for this book was a fork started with the sole purpose of creating problems for the original project.

In open source, the right to fork is in many ways synonymous with freedom: the freedom to explore and experiment, the freedom to benefit from work done by others, and the freedom to keep any project relevant and vibrant while not having to accept leadership decisions that are deemed insupportable. The right to fork is the guardian of freedom and the watchdog of meritocracy.

While open source is a superior development model, it is simultaneously a challenging foundation for a business venture. Corporate open source ventures must embrace the right to fork, while discouraging competitive forks. One key element of this is ensuring that the interests of the community and the interests of the company are in line with each other.

REFERENCES

Aker, B. (2008) Drizzle, Clouds, "What if?". *Brian "Krow" Aker's Idle Thoughts – Drizzle, Clouds, "What if?"*, available at: http://krow.livejournal.com/602409.html, accessed 11 February, 2014.

Aker, B. (2010) How many contributors does Drizzle have? *Brian "Krow" Aker's Idle Thoughts – How many contributors does Drizzle have*, available at: http://krow.livejournal.com/697962.html, accessed 27 October, 2014.

Aksulu, A. and Wade, M. (2010) A Comprehensive Review and Synthesis of Open Source Research. *Journal of the Association for Information Systems*. Vol. 11, Issue 11.

Allbery, R. (1996) Lignux? 29 May. Available at: https://groups.google.com/forum/#!topic/gnu.misc.discuss/XXUp2GXU2co[1-25-false], accessed 29 January, 2014.

Allman, E. (1980) An Introduction to the Source Code Control System. Available at: http://sccs.berlios.de/man/sccs.me.html, accessed 29 January, 2014.

Aslett, M. (2011a) The trend towards permissive licensing. *The 451 group blog*. Available at: http://blogs.the451group.com/opensource/2011/06/06/the-trend-towards-permissive-licensing/, accessed 27 September, 2013.

Aslett, M. (2011b) On the continuing decline of the GPL. *The 451 group blog*. Available at: http://blogs.the451group.com/opensource/2011/12/15/on-the-continuing-decline-of-the-gpl/, accessed 27 September, 2013.

Assay, M. (2013) Red Hat CEO: Go Ahead, Copy Our Software. *Readwrite*, 13 August. Available at: http://readwrite.com/2013/08/13/red-hat-ceo-centos-open-source#, accessed 10 February, 2014.

AT&T *A Brief History: The Bell System*. Available at: http://www.corp.att.com/history/history3.html, accessed 30 January, 2014.

Azarbakht, A. and Jensen, C. (2014) Drawing the Big Picture: Temporal Visualization of Dynamic Collaboration Graphs of OSS Software Forks. *10th IFIP WG 2.13 International Conference on Open Source Systems, OSS 2014*, San José, Costa Rica, May 6-9, 2014. Proceedings, Volume 427, Berlin, Heidelberg, pp. 41–50.

Bacon, J. (2009) *The Art of Community*. O'Reilly, Sebastopol, CA. Available at: http://www.artofcommunityonline.org/downloads/jonobacon-theartofcommunity-1ed.pdf, accessed 23 June 2013.

Bailetti, T. (2009) How Open Source Strengthens Business Models. *Technology Innovation Management Review*, February.

BBC (2004) *Apple investigates iPod batteries*. 10 February, 2004. Available at: http://news.bbc.co.uk/2/hi/technology/3477497.stm, accessed 28 January, 2014.

Black Duck Software (s.a.) *Top 20 Open Source Licenses*. Available at: https://www.blackducksoftware.com/resources/data/top-20-open-source-licenses, accessed 4 November, 2014.

Bonaccorsi, A. and Rossi, C. (2003) Comparing motivations of individual programmers and firms to take part in the open source movement: From community to business. *Knowledge, Technology & Policy*, Vol. 18, Issue 4, pp. 40-64.

Brooks, F. (1975) *The Mythical Man-Month* (1995 Anniversary edition). Addison-Wesley, Boston, MA.

Carbone, P. (2007) Competitive Open Source. *Technology Innovation Management Review,* July.

Cheatham, T. and Leonard, G. (1963) An Introduction to the CL-II Programming System, *Computer Associates Incorporated Report* No. CA-63-7-SD, August 1963.

Ciffolilli, A. (2004) The economics of open source hijacking and the declining quality of digital information resources: A case for copyleft. *First Monday*, Vol. 9, No. 9, September.

Clark, L. (2014) CentOS Project Leader Karanbir Singh Opens Up on Red Hat Deal. Linux.com, 16 January. Available at: https://www.linux.com/news/featured-blogs/200-libby-clark/757524-centos-project-leader-karanbir-singh-opens-up-on-red-hat-deal, accessed 10 February, 2014.

Connelly, S. (2007) Mapping Sustainable Development as a Contested Concept. *Local Environment: The International Journal of Justice and Sustainability*. Vol. 12, Issue 3, pp. 259-278.

Conway, M. (1963) A Multiprocessor System Design. *AFIPS Proceedings of the fall joint computer conference*, pp. 139-146.

Conway, M. (1968) How do Committees Invent? *Datamation*, Vol. 14, Issue 5, pp. 28-31, April. Available from the author's webpage at: www.melconway.com/Home/Conways_Law.html, last accessed 28 January, 2014

Corbet, J., Kroah-Hartman, G., and McPherson, A. (2012) *Linux Kernel Development: How Fast it is Going, Who is Doing It, What They are Doing, and Who is Sponsoring It* (A White Paper By The Linux Foundation). Available at: http://go.linuxfoundation.org/who-writes-linux-2012, accessed 23 November, 2014.

Coverity (2013) Coverity Scan: 2013 Open Source Report. Available at: http://softwareintegrity.coverity.com/rs/coverity/images/2013-Coverity-Scan-Report.pdf, accessed 3 December, 2014.

Daffara, C. (2011) Open Source License Selection in Relation to Business Models. *Technology Innovation Management Review*, February.

Dannoritzer, C. (Producer & Director) (2010) *The Light Bulb Conspiracy* [Documentary]. Production companies: Arte France, Article Z, Media 3.14, Televisió de Catalunya (TV3), Televisión Española (TVE).

Davidson, A. (2001) *Technology and the Contested Meanings of Sustainability*. State University of New York Press. New York, NY.

de Vaan, M. (2008) Etymological Dictionary of Latin and the other Italic Languages. *Etymological Dictionaries Online*. Edited by Alexander Lubotsky. Brill Online. 20 November, 2012. http://iedo.brillonline.nl/dictionaries/content/latin/index.html.

Dennis, J. and Van Horn, E. (1966) Programming Semantics for Multiprogrammed Computations. *Communications of the ACM*, Vol. 9, No. 3, pp. 143-155.

Dreyfus, P. (1961) Programming on a Concurrent Digital Computer, *Notes of University of Michigan 1961 Engineering Summer conference*, Theory of Computing Machine Design.

Fink, M. (2003) *The Business and Economics of Linux and Open Source*. Pearson Education, Upper Saddle River, NJ.

Fitzgerald, B. (2006) The Transformation of Open Source Software. *MIS Quarterly*, Vol. 30, Issue 3, pp: 587-598.

Fogel, K. (2006) *Producing Open Source Software*. O'Reilly, Sebastopol, CA.

Franke, N. and von Hippel, E. (2003) Satisfying heterogeneous needs via innovation toolkits: the case of Apache security software. *Research Policy*, Vol. 32, No. 7, pp. 1199-1215.

Fung, K.H., Aybuke, A., and Tang, D. (2012) Social Forking in Open Source Software: An Empirical Study. *Proceedings of the CAiSE'12 Forum at the 24th International Conference on Advanced Information Systems Engineering (CAiSE),* Gdańsk, Poland, June 28, 2012.

Gamalielsson, J. and Lundell, B. (2012) Long-Term Sustainability of Open Source Software Communities beyond a Fork: A Case Study of LibreOffice. *Proceedings of the 8th IFIP WG 2.13 International Conference*, OSS 2012.

Gamalielsson, J. and Lundell, B. (2014) Sustainability of Open Source Software communities beyond a fork: How and why has the LibreOffice project evolved? *Journal of Systems and Software*, Vol. 89, March, pp. 128-145.

Gilmore, J. (1983) *Can somebody fork off a "net.philosophy"?* Available at: https://groups.google.com/forum/#!topic/net.misc/sOn4UxVYt-k, accessed 29 January, 2014.

Glass, A. (1994) Theo De Raadt. Email to netbsd usergroup. Available at: http://mail-index.netbsd.org/netbsd-users/1994/12/23/0000.html, accessed 11 February, 2014.

Grad, B. (2002) A Personal Recollection: IBM's Unbundling of Software and Services, *IEEE Annals of the History of Computing*, Vol. 24, No. 1, pp. 64–71.

Haefliger, S., von Krogh, G., and Spaeth, S. (2008) Code Reuse in Open Source Software. *Management Science*, Vol. 54, No. 1, pp. 180-193.

Hair, J., Money, A., Samouel, P., and Page, M. (2007) *Research Methods for Business*. John Wiley & Sons, Chichester, West Sussex.

Hammouda, I., Mikkonen, T., Oksanen, V., and Jaaksi, A. (2010) Open Source Legality Patterns: Architectural Design Decisions Motivated by Legal Concerns. *Proceedings of the 14th International Academic MindTrek Conference: Envisioning Future Media Environments*. ACM, New York, NY.

Hauge, O., Ayala, C., and Conradi, R. (2010). Adoption of open source software in software-intensive organizations - A systematic literature review. *Information and Software Technology archive,* Volume 52, Issue 11, November, 2010, pp. 1133-1154.

Hecker, F. (1999) Setting Up Shop: The Business of Open-Source Software. *IEEE Software*, January/February.

Henkel-Wallace, D. (1997) *A new project to merge the existing GCC forks*. Available at: https://gcc.gnu.org/news/announcement.html, accessed 2 December, 2014.

Himanen, P. (2001) *The hacker ethic, and the spirit of the information age*. Random House, New York, NY.

IBM archives for 1969 (s.a.). Available at: http://www-03.ibm.com/ibm/history/ history/ year_1969.html, accessed 25 September, 2013.

Jolla (2012) *MeeGo Smartphones and Operating System Find a New Life in Jolla, Ltd*. Press release, 7 July, 2012. Available at: http://www.intomobile.com/wp-content/uploads/2012/07/ Jolla_press_release_07072012.pdf, accessed 10 February, 2014.

Kelty, C. (2008) *Two Bits*. Duke University Press. Available at: http://twobits.net/pub/Kelty-TwoBits.pdf, last accessed 27 January 2014.

Lakhani, K. and Wolf, R. (2005) Why Hackers Do What They Do: Understanding Motivation and Effort in Free/Open Source Software Projects. In Feller, J., Fitzgerald, B., Hissam, S., and Lakhani, K. (eds) *Perspectives on Free and Open Source Software*. The MIT Press, Cambridge, MA.

Lakhani, K., Wolf, B., Bates, J., and DiBona, C. (2002) The Boston Consulting Group Hacker Survey. O'Reilly Open Source Conference (July). Presentation available at: mirror.linux.org.au/linux.conf.au/2003/papers/Hemos/Hemos.pdf, accessed 19 February, 2014.

Lehman, M. (1980) Programs, Life Cycles, and Laws of Software Evolution. *Proc. IEEE* Vol. 68, No. 9, pp. 1060-1076.

Lehman, M., Ramil, J., Wernick, P., Perry, D., and Turski, W. (1997) Metrics and laws of software evolution—the nineties view. *Proc. 4th International Software Metrics* Symposium (METRICS '97), pp. 20–32.

Lerner, J. and Tirole, J. (2002) Some Simple Economics of Open Source. *The Journal of Industrial Economics*, Vol. 50, No. 2, pp. 197-234.

Lévénez, E. (2013) *Unix History*, available at: http://www.levenez.com/unix/, accessed 23 May, 2014.

Levy, S. (1974) *Hackers*. (2010 reprint) O'Reilly, Sebastopol, CA.

Lindman, J. and Rajala, R. (2012) How Open Source Has Changed the Software Industry: Perspectives from Open Source Entrepreneurs. *Technology Innovation Management Review*, 2(1): 5-11.

Lindman, J., Juutilainen, J-P., and Rossi, M. (2009) Beyond the Business Model: Incentives for Organizations to Publish Software Source Code. In: C. Boldyreff et al. (Eds.): *OSS 2009*, IFIP AICT 299, pp. 47–56.

Lindman, J., Rossi, M., and Puustell, A. (2011) Matching Open Source Software Licenses with Corresponding Business Models. *IEEE Software*, Vol. 28, Issue 4, pp. 31-35.

Lokhman, A., Mikkonen, T., Hammouda, I., Kazman, R., and Hong-Mei, C. (2013) A Core-Periphery-Legality Architectural Style for Open Source System Development. *Proceedings of the 46th Hawaii International Conference on System Sciences (HICSS)*.

Lundell, B. and Gamalielsson, J. (2011) Towards a Sustainable Swedish e-Government Practice: Observations from unlocking digital assets. Electronic Government and Electronic Participation: *Joint Proceedings of Ongoing Research and*

Projects of IFIP EGOV and ePart 2011. Eds: Janssen, M., Macintosh, A., Scholl, H. J., Tambouris, E., Wimmer, M. A., de Bruin, H., and Tan, Y.-H.

Lundqvist, A. (2012) *GNU/Linux Distribution Timeline*, available at: http://futurist.se/gldt/, accessed 23 May, 2014.

MacDonald, K. (2014) IGN Presents: the History of Atari. *IGN*, 20 March. Available at: http://www.ign.com/articles/2014/03/20/ign-presents-the-history-of-atari, accessed 14 May, 2014.

Markus, L. (2007) The Governance of Free/Open Source Software Projects: Monolithic, Multidimensional, or Configurational?" *Journal of Management and Governance*, 11 (2), 151-163.

McGowan, D. (2000) The Legal Implications of Open-Source Software. Available at SSRN: http://ssrn.com/abstract=243237 or doi:10.2139/ssrn.243237, accessed 6 March, 2011.

McGowan, D. (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.

Meeker, H. (2008) *The Open Source Alternative*. John Wiley & Sons, Inc. Hoboken, NJ.

Meeker, H. (2013) Open Source and the Eradication of Viruses. *Open Source Delivers*, 19 March. Available at: http://osdelivers.blackducksoftware.com/2013/03/19/open-source-and-the-eradication-of-viruses/, accessed 21 May, 2014.

Moen, R. (1999) WHY LINUX WON'T FORK And why being able to fork is still A Good Thing. Available online at: http://linuxmafia.com/faq/Licensing_and_Law/forking.html, accessed 5 February, 2014.

Moody, G. (2001) *Rebel Code – Inside Linux and the Open Source Revolution*. Basic Books, Cambridge, MA.

Moody, G. (2009a) Who owns commercial open source and can forks work? *Linux Journal*, 23 April. Available at: http://www.linuxjournal.com/content/who-owns-commercial-open-source-%E2%80%93-and-can-forks-work, accessed Feb 24th 2010.

Moody, G. (2009b) Of Purists and Pragmatists. *ComputerworldUK*, 23 July. Available at: http://www.computerworlduk.com/blogs/open-enterprise/of-purists-and-pragmatists-3568726/, accessed 31 October, 2014.

Moody, G. (2011) The Deeper Significance of LibreOffice 3.3. *ComputerWorld UK*, 28 January.

Mundie, C. (2001) Speech Transcript - Craig Mundie, The New York University Stern School of Business. Available at: http://www.microsoft.com/en-us/news/exec/craig/05-03sharedsource.aspx, accessed 26 August, 2013.

MySQL (2008) *Sun to Acquire MySQL*. Company news post, 16 January, 2008. Available at: https://www.mysql.com/news-and-events/sun-to-acquire-mysql.html, accessed 10 February, 2014.

Nelson, R. (1993) Shattering – good or bad. Available at: https://groups.google.com/forum/#!topic/gnu.misc.discuss/G7pwuPhnbEM, accessed 29 January, 2014.

Nokia (2011) *Nokia outlines new strategy, introduces new leadership, operational structure.* Press release, 11 February, 2011. Available at: http://press.nokia.com/2011/02/11/nokia-outlines-new-strategy-introduces-new-leadership-operational-structure/, accessed 10 February, 2014.

Nyman, L., Mikkonen, T., Lindman, J., and Fougère, M. (2011) Forking: the invisible hand of sustainability. In Hammouda, I. and Lundell, B. (eds), *Proceedings of the OSS 2011: Towards Sustainable Open Source*.

Nyman, L., Mikkonen, T., Lindman, J., and Fougère, M. (2012) Perspectives on Code Forking and Sustainability in Open Source Software. In: Hammouda, I., Lundell, B. Mikkonen, T., Scacchi, W. (eds.), *Open Source Systems: Long-Term Sustainability, Proc. 8th IFIP WG 2.13 International Conference, OSS 2012*. Vol. 378, pp. 274-279.

O'Mahoney, S. (2007) The governance of open source initiatives: What does it mean to be community managed? *Journal of Management and Governance*, 11 (2), 139-150.

OpenOffice.org (2012) The Apache OpenOffice Project Announces Apache OpenOffice™ 3.4. *News from Apache OpenOffice (8 May 2012)*. Available at: http://www.openoffice.org/news/a0034.html, accessed 23 November, 2014

Oracle (2009) *Oracle Buys Sun*. Press release, 20 April, 2009. Available at: http://www.oracle.com/us/corporate/press/018363, accessed 10 February, 2014.

Oracle (2011a) Oracle Announces Its Intention to Move OpenOffice.org to a Community-based Project. Press release, 15 April, 2011. Available at: https://emeapressoffice.oracle.com/Press-Releases/Oracle-Announces-Its-Intention-to-Move-OpenOffice-org-to-a-Community-based-Project-1ca9.aspx, accessed 23 November, 2014.

Oracle (2011b) Statements on OpenOffice.org Contribution to Apache. *Marketwired News Room*. Available at: http://www.marketwired.com/press-release/statements-on-openofficeorg-contribution-to-apache-nasdaq-orcl-1521400.htm, accessed 23 November, 2014.

Peters, S. (2007) Enterprise Open Source Support. *Technology Innovation Management Review*, November.

Pugh, E. (2002) Origins of Software Bundling. *IEEE Annals of the History of Computing*, Vol. 24, No. 1 (Jan–Mar 2002): pp. 57–58.

Raymond, E. (1997) How to become a hacker. Available at http://www.catb.org/esr/faqs/hacker-howto.html#what_is, accessed 25 September, 2013.

Raymond, E. (1998) Homesteading the Noosphere, available at http://www.catb.org/esr/writings/cathedral-bazaar/homesteading/index.html, accessed 25 September, 2013.

Raymond, E. (1999) *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/, accessed 21 November, 2011.

Raymond, E. (2000) "Forked", entry in *The Jargon file*. Available at: http://catb.org/jargon/html/F/forked.html, accessed 12 February, 2014.

Raymond, E. (2003) The Art of Unix Programming. Available online at: http://www.faqs.org/docs/artu/index.html, accessed 30 January, 2014.

Red Hat (2013a) *Financial Statements*. Available at: http://investors.redhat.com/financials-statements.cfm, accessed 2 September, 2013.

Red Hat (2013b) *Fedora Project and Red Hat Deliver New Linux Platform to Open Source Community*. Press release, 6 November, 2013. Available at: http://www.redhat.com/about/news/press-archive/2003/11/press-fedora, accessed 10 February, 2014.

Red Hat (2014) *Red Hat and the CentOS Project Join Forces to Speed Open Source Innovation*. Press release, 7 January, 2014. Available at: http://www.redhat.com/about/news/press-archive/2014/1/red-hat-and-centos-join-forces, accessed 10 February, 2014.

Richards, P. (1960) Parallel Programming, *Technical Operations Incorporated Report No. TO-B 60-27*, August 1960, p. 4.

Riehle, D. (2007). The economic motivation of open source software: Stakeholder perspectives. *Computer*, *40*(4), 25-32.

Ritchie, D. (1979) The Evolution of the Unix Time-sharing System. Presented at the *Language Design and Programming Methodology Conference*, Sydney, Australia, September 1979. Available online at: http://cm.bell-labs.com/cm/cs/who/dmr/ hist.html, accessed 31 January, 2014.

Ritchie, D. and Thompson, K. (1974) The UNIX time-sharing system. *Communications of the ACM*. Vol. 17, Issue 7, pp: 365-375.

Robles, G. and González-Barahona, J. (2012) A Comprehensive Study of Software Forks: Dates, Reasons, and Outcomes. *Proceedings of the 8th IFIP WG 2.13 International Conference*, OSS 2012, Hammamet, Tunisia.

Robles, G., Scheider, H., Tretkowski, I., and Weber, N. (2001) *Who Is Doing It? A research on Libre Software developers*. Available at: http://widi.berlios.de/paper/study.html, accessed 19 February, 2014.

Schindler, E. (2007) Enterprise Developers Programming Speed? Check. Time to Fix Bugs? Not So Fast. *Cio.com*, 26 September. Available at: http://blogs.cio.com/esther_schindler/enterprise_developers_programming_speed_check_time_to_fix_bugs_not_so_much?, accessed 24 January, 2014.

Skerrett, I. (2011) Best Practices in Multi-Vendor Open Source Communities. *Technology Innovation Management Review*, January.

Smith, D., Alshaikh, A., Bojan, R., Kak, A., and Manesh, M. (2014) Overcoming Barriers to Collaboration in an Open Source Ecosystem. *Technology Innovation Management Review*, 4(1): 18-27.

Spinrad, P. and Meagher, P. (2007) Project Genie: Berkley's piece of the computer revolution. *Forefront*. Fall issue. Available at: http://coe.berkeley.edu/news-center/publications/forefront/archive/forefront-fall-2007/features/berkeley2019s-piece-of-the-computer-revolution, accessed 28 January, 2014.

Sprewell (2010) Towards A Real Business Model For Open-Source Software. *Phoronix.com*, 29 April. Available at: http://www.phoronix.com/scan.php?page=article&item=sprewell_licensing, accessed 8 September 2014.

St. Laurent, A. (2004) *Open Source & Free Software Licensing*. O'Reilly, Sebastopol, CA.

Stallman, R. (1983) *Free Unix!* Reprint of original Usenet post available at: http://www.gnu.org/gnu/initial-announcement.html, accessed 24 January, 2014.

Stallman, R. (1985) The {GNU} Manifesto. *Dr. Dobb's Journal of Software Tools, Vol. 10, Number 3*. Reprint available at: http://www.gnu.org/gnu/manifesto.html, accessed 24 January, 2014.

Stallman, R. (1996) *Linux and the GNU system*. Available at: http://lorenzopena.es/linux/gnulinux.htm, accessed 29 January, 2014.

Stallman, R. (1999) The GNU Operating System and the Free Software Movement In: Dibona, C., Ockman, S., Stone, M. (eds), *Open Sources: Voices from the Open Source revolution*. O'Reilly, Sebastopol, CA.

Suehle, R. (2012) What is the secret of Red Hat's Success? *Technology Innovation Management Review*, 2(1): 25-28.

Tai, L. (2001) *The History of the GPL*. Available at: http://www.free-soft.org/gpl_history/, accessed 14 January, 2014.

Tech News Central (2013) *First Jolla Phone with Sailfish OS to launch on November 27th*. News item, 14 November, 2013. Available at: http://www.technewscentral.co.uk/first-jolla-phone-with-sailfish-os-to-launch-on-november-27th/id_7134, accessed 3 February, 2014.

The Document Foundation (2011) The Document Foundation Launches LibreOffice 3.3. *The Document Foundation Blog*. Available at: http://blog.documentfoundation.org/2011/01/25/the-document-foundation-launches-libreoffice-3-3/, accessed 23 November, 2014.

The Economist (2009) *Planned Obsolescence*. 23 May, 2009. Available at: http://www.economist.com/node/13354332, accessed 22 May, 2014.

Thierer, A. (1994) Unnatural Monopoly: Critical Moments in the Development of the Bell System Monopoly. *Cato Journal*, Vol. 14, No. 2.

Thompson, R. and Wilkinson, J. (1963) The D825 Automatic Operating and Scheduling Program, *AFIPS '63 (Spring) Proceedings of the May 21-23, 1963, spring joint computer conference,* pp. 41-49.

Tirole, J. and Lerner, J. (2005) The Scope of Open Source Licensing. *Journal of Law, Economics, and Organization*. Vol. 21, No. 1, pp. 20-56.

Torvalds, L. and Diamond, D. (2001) *Just for Fun: the story of an accidental revolutionary*. Harper-Collins Publishers, New York, NY.

Vaughan-Nichols, S. (2009) Red Hat Linux without the Red Hat. *Computerworld*, 3 December. Available at: http://blogs.computerworld.com/15191/red_hat_linux_without_the_red_hat, accessed 10 February, 2014.

Viseur, R. (2012) Forks impacts and motivations in free and open source projects. *International Journal of Advanced Computer Science and Applications*, Vol. 3, No. 2.

von Hippel, E. and von Krogh, G. (2003) Open Source Software and the "Private-Collective" Innovation Model: Issues for Organization Science. *Organization Science*, Vol. 12, Issue 2, pp. 208-223.

von Krogh, G. and von Hippel, E. (2006) The Promise of Research on Open Source Software. *Management Science*, Vol. 52, No 7, pp. 975-983.

von Krogh, G., Spaeth, S., and Lakhani, K. (2003) Community, joining, and specialization in open source software innovation: a case study. *Research policy*, Vol. 32, Issue 7, pp. 1217-1241.

Välimäki, M. (2005) *The Rise of Open Source Licensing: A Challenge to the Use of Intellectual Property in the Software Industry*. ISBN: 952-91-8779-3. Available at: http://lib.tkk.fi/Diss/2005/isbn9529187793/isbn9529187793.pdf, accessed 5 November, 2014.

Weber, S. (2004) *The Success of Open Source*. Harvard University Press, Cambridge, MA.

Weiss, M. (2011) Control and Diversity in Company-Led Open Source Projects. *Technology Innovation Management Review*, April.

West, J. (2003) How open is open enough? Melding proprietary and open source platform strategies. *Research policy* 32, pp. 1259-1285.

West, J. (2006) Patterns of Open Innovation in Open Source Software. In Chesbrough, H., Vanhaverbeke, W., and West, J. (eds), *Open Innovation*. Oxford University Press. New York, NY.

West, J. (2007) Value Capture and Value Networks in Open Source Vendor Strategies. *Proceedings of the 40th Annual Hawaii International Conference on Systems Sciences*.

West, J. and Gallagher, S. (2006) Challenges of Open Innovation: The Paradox of Firm Investment in Open Source Software. *R&D Management,* 36, 3, pp. 315-328.

West, J. and O'Mahoney, S. (2005) Contrasting Community Building in Sponsored and Community Founded Open Source Projects. *Proceedings of the 38th Annual Hawai'i International Conference on System Sciences*, Waikoloa, Hawaii.

West, J. and O'Mahoney, S. (2008) The Role of Participation Architecture in Growing Sponsored Open Source Communities. *Industries & Innovation*, 15, 2 (April 2008): 145-168.

Wheeler, D. (2007a) Why Open Source Software / Free Software (OSS/FS, FLOSS, or FOSS)? Look at the Numbers! Available at: http://www.dwheeler.com/oss_fs_why.html, accessed 24 January 2014.

Wheeler, D. (2007b) The Free-Libre / Open Source Software (FLOSS) License Slide. Available at: http://www.dwheeler.com/essays/floss-license-slide.html, accessed 24 January, 2014.

Wheeler, D. (2009) F/LOSS is Commercial Software *Technology Innovation Management Review*, February.

Widenius, M. (2008) What if. *Monty Says*, 22 July. Available at: http://monty-says.blogspot.fi/2008/07/what-if.html, accessed 11 February, 2014.

Widenius, M. and Nyman, L. (2013) Introducing "Business Source": The Future of Corporate Open Source Licensing? *Technology Innovation Management Review*, 3(6): 5-12.

Widenius, M. and Nyman, L. (2014) The Business of Open Source – A Primer. *Technology Innovation Management Review*, 4(1): 4-11.

Williams, S. (2002) *Free as in Freedom: Richard Stallman's Crusade for Free Software*. O'Reilly Media, Sebastopol, CA.

Yin, R. (2003) *Case study research: design and methods* (Third Edition).Sage, Thousand Oaks, CA.

Yin, R. (2009) *Case Study Research* (Fourth Edition). Sage, Thousand Oaks, CA.

Young, R. (2002) Giving It Away: How Red Hat Software Stumbled Across a New Economic Model and Helped Improve an Industry. In: Dibona, C., Ockman, S., Stone, M. (eds.), *Open Sources*. O'Reilly, Sebastopol, CA.

References for personal correspondence

Conway, M. (2012) Email to author, 2 November, 2012.

Deutsch, L P. (2012) Email to author, 2 November, 2012.

Lampson (2012) Email to author, 27 September, 2012.

Lichtenberger, W. (2012) Email to author, 1 November, 2012.

Meeker, H. (2010) Email to author, 2 September, 2010.

Russell, S. (2013) Email to author, 21 June, 2013.

Spencer, D. (2012) Senior Curator of the Computer History Museum. Email to author, 24 November, 2012.

Thompson, K. (2012) Email to author, 27 September, 2012.

Tucker, E., Dr. (2012) Faculty of Oriental Studies at the University of Oxford. Email to author, 5 December, 2012.

APPENDIX 1 THE OPEN SOURCE DEFINITION

The following is the Open Source Definition (OSD), maintained by the Open Source Initiative. The OSD can be found at: http://opensource.org/osd. *The version included here was accessed 11 February, 2014.*

Introduction

Open source doesn't just mean access to the source code. The distribution terms of open-source software must comply with the following criteria:

1. Free Redistribution

The license shall not restrict any party from selling or giving away the software as a component of an aggregate software distribution containing programs from several different sources. The license shall not require a royalty or other fee for such sale.

2. Source Code

The program must include source code, and must allow distribution in source code as well as compiled form. Where some form of a product is not distributed with source code, there must be a well-publicized means of obtaining the source code for no more than a reasonable reproduction cost preferably, downloading via the Internet without charge. The source code must be the preferred form in which a programmer would modify the program. Deliberately obfuscated source code is not allowed. Intermediate forms such as the output of a preprocessor or translator are not allowed.

3. Derived Works

The license must allow modifications and derived works, and must allow them to be distributed under the same terms as the license of the original software.

4. Integrity of The Author's Source Code

The license may restrict source-code from being distributed in modified form *only* if the license allows the distribution of "patch files" with the source code for the purpose of modifying the program at build time. The license must explicitly permit distribution of software built from modified source code. The license may require derived works to carry a different name or version number from the original software.

5. No Discrimination Against Persons or Groups

The license must not discriminate against any person or group of persons.

6. No Discrimination Against Fields of Endeavor

The license must not restrict anyone from making use of the program in a specific field of endeavor. For example, it may not restrict the program from being used in a business, or from being used for genetic research.

7. Distribution of License

The rights attached to the program must apply to all to whom the program is redistributed without the need for execution of an additional license by those parties.

8. License Must Not Be Specific to a Product

The rights attached to the program must not depend on the program's being part of a particular software distribution. If the program is extracted from that distribution and used or distributed within the terms of the program's license, all parties to whom the program is redistributed should have the same rights as those that are granted in conjunction with the original software distribution.

9. License Must Not Restrict Other Software

The license must not place restrictions on other software that is distributed along with the licensed software. For example, the license must not insist that all other programs distributed on the same medium must be open-source software.

10. License Must Be Technology-Neutral

No provision of the license may be predicated on any individual technology or style of interface.

APPENDIX 2 THE INTERVIEW GUIDE FOR PAPERS 4 AND 5

Question 3, and its many follow-up questions, focus on the research questions of paper 5, while questions 3 through 9 address the research questions of paper 4. The first two questions were included both to break the ice as well as to gain a better understanding of the programmers' background and level of experience. Even with this primary division of purpose behind the questions, many answers offered relevance to other research questions as well. Additionally, for paper 4, question 3 was used to calibrate the answers given and exclude those with widely different views on what a fork is.

1) How did you become a programmer?
2) What is your open source background?
3) When, in your opinion, is it appropriate to use the term "fork"?
   - i Does it matter to the definition whether I plan to merge the program back with the original?
   - ii Does it matter to the definition whether I plan to modify the program?
   - iii Is it a fork if a functionality has been reused as part of the new program, but it is buried/invisible inside it, or at least not prominent?
     - What if the functionality is the entire forked program vs. only part of it?
   - iv Does the term require community involvement? Is it a fork if I take code for a personal program that I never intend to share with anyone else?
4) What are the benefits of the existence of forking as a practice?
5) What are the drawbacks of the existence of forking as a practice?
6) When, or under what circumstances, is it acceptable/OK to fork a program?
   - i Have you ever forked a program, or been involved in a forked program?
   - ii Under what kinds of circumstances would you yourself consider forking a program in which you were participating?
   - iii Can you think of any examples of forking where you felt it was good that a fork had happened?
7) When is it not acceptable/OK to fork a program?
   - i Under what kind of circumstances would you consider it a bad thing that a fork happened?
   - ii What kinds of forks do you hope could be avoided?
   - iii Can you think of any instances of forking which you felt it was a bad thing that a fork had happened?
8) When someone forks a program, are there any moral obligations towards anyone? The original developers? The community? Others?
9) Is there a proper code of conduct when one wants to fork a program, and if so, what is it?

APPENDIX 3 PAPERS

Paper 1: Nyman and Lindman (2013) Code Forking, Governance, and Sustainability in Open Source Software. Technology Innovation Management Review, 3(1): 7-12.

Paper 2: Nyman and Mikkonen (2011) To Fork or not to Fork: Fork Motivations in SourceForge Projects. Published in: i) Proceedings of the OSS 2011, ii) The International Journal of Open Source Software and Processes. Vol. 3, Issue 3. DOI: 10.4018/jossp.2011070101

Paper 3: Nyman (2013) Freedom and forking in open source software: the MariaDB story. Proceedings of the 22nd NFF/Nordic Academy of Management conference, Reykjavik, 21-23 August.

Paper 4: Nyman (2014) Hackers on Forking. *Proceedings of the 10th International Symposium on Open Collaboration* (OpenSym 2014). ACM, 2014.

Paper 5: Nyman (2014) When is a Fork a Fork? On the Definition(s) of Code Forking. Unpublished essay.

PAPER 1

Code Forking, Governance, and Sustainability in Open Source Software

Nyman and Lindman (2013). Code Forking, Governance, and Sustainability in Open Source Software. Technology Innovation Management Review, 3(1): 7-12.

Technology Innovation Management Review January 2013

Code Forking, Governance, and Sustainability in Open Source Software

Linus Nyman and Juho Lindman

> *"The ability to fork code – a central freedom of open source software – is what keeps communities vibrant and companies honest."*
>
> Glyn Moody
> Technology writer and journalist

The right to fork open source code is at the core of open source licensing. All open source licenses grant the right to fork their code, that is to start a new development effort using an existing code as its base. Thus, code forking represents the single greatest tool available for guaranteeing sustainability in open source software. In addition to bolstering program sustainability, code forking directly affects the governance of open source initiatives. Forking, and even the mere possibility of forking code, affects the governance and sustainability of open source initiatives on three distinct levels: software, community, and ecosystem. On the software level, the right to fork makes planned obsolescence, versioning, vendor lock-in, end-of-support issues, and similar initiatives all but impossible to implement. On the community level, forking impacts both sustainability and governance through the power it grants the community to safeguard against unfavourable actions by corporations or project leaders. On the business-ecosystem level forking can serve as a catalyst for innovation while simultaneously promoting better quality software through natural selection. Thus, forking helps keep open source initiatives relevant and presents opportunities for the development and commercialization of current and abandoned programs.

Introduction

This article addresses the question of how the right to fork open source projects – to use the source code of an existing program to start a new, independent version – works as a governance mechanism to provide sustainability in open source software. The concept of sustainability is under debate, with numerous rubrics against which the sustainability of a product may be measured (e.g., Connelly, 2007: tinyurl.com/atjcgq3; Davison, 2001: tinyurl.com/aukl5ch; McManus, 1996: tinyurl.com/a5usfo3). Within the context of the current study, sustainability is defined as the possibility of an open source program to continue to serve the needs of its developers and users.

While code forking may lead to redundant independent efforts, it represents the single greatest tool available for guaranteeing sustainability in open source software. In this article, we examine code forking within open source initiatives and discuss the managerial implications of code forking. The article is structured as follows: first, we offer some background on code forking; second, we look at how code forking affects governance on the three levels mentioned; finally, we explain the relevance of these findings and their management implications.

Background

Code forking has often been viewed in a negative light. At the core of this negative view is the continued use of a restrictive, and perhaps outdated, definition of the term forking. Until recently, the term fork was mainly used to describe a situation in which a developer community had split into competing camps, each continuing work on their own, incompatible version of the software (see, for example, Raymond, 1999: tinyurl.com/3ald3; Fogel, 2006: tinyurl.com/3dx2py). Hence, the negative tone found in discussions of forking has been related to concerns regarding the hindered progress, wasted resources, and potential demise of one or both of the projects. In recent years, the term forking

Code Forking, Governance, and Sustainability in Open Source Software

Linus Nyman and Juho Lindman

has come to be used in a much broader context, encompassing all cases in which one takes an existing code base and implements it in a separate project (see, for instance, GitHub: tinyurl.com/7uc94sk). In the context of this study, we adhere to this broader definition of forking.

While there are many reasons why projects are forked, the most common reason is the desire to modify the original program to better address a specific need (Nyman and Mikkonen, 2011; tinyurl.com/arntyur). Forks may also be planned, temporary divergences intended to test new ideas and features, with the intention of later integrating effective improvements back into the original (Nyman and Mikkonen, 2011: tinyurl.com/arntyur; see also GitHub: tinyurl.com/7uc94sk). The right to fork code is built into the very definition of what it means to be an open source program. The third criteria of the Open Source Initiative's (OSI; opensource.org/osd.html) definition of open source states that the license "must allow modifications and derived works." Similarly, the Free Software Foundation's Free Software Definition (FSD; gnu.org/philosophy/free-sw.html) states that users have the freedom to "run, copy, distribute, study, change and improve the software." All spinoff initiatives can be considered forks as they are "modified or derived" (OSI) or "copied, changed and improved". The possibility of forking any project affects the governance and sustainability of all open source programs.

Software is editable, interactive, reprogrammable, distributed, and open (Kallinikos et al., 2010; tinyurl.com/4zn6cun). These characteristics dictate that software is prone to being changed, repaired, and updated rather than remaining fixed from the early stages of the design process. The openness combined with the granular composition of the software offer new ways of governance (Benkler, 2006; tinyurl.com/6ftot3). This governance is not tied to over-appropriating a natural resource (Ostrom, 1991; tinyurl.com/b8rc2pu), but rather related to ways in which a group of developers, following institutional rules, collectively produce a public good (Schweik et al., 2010; tinyurl.com/aqxy2jp).

Three Levels of Governance

1. Software level

The nature of the industry dictates that programs cannot maintain a stable steady state for an extended period of time. They must continue to evolve in order to remain useful and relevant. Without continual adaptation, a program will progressively become less satisfactory (Lehman, 1980; tinyurl.com/b2mpkw3). Conversely, truly successful software is able to adapt and even outlive the hardware for which it was originally written (Brooks, 1975; tinyurl.com/awg3rrw). Therefore, the ability to change and evolve is a key component of software sustainability. Although stagnation may be a precursor to obsolescence, obsolescence need not creep into a project over time; it is often a design feature.

Popularized in the 1950s by American industrial designer Brooks Stevens (The Economist, 2009; tinyurl.com/ahws66g), the concept of planned obsolescence stands in stark contrast to the concept of sustainability. Stevens defined planned obsolescence as the act of instilling in the buyer "the desire to own something a little newer, a little better, a little sooner than is necessary" (Brooks Stevens' biography; tinyurl.com/bbs8a3c). Considered "an engine of technological progress" by some (Fishman et al., 1993; tinyurl.com/bye2n5r), yet increasingly problematized in the business ethics literature (Guiltinan, 2009; tinyurl.com/alr2c92), planned obsolescence is part of every consumer's life. Although contemporary software development and distribution have characteristics that differ substantially from the industrial products of the 1950s, the revenue models of companies in the software marketplace often welcome elements such as system versioning, to encourage repurchases of a newer version of the same system, or vendor lock-ins that limit the customer choice to certain providers of system or product (for a further review, see Combs, 2000; tinyurl.com/aq2wl7h). Newer versions of programs may introduce compatibility problems with earlier operating systems or programs (e.g., lack of backwards compatibility in Internet Explorer, Microsoft Office, or OS X's OpenStep APIs). Some programs also introduce new file formats, which can cause compatibility issues with earlier versions of the program (e.g., docx vs. doc). Furthermore, end-of-life announcements and concerns over end-of-support deadlines may encourage users to upgrade, regardless of the real need to do so.

The right to fork code makes implementing such elements impracticable in open source. The right to improve a program, the right to combine many programs, and the right to make a program compatible with other programs and versions are all fundamental rights that are built into the very definition of open source. Research has shown these rights are often exercised (Fitzgerald, 2006; tinyurl.com/al995aj). The result of this constant collaborative improvement in open source systems is that any program with the support of the open source community can enjoy assured relevance rather than planned obsolescence. Furthermore, with renewed community interest, programs that have decayed and fallen into disuse can be revived and up-

Code Forking, Governance, and Sustainability in Open Source Software

Linus Nyman and Juho Lindman

dated by forking the code from the original program. In fact, this is a fairly common practice: of the almost 400 forks studied by Nyman and Mikkonen (2011; tinyurl.com/arntyur), 7% involved the reviving of an abandoned project. As long as there is sufficient community interest in a project, forking can allow for constant improvement in software functionality.

2. Community level

The possibility to fork is central to the governance of any open source community. The shared ownership of open source projects allows anyone to fork a project at any time. Therefore, no one person or group has a "magical hold" over the project (Fogel, 2006; tinyurl.com/ahbh8nt). Since a fork involving a split of the community can hurt overall productivity, Fogel notes that the potential to fork a program is "the indispensable ingredient that binds developers together".

One of the concerns among open source communities is what Lerner and Tirole (2002; tinyurl.com/bfmaxl4) call the hijacking of the code. Hijacking occurs when a commercial vendor attempts to privatize a project's source code. The 2008 acquisition of MySQL (mysql.com), an open source relational database management system, by Sun Microsystems and subsequent acquisition of Sun by Oracle is an example of a case involving community concern over potential hijacking. It had been argued that such a series of acquisitions would lead to the collapse of both MySQL and the open source movement at large (Foremski, 2006; tinyurl.com/yesjhw7). Responding to such claims, Moody (2009; tinyurl.com/cbrq7g) noted that, while open source companies can be bought, open source communities cannot. Forking provides the community that supports an open source project with a way to spin off their own version of the project in case of such an acquisition. Indeed, this is what happened in the case of MYSQL. The original MySQL developer, Michael ("Monty") Widenius, forked the MySQL code and started a new version under a different name, MariaDB, due to concerns regarding the governance and future openness of the MySQL code (for details, see Widenius' blog [February 5, 2009: tinyurl.com/btr9bm6 and December 12, 2009: tinyurl.com/ba58vpp] and press release [tinyurl.com/auvaxbn]).

Similarly, in 2010, community concerns regarding governance led to a forking of the OpenOffice (OO; openoffice.org) project. The Document Foundation, which included a team of long-term contributors to OO, forked the OO code to begin LibreOffice (libreoffice.org). The spinoff project emphasized the importance of a "transparent, collaborative, and inclusive" government (The Document Foundation; tinyurl.com/bzmw5p2). A recent analysis of the LibreOffice project indicates that this fork has resulted in a sustainable community with no signs of stagnation (Gamalielsson and Lundell, 2012; tinyurl.com/a9ev4hu). Given that forking ensures that any project can continue as long as there is sufficient community interest, we have previously described forking as the "invisible hand of sustainability" in open source software (Nyman et al., 2011; tinyurl.com/b8bzorg).

Commonly, forking occurs due to a community's desire to create different functionality or focus the project in a new direction. Such forks are based on a difference in software requirements or focus, rather than a distrust of the project leaders. When they address disparate community needs, different versions can prosper.

In a traditional company, it is the management, headed by the CEO and board of directors, that controls the company and provides the impetus for continued development. While the vision of the leadership is similarly integral to the eventual success of any open source project, their continued control is more fragile and hinges upon their relationship with and responses to the community. Forking cannot be prevented by business models or governance systems. The key lies in appropriate resource allocation and careful community management. Managers must strike a delicate balance between providing a driving force while appeasing and unifying the community. (For an overview of open source governance models, see OSS Watch [tinyurl.com/bjqpnkn]; for discussion on building technical communities, see Skerrett, 2008: [timreview.ca/article/160]; for discussion on open source community management, see Byron, 2009: [timreview.ca/article/258].)

3. Business-ecosystem level

Within the dynamic world of open source software, natural selection acts as a culling force, constantly choosing only the fittest code to survive (Torvalds, 2001; tinyurl.com/aaxqux7). However, the right to fork means that any company can duplicate any competitor's open source software distributions; thus, competitive advantage cannot depend on the quality of the code alone. However, it is worth stressing that possibility does not equal success. The right to fork a commercially successful program with the intention of competing for the same customer base still leaves the would-be competitor with issues regarding trademarks, brand value and recognition, as well as the existing developer and user base of the original program. Even though forking allows companies to compete with identical open source software, it is nevertheless cooperation that is con-

Code Forking, Governance, and Sustainability in Open Source Software

Linus Nyman and Juho Lindman

sidered to be the key to corporate success (Skerrett, 2011: timreview.ca/article/409; Muegge, 2011: timreview.ca/article/495).

Open source software is free, but it is also increasingly developed and supported for commercial gains (Wheeler, 2009: timreview.ca/article/229). While the right to fork may seem to make for a harsh business environment, open source companies can and do thrive. With its billion-dollar revenue (tinyurl.com/b7py36u), Red Hat is one such example. While their revenue primarily comes from subscriptions and services related to their software (see Suehle's [2012; timreview.ca/article/513] TIM Review Q&A for a more in-depth look at the secret of Red Hat's success), Red Hat's programs themselves are largely based on forks of programs by other developers. This phenomenon of combining forked programs is not unique to Red Hat: the hundreds of different Linux distributions (tinyurl.com/85r9o) are all made possible by the forking of existing products and repackaging them as a new release.

Forking lays the building blocks for innovators to introduce new functionalities into the market, and the plethora of online forges have hundreds of thousands of programs available for forking and reuse in any new, creative way the user can imagine, allowing for the rapid adaptation to the needs of end users. Hence, the practice of forking allows for the development of a robust, responsive software ecosystem that is able to meet an abundance of demands (Nyman et al., 2012; tinyurl.com/acg3fp2).

The old adage, "one man's trash is another man's treasure" is particularly salient in open source software development. Soon after Nokia's abandonment of the MeeGo project in 2011 (press release: tinyurl.com/ad5lh6b; MeeGo summary: tinyurl.com/9u4xrno), the Finnish company Jolla announced that it would create a business around its revival, made possible by forking the original code (press release: tinyurl.com/7bzbo9h). On July 16, 2012, Jolla announced a contract with D. Phone, one of the largest cell phone retailers in China, and on November 21 they launched Sailfish OS (tinyurl.com/a4yot8h). However, one does not need to be an open source business to benefit from the right to fork. Forking can also aid companies who choose to use an existing program, or develop it for personal use. The requirement in open source to share one's source code is linked with distribution, not modification, which means that one can fork a program and modify it for in-house use without having to supply the code to others. However, a working knowledge of licenses as well as license compatibility (when combining programs) is crucial before undertaking such an endeavour (for a discussion of licenses, see St. Laurent [2004; tinyurl.com/befxwvc], Välimäki [2005; tinyurl.com/ahljzwu], or Meeker [2008; tinyurl.com/am93qol] for a discussion of architectural design practices in the combining of licenses, see Hammouda and colleagues [2010; tinyurl.com/bfp82mw].

A summary of the ways in which forking can affect governance and help ensure sustainability is provided in Table 1.

Managerial Implications

Managers should consider the following implications of code forking:

• An abandoned project can become a business opportunity.

• Neither business models nor governance systems can completely prevent forking. Thus, developer and community satisfaction is of key importance.

• A strong, vibrant community is a key issue to consider when implementing an open source program. When acquiring systems, the potential of forking in open source software – in particular when coupled with a strong community – provides opportunities to avoid versioning and vendor lock-in to one provider of a product or system. However, while community is important, it is not the only factor to consider. For more on evaluating and selecting open source software for corporate use, see the May 2008 issue of *TIM Review*, including topical articles by Golden (2008; timreview.ca/article/145), von Rotz (2008; timreview.ca/article/147), and Semeteys (2008; timreview.ca/article/146).

• There are thousands of open source programs already in existence, which can be forked. If a need for software arises and open source is an option, begin by analyzing what already exists on code repositories such as SourceForge (sourceforge.net) and GitHub (github.com). Keep in mind that it is distribution, not modification, that obligates the sharing of the source code. Be sure to read up on licenses first!

Code Forking, Governance, and Sustainability in Open Source Software

Linus Nyman and Juho Lindman

Conclusion

Forking sits at the intersection of several different open source topics, such as software development, governance, and company participation in communities and business ecosystems. In the interest of clarity, we have simplified the categorization of the multifaceted concept of forking. In actuality, there is overlap among the categories: a strong community offers better insurance of sustainability of the software level, while better software can more easily attract a bigger community. Both a poorly handled community and an abandoned project can spawn a business ecosystem competitor.

The right to fork code is intrinsic to open source software and is guaranteed by all open source licenses. This right to fork has a significant effect on governance and helps ensure the sustainability of open source software. We have analyzed the effect of forking on three different levels: the software level, the community level, and the ecosystem level. On a software level, code forking serves as a governance mechanism for sustainability by offering a way to overcome planned obsolescence and decay, as well as versioning, lock-in, and related concerns. On a community level, code forking ensures sustainability by providing the community with an escape hatch: the right to start a new version of the program. Finally, on an ecosystem level, forking serves as a core component of natural selection and as a catalyst for innovation. Online forges offer a plethora of publically available programs that can serve as the building blocks of a new creation. Current projects can be forked, abandoned projects can be revived and commercialized, or programs can be combined in novel ways to better meet the needs of both the developers and end users. It is the right to fork that moulds the governance of open source projects and provides the dynamic vigour found in open source computing today.

Table 1. Forking and its effect on governance

Level	How Forking Provides Sustainability	Examples
Software	The right to fork protects against planned obsolescence, versioning, and vendor lock-in	Microsoft Word vs. LibreOffice
	Disuse due to decay can be countered by forking and updating	Fairly common open source practice (for examples, see Nyman [2011; tinyurl.com/arntyur])
Community	Prevents hijacking and other unfavorable actions by project leaders or owners through giving developers the option to continue their own version of the program	MariaDB forked from MySQL, LibreOffice forked from OpenOffice
Ecosystem	Increases innovative potential by allowing for the combination and modification of open source projects	Plethora of different Linux distributions
	Abandoned (or badly handled) projects can be revived, creating new business opportunities	(Abandoned) MeeGo forked to create Sailfish

Code Forking, Governance, and Sustainability in Open Source Software

Linus Nyman and Juho Lindman

About the Authors

Linus Nyman is a doctoral student at the Hanken School of Economics in Helsinki, Finland, where he studies code forking in open source software. When not researching, he can sometimes be found lecturing on corporate strategy or open source software. Other areas of interest include freemium business models and MMORPGs (online gaming). Linus has a Master's degree in economics from the Hanken School of Economics.

Juho Lindman is an Assistant Professor of Information Systems Science at the Hanken School of Economics in Helsinki, Finland. Juho defended his doctoral dissertation focusing on open source software development organization at the Aalto University School of Economics in Helsinki. In the field of information systems, his current research is focused in the areas of open source software development, open data, and organizational change.

Citation: Nyman, L. and J. Lindman. 2013. Code Forking, Governance, and Sustainability in Open Source Software. *Technology Innovation Management Review*. January 2013: 7-12.

(cc) BY

PAPER 2

To Fork or Not to Fork: Fork Motivations in SourceForge Projects

Nyman and Mikkonen (2011) To Fork or not to Fork: Fork Motivations in SourceForge Projects. Published in: i) Proceedings of the OSS 2011, ii) The International Journal of Open Source Software and Processes. Vol. 3, Issue 3. DOI: 10.4018/jossp.2011070101

To Fork or Not to Fork: Fork Motivations in SourceForge Projects

Linus Nyman[1] and Tommi Mikkonen[2]
1 Hanken School of Economics, Helsinki, Finland
linus.nyman@hanken.fi
2 Tampere University of Technology, Tampere, Finland
tommi.mikkonen@tut.fi

Abstract. A project fork occurs when software developers take a copy of source code from one software package and use it to begin an independent development work that is maintained separately from its origin. Although forking in open source software does not require the permission of the original authors, the new version, nevertheless, competes for the attention of the same developers that have worked on the original version. The motivations developers have for performing forks are many, but in general they have received little attention. In this paper, we present the results of a study of forks performed in SourceForge (http://sourceforge.net/) and list the developers' motivations for their actions. The main motivation, seen in close to half of the cases of forking, was content modification; either adding content to the original program or focusing the content to the needs of a specific segment of users. In a quarter of the cases the motivation was technical modification; either porting the program to new hardware or software, or improving the original.

1 Introduction

A project fork takes place when software developers take a copy of the source code from one software package and use it to begin an independent development work. In general, forking results in an independent version of the system that is maintained separately from its origin. In open source software development no permission from the original authors is needed to start a fork. Therefore, if some developers are unhappy with the fashion in which the project is being managed, they can start an independent project of their own. However, since other developers must then decide which version of the project to support, forking may dilute the community as the average number of developers per system under development decreases.

Despite some high-visibility forks, such as the forking of OpenOffice (http://www.openoffice.org/) into LibreOffice (http://www.libreoffice.org/) and the creation of various projects from the code base of MySQL (http://www.mysql.com/), the whole concept of forking has seen little study. Furthermore, developers' motivations for forking are understood even less, although at times it seems rational and straightforward to identify frustration with the fashion in which the main project is being managed as a core reason.

In this paper, we present the results of our investigation of SourceForge (http://sourceforge.net/) for forked projects and the motivations the authors have identified for performing a fork. Furthermore, we categorize the different motivations and identify some common misbeliefs regarding forking in general.

The rest of this paper is structured as follows: Section 2 discusses the necessary background for explaining some of the technical aspects associated with forking, Section 3 introduces the fashion in which the research was carried out, Section 4 offers insight into our most important findings, and Section 5 discusses them in more detail. Section 6 proposes some directions for future research, and Section 7 concludes the paper with some final remarks.

2 Background

When pushed to the extreme, forks can be considered an expression of the freedom made available through free and open source software. A commonly associated downside is that forking creates the need for duplicated development efforts. In addition, it can confuse users about which forked package to use. In other words, developers have the option to collaborate and pool resources with free and open source software, but this is enforced not by free software licenses, but only by the commitment of all parties to cooperate.

There are various ways to approach forking and its study. One is to categorize the different types to differentiate between, on the one hand, forks carried out due to amicable but irreconcilable disagreements and interpersonal conflicts about the direction of the project, and on the other, forks due to both technical disagreements and interpersonal conflicts [1]. Still, the most obvious form of forking occurs when, due to a disagreement among developers, a program splits into two versions with the original code serving as the basis for the new version of the program.

Raymond [2] considers the actions of the developer community as well as the compatibility of new code to be a central issue in differentiating code forking from code fragmentation. Different distributions of a program are considered ‘pseudo-forks’, because at first glance they appear to be forks, but in fact are not, since they can benefit enough from each others’ development efforts not to be a waste, either technically or sociologically. Moody [3] reflects Raymond’s sentiments, pointing out that code fragmentation does not traditionally lead to a split in the community and is thus considered less of a concern than a fork of the same program would be. These sentiments both echo a distinction made by Fogel [1]: it is not the existence of a fork which hurts a project, but rather the loss of developers and users. Here it is worth noting, however, that forking can potentially also increase the developer community. In cases in which developers are not interested in working on the original (for instance due to frustration with the project direction, disagreements with a lead developer, or not wanting to work on a corporate sponsored project), not forking would lead to fewer developers as the developers in question would likely simply quit the project rather than continue work on the original.

Both Weber [4] and Fogel [1] 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 harm the individual project.

Another dimension to forking lies in the intention of the fork. Again, several alternatives may exist. For instance, the goal of forking can be to create different branches for stable and development versions of the same system, in which case forking is commonly considered to serve the interests of the community. At the other extreme lies the hostile takeover, which means that a commercial vendor attempts to privatize the source code [5]. Perhaps somewhat paradoxically, however, the potential to fork any open source code also ensures the possibility of survival for any project. As Moody [6] points out, the open source community and open source companies 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 for 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 it chooses.

3 Research Approach

In the study, we used SourceForge (http://sourceforge.net/) as the repository of open source programs from which we collected forks. SourceForge contains over 260,000 open source projects created by over 2.7 million developers. Creating new projects, participating in those that already exist, or downloading their contents is free, and developers exercise this freedom: programs are downloaded from SourceForge at a pace of more than 2,000,000 downloads daily.[1]

SourceForge offers programmers the opportunity to briefly describe their program, and these descriptions can be searched using keywords. Using this search function, we compiled a list of all of the programs with the word "fork" – as well as dozens of intentionally misspelled variations of the word fork, none of which turned up any hits – in their description. We then analyzed all the descriptions individually to differentiate between them and to sort out programs that the developers claimed had forked their code base from another program (which we call "self-proclaimed forks") from those which included the term 'fork' for some other reason, either to describe a specific functionality of the program or as part of its name (i.e. false positives). Consequently, a program that stated "This is a fork of …" was considered a fork, while a program which noted that it "…can be used to avoid common security problems when a process forks or is forked" was not. If it was impossible to categorize a project based on the available data, it was discarded. Our data consisted

[1] Source: http://sourceforge.net/about, accessed March 9, 2011

of all programs registered on SourceForge from its founding in late 1999 through 31 December 2010, resulting in a time span of slightly more than 11 years. This search yielded a total of 566 programs that developers reported to be forked.

We then analyzed the motivations stated in the descriptions of the forked programs. The coding process was done in three phases. First, we went through all of the descriptions and wrote a brief summary of the motivations, condensing the stated reasons to as few words as possible. Then, we went through all of the motivations and identified common themes, or subgroups of motivations, among them. In cases where the fork included elements from more than one theme, we placed it in the subgroup that seemed the most central to the motivation behind the fork. Finally, we examined the subgroups to identify overarching groups of themes.

To give some examples of the coding, one fork stated: "[Project name] is a fork of the [original project name] project. [The] purpose of [project name] is to add many new features like globule reproduction, text to speech, and much more." The motivation behind the fork was identified as belonging to the subgroup "add content", which in the final step was combined (with a subgroup of programs which sought to focus content) into a group called content modifications. A fork which sought to fix bugs, and a fork which was motivated by porting a program, were first put into separate subgroups, "technical: improvement" and "technical: porting", and then these subgroups were combined into the "technical modifications" group. Further examples from the data are presented in the next section.

Based on the descriptions entered by the developer, we were able to identify motivations for 381 of the forks. The group of forks which we were unable to categorize consisted of two main types of descriptions: firstly, descriptions which offered no insights as to underlying motivations, e.g. programs which simply stated which program they were forked from; secondly, cases in which it was unclear from the description if the elements described were added in the fork or if they existed in the original; in other words, one couldn't determine if the description of the program included the motivation behind the fork, for instance new technical features, or if they were describing pre-existing features common to both the original and the fork.

4 Reasons for Forking

Based on the data obtained, developers commonly attribute their reasons for forking the code to pragmatism. For a variety of reasons, some of which were well documented and some of which were unclear, the original version of the code failed to meet developers' needs. To expand the scope of the system, the developers then decided to fork the program to a version which serves their own needs. The descriptions of the forks include programs which note that certain changes have been made to the fork, as well as those programs which discuss which changes will or should be made to the forked version. In this paper, we have not distinguished between the two: both planned and already implemented changes are treated equally, since the goal was to study motivations rather than eventual implementations. In

general, the forks appear to stem from new developers rather than the original developing team splitting into two camps. In fact, the data contain almost no references to disagreements among developers that might have led to the fork. However, this does not mean that such disagreements could not have existed.

In the following section, we provide a more detailed view of the different motivations we were able to find in the data (n = 381). The main motivations fall into two large groups (content and technical modifications) which comprise nearly three quarters (72%) of all forking motivations. Four smaller groups, all of similar size, comprise an additional 23% of the motivations. These four groups included the reviving of a project, license- or FOS-related motivations, language- or country-related reasons, and experimental forks. The remaining motivations, grouped simply as "other", consisted of diverse yet uncommon reasons. An overview reflecting the numbers of forks appears in Figure 1.

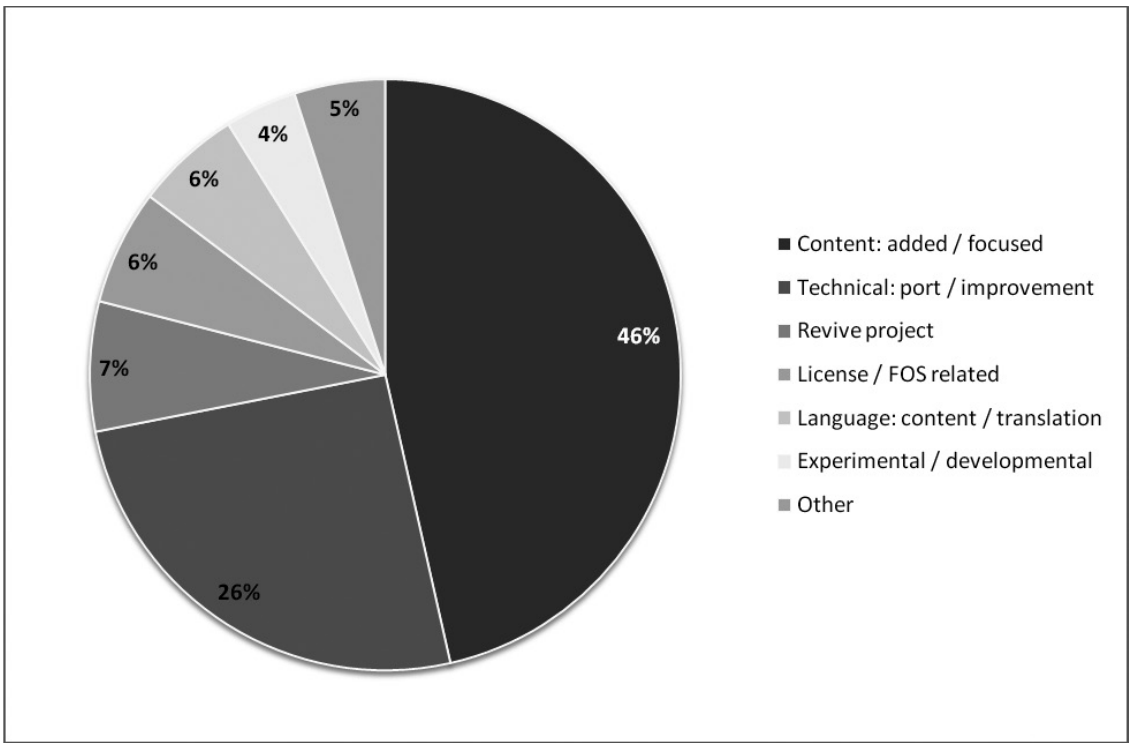


Fig. 1. Fork motivations in SourceForge projects

4.1 Content modifications

Comprising almost half of all forks, content modifications is the largest group. The two main subgroups within the content modifications category, both of which are nearly equal in size, were the adding and the focusing of content; these are briefly discussed below.

Adding content is a self-explanatory reason for making a fork. The developers added new features or other content (e.g. adding better documentation, helper utilities, or larger maps to a game). Quite often, developers didn't describe additions in detail; one developer, for instance, simply noted that the program was a fork "that has the features I'm missing from [the original]." Another developer stated that the fork was "A [program name] fork with more features". In several cases, this group of forks also included bugfixes.

Focusing content implies focusing on the needs of a specific user segment. This category includes forks with both a technical and content-related focus, along with the addition of functionalities and features as well as the removal of elements or features unnecessary for a specific segment or purpose. Examples of content-related focus include programs forked in order to focus on serving the needs of dance studios, radio talk shows, catering companies, program developers, and astronomers, to name but a few. Examples of technical focus include forks "aimed at higher-resolution iOS devices", a fork which "features improvements and changes that make it more oriented for use in a Plone intranet site", and a fork intended "to run on machines that have 800x600 screen resolution". In a minority of the cases in the focusing content category, the original program was forked mainly to remove elements from the original. The main goal in this group was to create a lighter or simpler version of the original, with speed and ease of use as the main focus. One developer stated that the fork was "lightweight, less bloated" and that it was forked to "make [the original] simpler, faster, more useable." Another developer noted that the fork was "Smaller, faster, easy to use."

4.2 Technical modifications

This group, comprising just over a quarter of all forks, can be divided into two subcategories: porting and improving. A characteristic of this category was that little if anything was visibly different to the user; the forked programs simply focused on either porting or improving the original.

Porting the original code to new hardware or software was the more common of the technical motivations for forking, usually involving porting the original to fit a certain operating system, hardware, game, plug-in, migrate to a different protocol, or other such reasons. Examples from the data for this group include a "fork of [program name] to GNU/Linux", a fork "compatible with the NT architecture", "a simple C library for communicating with the Nintendo Wii Remote […] on a Linux system", and a program fork whose main target was to create a version "which works with ispCP." Some forks were ported to reduce a dependency; for instance, one developer who noted that the fork was "geared towards 'freeing' [the original program] from its system dependence, [thus] enabling it to run natively on e.g. Mac OS X or Cygwin." Another developer noted that the program was forked because the developer could not find a "good and recent [program type] without KDE dependency."

Improving the original program was slightly less common in the technical motivations category than porting, which focuses on improving already existing features and contains mostly bugfixes, code improvement and optimization, and security improvements. Some cases were very general in their descriptions, noting only that it was an "upgraded" or "improved" version of the original, or that the code was forked "to fix numerous problems in the code" or to "improve the quality of emulation". Others were more specific, as with the developer of one fork, who notes that "The main goal is to build a new codebase which handles bandwidth restrictions as well as upcoming security issues and other hassles which showed up [during] the last 6 months."

4.3 Reviving an abandoned project

The third common motivation for forking was to continue development of a project considered abandoned, deceased, stalled, retired, stagnant, inactive, or unmaintained. In several of these cases, the developers of the fork note who the original developers are and credit them. In a few cases, the developers of the fork note that they attempted (unsuccessfully) to reach the original developers; in other words, forking the code was the last available option for these developers, as the original developers could no longer be reached. One such example is a fork which the developer notes was "due to long-time inactivity" and then goes on to state "We want to thank the project founder [name] for starting this project and we intend to continue the work". In another case, also due to the inactivity of the original developer, the developer of the fork acknowledges the original author and notes that the fork "includes changes from comments made on his forum." Other examples from the data are: "This project is a fork of the excellent but dead [project name] project", "This project is a fork of the stalled [project name]", "a code fork from the (deceased) [project name] source", and, finally, "The previous maintainer is unresponsive since 2008 and the library [has] some deficiencies that need to [be] fixed. Anyway, thanks for creating this great library [name of original developer]!"

4.4. License/FOS-related issues

This group consists of forks which were motivated by license-related issues or a concern for the freedom of the code. Some of the forks appear to be simply a form of backup copies: stored open source versions of well-known programs. The motivation for this subgroup was a concern that the original version might become closed source. In one case, the developer stated that the fork was due to concern about the future openness of the code. In a similar case, a developer noted about the fork that "This is a still-GPL version of [program name,] just in case." One fork simply identifies the motivation as a "license problem". In five cases, the program was forked because the original was deemed to have become either closed source or

commercial, and in one case, developers noted that the fork occurred because certain bits of the original code were closed source. One fork notes that the new version removes proprietary (boot) code from the program, but that "there is no need to use this version unless you are concerned about the copyright status of the embedded boot code."

4.5 Language- and/or country-specific modifications

A small group of the forks were motivated by language and country. This group could well be considered a subcategory of the "focusing content" group, but was considered separate due to its clear language- and country-related focus. The simplest, though not most common, form of forks included programs which were merely translated into one or more languages; in most cases, however, new content was also added to customize the fork for a specific country and/or group. Some examples are forks created for elections in New Zealand, the right-to-left reading of Hebrew texts, and a program "customized to meet German requirements regarding accounting and financial reporting."

4.6 Experimentation

This group consisted mostly of forks which declared that they existed for experimental purposes, with a handful citing development reasons. A feature common among many of these forks is that the developers state that the fork is temporary and that successful new features or improvements will be incorporated into the original program. Some describe the fork as simply "for testing", while others go into greater detail, noting for instance that the fork is "aimed at experimenting with a number of features turned up to maximum." One developer notes that the fork is simply "for fun", and then goes on to tell readers where they can find the original project.

4.7 Other reasons

Of the remaining forks, a handful described it as a "community fork." In some of these cases, it was possible to identify an overarching motivation behind the community fork; in others it was not, the implications of the term in those cases remaining unclear. Two cases cite a reprogramming in a different programming language as the reason for the fork. The remaining reasons for the forks defied categorization, and included such motivations as a desire to create a study tool for the developer, as well as to test SourceForge for a different project.

Finally, the most surprising of the remaining groups was the group motivated by disagreement or breach of trust. In the beginning of the study, we assumed that a significant number of forks would stem from disagreement between developers. In reality, we were able to identify such forks, but their proportion is quite small: we

identified only four cases, three of which stated that the users sought something the original developers did not intend to implement and one which noted that the fork was a reaction to a breach of trust. Furthermore, even some of these cases may be attributed to the original developers' loss of interest in the project.

5 Discussion

The data in this paper are based on information provided by developers themselves. Many of the cases of self-proclaimed forking – such as when a developer continues an abandoned project – could arguably be defined as something other than a true fork. However, determining forks any other way (other than through the self-proclaimed approach used here) would require a technical definition of a fork that would have to be mined from the project data. At present, no such mechanism seems to exist, and in general, differentiating between forked and fragmented code is an ambiguous practice, unless defined by elements outside of the code itself. Consequently, we have identified the developers as the most reliable source of information, at least at present.

Beyond the challenge of defining a fork, one here also needs to note two issues: how the choice of SourceForge as a sampling frame might affect the data, as well as how accurate, or complete, the descriptions offered there are. The choice of SourceForge could affect the data in several ways. The main question would seem to be whether the characteristics of the average program – or program fork – on SourceForge differ from those of programs hosted on other sites, or from independently hosted programs. For example, given that larger projects often have their own hosting, it is possible that we are seeing only a small number of forks in some categories because projects that would face such issues are not using SourceForge. As to the completeness of the motivations offered by developers, there could be a number of reasons why the information offered is incomplete. For instance, the low frequency of disagreements as a motivational factor in forking may perhaps in part be explained by either a reluctance to mention such disagreements or the limited space offered by SourceForge in which to describe the program. It is also possible that such information, while not stated on SourceForge, would be available on project homepages. Indeed, we came across a project which noted elsewhere that a disagreement among the developers of the original was a factor in the fork; however, the same project did not mention this disagreement in their description on SourceForge.

In general, the results of our study suggest that forking is not a particularly extreme situation in real-life projects. For the most part, developers' motivations are easily understandable, and forking can be considered a reasonable action. However, this does not mean that hostile takeovers are absent from high-profile projects, but simply that in the vast majority of cases, developers appear simply to seek to satisfy their own needs and to develop interesting systems. Such motivations were evident

in the documentation in many ways. Furthermore, crediting the original developers was a rather common practice among those who forked a program, which further emphasizes the fact that forks sought to achieve certain goals, not to compete with existing communities. Perhaps more telling still is that a number of forks noted that they hoped to be temporary, and clearly stated their desire that the bugfixes and improvements introduced in their fork be incorporated into the original program.

6 Future Work

Future work regarding issues associated with forking could take numerous directions. Below we list some of the most promising directions that merit further investigation.

Defining a fork. All of the programs in the data for this article define themselves as forks. In practice, upon more careful review, many of them could perhaps more accurately be categorized as pseudo-forks, code fragmentation, or simply different distributions of a code. The creation of a commonly agreed-upon view of forking vs. fragmentation (or distributions) vs. code reuse would be a very practical step that could benefit both researchers as well as the entire open source community. It could also be possible to define a fork based on technical details, rather than depend on information provided solely by the developers.

Licenses before and after forking. Future researchers could conduct a survey of developers who have forked a program in which they explain their choice of license in comparison to the license of the original program from which they forked.

Perception of forking. Another practical aspect related to forking is how programmers view it; in other words, when is it acceptable to fork, and when is it not? Furthermore, discovering whether certain behaviors make forking more acceptable among developers would be an important direction for such work.

Expanding the data set. Performing a similar study for other sites that host open source projects would contribute to a deeper understanding of forking. Because all the data come from only one source, certain aspects may skew the results. Furthermore, it would be interesting to test if one can tie the observed categories to antecedents or consequences, e.g., are particular kinds of software more likely to fork in particular ways or are particular kinds of forks more successful?

Forking in relation to business. A number of forks we have identified occurred because the original project became closed source. Examining what happened to these projects would deepen our understanding and view of forking in relation to business.

7 Conclusions

Forking is one of the least understood topics in open source development. While often perceived initially as something malicious, the developers who perform the actual forking cite rather straightforward reasons for their actions.

In this paper, we addressed the motivations of developers for performing a fork. The data used in the project originate from SourceForge (http://sourceforge.net/), one of the best-known hosts of open source projects, and focus on "self-proclaimed forks", or programs that the program developers themselves consider to be forks. The motivations behind forking are based on developer input, not on mining technical qualities of the project. However, using only the latter to determine forking would be difficult, as separating forking from other open source-related phenomena is problematic and inconclusive. At the very least, additional data from developers are needed to define forking.

In conclusion, while hostile takeovers and the hijacking of a project as well as a loss of developers after a fork are often associated with forking, the reality is that forks seem to be a lot less dramatic. In fact, forking appears to be more or less business as usual, and developers fork because doing so provides certain benefits for their own goals. While we were able to find forks where the rationale for forking lay in disagreement or trust issues, such cases were few in comparison to the total number of projects we studied.

References

[1] Fogel (2006) Producing Open Source Software. O'Reilly, Sebastopol, CA.
[2] Raymond (2001) The Cathedral & the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary. O'Reilly, Sebastopol, CA.
[3] Moody (2011) The Deeper Significance of LibreOffice 3.3. ComputerWorld UK, January 28.
[4] Weber (2004) The Success of Open Source. Harvard University Press, Cambridge, MA.
[5] Lerner and Tirole (2002) Some Simple Economics of Open Source. The Journal of Industrial Economics, Vol. 50, No. 2, pp. 197-234.
[6] Moody (2009) Who owns commercial open source and can forks work? Linux Journal, April 23.

PAPER 3

Freedom and Forking in Open Source Software: the MariaDB Story

Nyman (2013) Freedom and forking in open source software: the MariaDB story. Proceedings of the 22nd NFF/Nordic Academy of Management conference, Reykjavik, 21-23 August.

Freedom and forking in open source software: the MariaDB story

LINUS NYMAN
Hanken School of Economics
linus.nyman@hanken.fi

Abstract

While significant factors that affect the open source community's interest to participate in a development project have been studied, there has been little focus on the motivating factors that can cause a contributor to become a competitor by utilizing the right to fork a program i.e., to copy an existing program's code base and use it to begin a separate development.

The right to copy an existing program's code base and use it to begin a separate development is guaranteed by all open source licenses. However, this right to fork a program is rarely exercised. Indeed, there is strong social pressure against code forking stemming from the negative side effects of code forking, such as conflict and duplicated efforts among developers.

This paper details the events that led Widenius, the founder of the MySQL project, to decide to fork MariaDB from MySQL. Our findings confirm the previously held notion that there is a high threshold for starting a competing fork. While the few studies that exist of competitive forks find the reasons to be due to disagreement among developers, in the case of MariaDB the fork was caused by Widenius' concerns regarding the uncertainty of the future freedom and openness of the MySQL codebase.

This article makes three contributions. Firstly, it further validates the existing notion that there is a strong threshold to starting a competing fork. Secondly, it offers an in-depth analysis of the events and motivations behind the birth of a fork. Thirdly, it contributes to theory by introducing the freedom factor hypothesis: limiting either developers' freedoms to contribute to a project or the freedom inherent in a project's license increases the likelihood of a fork.

Keywords

Code forking, open source software development, open source business models

Introduction

Before software became a viable market in itself, there was not "open source" or "proprietary" software; there was just software. The computer hardware industry saw software as a tool through which one could access their proprietary hardware. Therefore, they freely provided software to be distributed with their hardware. At this time, much of what users needed was developed by the users themselves sharing code

and ideas among one another. Far from being frowned upon, this practice was actually encouraged by most vendors, as it aided the sale of hardware (Levy, 2010). Out of this group of people who developed and shared software evolved the "hackers". Many of these hackers were MIT computer enthusiasts with their own subculture, values and ethic. Among the central tenets in the hacker ethic were openness, the sharing of knowledge that others can benefit from, and the freedom of information (ibid.). A further element of the hacker attitude was that "no problem should ever have to be solved twice", with closed source licensing (i.e. proprietary software) commonly seen as erecting "artificial technical, legal, or institutional barriers […] that prevent a good solution from being re-used and *force* people to re-invent wheels" (Raymond, 1999*a*; emphasis in original).

Therefore, the open source development model has its roots in the early 'hacker' culture. As computers became more affordable, and thus more commonplace, the need for software grew. In 1969 IBM unbundled their software and hardware (IBM Archives), considered a pivotal event in the birth and growth of the commercial software industry (e.g. Grad, 2002), and with it the growth of the proprietary license. Richard Stallman, who had been a part of the hacker community at MIT, saw the rise of proprietary code around him and wanted to "create a new software-sharing community" (Stallman, 1999). In 1984, in what can arguably be called the beginning of the free (and open source) software movement, Stallman began the GNU project, the goal of which was to create a free, and freely sharable, operating system (ibid.)[1]. In essence, the free and open source movement began not so much to instigate a change in software licensing, but to ensure that software development could continue on as it had before: with free access to code and the right to modify and share programs according to one's own needs.

Over time, corporations began producing, acquiring and distributing open source software, creating a dynamic in which the community desire for freedom and access to the code must coexist with corporate needs. While the direct profitability of a specific open source project may not be the goal of a corporation (e.g. Lerner and Tirole, 2002; West, 2003), corporate participation in the open source market necessitates a significant balance between the needs of the community and those of the corporation.

Types of OS projects, business models, and architectures of participation

Among extant categories of open source projects, a common dividing factor is that of community versus corporation as the owner and driving force behind the project (e.g. West and O'Mahoney, 2005; Markus, 2007; O'Mahoney, 2007). West and O'Mahoney (2008) use the classification terms "autonomous" versus "sponsored" projects. An *autonomous* project is a community-developed project, in which governance and control are shared widely among the community. In some cases, a non-profit foundation is created to support autonomous projects and delineate ownership. In a *sponsored* open source project, the community's short or long-term activities are controlled by one or more corporate entities (ibid.). Figure 1 shows a development model for a sponsored (or "corporate") open source project. In this model, the primary driving force is the sponsor

[1] For more information, see http://www.gnu.org/gnu/thegnuproject.html

company. However, the project also receives ancillary support from the open source community and corporate community, normally made up of companies that use the software or for whom its development is otherwise significant. Members of the corporate community commonly contribute either money or developer time to the project.

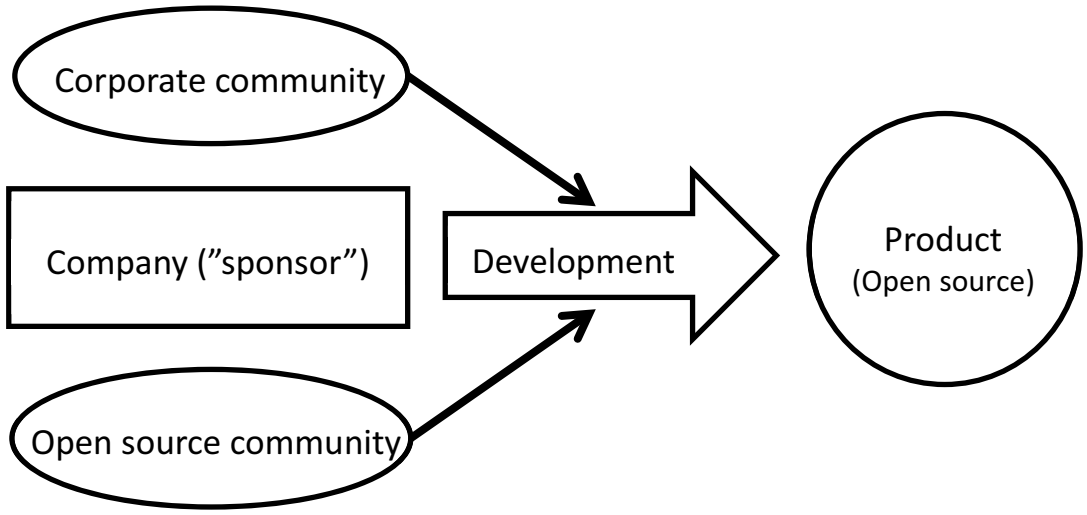


Figure 1: Example of a corporate (“sponsored”) open source project development

Sponsored and community projects differ both in their developer communities and in their architectures of participation (West and O’Mahoney, 2008). One key difference is that sponsored projects need to manage the tension between sponsor and participant goals, essentially community desires and profit. An array of business models have emerged to meet the financial needs of sponsored projects. A central difference between these business models is whether or not the entire product is offered under an open source license. Among those that do offer their entire product as open source, perhaps the most common means of generating income is through the sale of support and other services. Here, the software can be downloaded free of charge and income is generated through providing support and services for the software. Another monetization strategy is to license part of a program as open source, with commercial (i.e. closed source) extensions sold to complement the open source element. This approach is commonly called “open core”, as the “core” of the program is open source. A third approach is dual licensing, in which the program is made available under an open source license, but the company also sells a closed source version of the same program. This is typically of interest mainly for companies that need to embed the company's software within their own proprietary software.

Licensing concerns are known to be a significant factor in corporate software adoption decisions (Daffara, 2011); however, the question of how business models and licensing affect community participation has seen little direct study. The handling of different licenses and using them “in a fruitful manner” is a substantial challenge in the shaping of a corporation's relationship to its open source community (Dahlander and

Magnusson, 2005). In fact, "being specific about licensing practices is a prerequisite for firms to be trusted in the open source community" (Dahlander and Magnusson, 2008). Bonaccorsi and Rossi (2006) note that there is an "'implicit promise' based on the non-written rules of the Free Software community"; if a company breaks this promise, developers are likely to stop cooperating or migrate to other projects. Similarly, Bacon (2009) notes that "the governance body should be tasked with the responsibility of always maintaining and defending the primary values of the community and standing up against any improper requests that may result from commercial sponsors". While the hacker culture is not intrinsically against profitability, it has been documented that some members of the community object to profitability when project income is generated at the cost of openness. This situation is exemplified by the open core approach to licensing. On one side of the debate are those that underline the importance of generating income to fund future development efforts, on the other are those that consider open core to be a "bait-and-switch […] offering the promise of open source but not delivering it" (e.g. Phipps, 2010). Indeed, Dahlander and Magnusson (2005: 490) note that "the developers and users of the Roxen web server went to other projects after the firm released its proprietary add-on". Furthermore, of the four companies they studied, those using an open core business model were less successful in attracting and maintaining a community than those projects that were entirely open source (ibid.). A final community concern worth noting is that of a corporation "hijacking" the code (e.g. Lerner and Tirole, 2002; Ciffolilli, 2004) by changing a project's source code from an open source to a closed source license.

Code forking; when participation becomes competition

As early as the late 1960s, the reuse of code was proposed as a means of building large, reliable software systems in a controlled, cost-effective way (Naur and Randell, 1969; Krueger, 1992). Today, all open source licenses guarantee the right to reuse a program's code (see the Open Source Initiative's open source definition at opensource.org/osd). Thus, it is perhaps not surprising that code reuse has become a common practice in open source software (Haefliger, von Krogh, and Spaeth, 2008). Furthermore, open source licenses do not restrict the amount of code that can be reused, meaning programmers can even reuse entire programs. Such a practice, taking an entire code base to use as the base for a new project, is commonly called forking the code, or "code forking". The ability to fork has a significant effect on the governance and sustainability of open source software (e.g. Nyman et al. 2011 & 2012; Nyman and Lindman, 2013).

The majority of forks[2] are benign, started for a variety of non-competitive reasons such as: to modify a program to better suit their needs; as a means of experimenting with new solutions or features with the intent of merging them back into the original project[3]; or because the original has been abandoned and is no longer being actively developed (Nyman & Mikkonen, 2011; G. Robles and J.M. González-Barahona, 2012). However, a fork may also occur due to disagreements among the developers, resulting in the splitting of the developer group and the forming of a separate, competing project (e.g. Raymond, 1999*b*; Fogel, 2009). This situation will be termed "competing forks" for the

[2] As the term is used by OS developers; among academics the term fork is commonly more restrictive.
[3] Sometimes called "branches" rather than forks.

purposes of this paper. While even a situation in which the forks do compete can have benefits (e.g. Nyman et al. 2012), developers seek to avoid such forks as they are likely to result in a duplication of effort, a splitting and confusing of the community, as well as the potential demise of one (or both) of the projects. Therefore, while open source licenses guarantee developers the right to start a competing fork, social pressure discourages the exercising of that right (e.g. Meeker, 2008). Indeed, such cases are rare enough to be "remembered in the hacker folklore" (Raymond, 1999*b*).

Robles and González-Barahona (2012) studied 220 forks, noting that only 16 of the forks (7.3%) were due to differences among the developer team. Similarly, a study focused on the motivations developers stated for starting forks found only a few indications of differences of opinion[4] and no indications that disagreement among the developers influenced the decision to fork (Nyman and Mikkonen, 2011). Given the rarity of competing forks, it is unsurprising that there is little knowledge of the causes of these forks. Gamalielsson and Lundell (2012) studied the LibreOffice competing fork of OpenOffice, finding that a competing fork can be sustainable over time. However, beyond noting that it had occurred due to tensions in the original project, an in-depth analysis of the motivation behind the fork was not an area of focus. While the limited studies that exist support the notion that competitive forks are rare, we still know little about the exact nature of the disagreements that spawned them. Given that competitive forks are so actively avoided, cases in which they do happen are interesting extremes worthy of study.

Aim and Method

Given the scarcity of studies of the root causes of competitive forks, we examine one case in-depth to answer the question: why was the MariaDB fork started? Secondly, this article aims to determine why the fork was not instigated by earlier conflicts[5]. To answer the research questions, this paper details and analyzes a single-case study (Yin, 2009). The primary source of empirical data is based on interviews with Michael ("Monty") Widenius, the founder of both MySQL and MariaDB. Two in person interviews were conducted, averaging 45 minutes in length. Email and phone conversations were used for follow-up questions. The data set, consisting of transcriptions from the interviews, was complemented with archival research for triangulation (ibid.). The archival research consisted of Widenius' blog; press releases and other relevant information available on the MariaDB, MySQL, MariaDB Foundation, Monty program and Oracle websites, industry articles about MariaDB as well as interviews with Widenius reported in the following journals: ArsTechnica, Computer World, Forbes, H-Online, Info World, ITWire, Linux Journal, Techradar, and ZD Net. We also conducted searches for MariaDB, Widenius, Sun, and Oracle on community site Slashdot, and included all relevant articles. This author had interviewed Widenius once previously for a separate project. The transcripts from that interview were also included in the data used in the creation of this case study.

[4] With licensing being the most common motivating factors stated in such cases

[5] Assuming there is a certain amount of conflict in any given organization, it raises the question *why now*, rather than as a result of any earlier conflict.

Most of the secondary data was gathered and examined prior to the personal interview. The first personal interview was used to clarify the motivations for the fork and fill in the gaps left by the secondary data. The second interview was used to ensure that the representation and interpretation of the case were accurate and complete. As the purpose of this paper was to describe why MariaDB was forked, I have chosen to focus on the viewpoints of Widenius, the person behind the fork, and have not asked other parties of their viewpoints on the matter.

Discussion

This article began with a discussion of a common means of development for sponsored open source projects, an approach also adopted by MySQL. It went on to discuss business models, licensing, architectures of participation, as well as the concept of code forking. We will now discuss in further detail the issues and events relevant to the research questions: why was the MariaDB fork started and are forks actively avoided. A list of significant events during the lead up to the birth of MariaDB is presented in Table 1.

Table 1: Significant events leading up to the MariaDB fork

2001	MySQL takes on outside financing. The shareholder agreement includes a section forbidding the changing of MySQL's license without the consent of the founders (Widenius and Axmark), thereby guarding against a switch to a proprietary or open core business model. The same clause is included in all subsequent financing rounds.
2008, January	**MySQL is acquired by Sun**. (Widenius becomes financially independent.)
2008, January	First **Maria engine** is released. Attempts are made on MySQL's part to stop release unless Widenius changes the name. Widenius replies "Try and stop me."
2008, April	MySQL CEO announces MySQL will include commercial extensions. Widenius informs Sun about the announcement; Sun is against it.
2008, May	MySQL CEO announces that MySQL is and will remain open.
2008, Spring/Summer	**Developers** unhappy at Sun, say (Widenius notes) that they want to be moved to work under Widenius or they will leave.
2008, Fall	**Widenius** leaves Sun, posts an open invitation for MySQL programmers to join his new company Monty Program. The majority ("All but one") of the core MySQL developers join him.
2009, April	**Oracle and Sun** announce acquisition plans [6] [7]
2009-2010	**Widenius** petitions to have MySQL given to third party or to have its licensing changed (see Widenius' blog [8] for more detail). Attempts fail.

[6] http://www.oracle.com/us/corporate/press/018363.
[7] http://www.sec.gov/Archives/edgar/data/709519/000119312509126389/ddefm14a.htm.
[8] http://monty-says.blogspot.fi/2009/12/help-saving-mysql.html.

Research question 1: why was the MariaDB fork started?

After having faced many adverse events, the tipping point came when Widenius no longer felt confident in the future well-being and openness of the MySQL codebase. MySQL had been a strong competitor for Oracle, Widenius notes, in providing an open source alternative (with a dual licensing option) to Oracle's database management product. Widenius believed that Oracle would use the purchase of MySQL to generate income for Oracle in one of two ways: either by "killing" MySQL and thereby gaining customers and market share for their product, or by changing MySQL to an open core business model by combining a mixture of open and closed source parts. Both of these scenarios were unacceptable to Widenius, and the spectre of such actions provided the impetus for starting a competing fork.

Scenario 1: The "killing" of MySQL

With the acquisition of MySQL by Sun then later by Oracle, the companies were essentially paying for control over the project. With this control, the company was then responsible for deciding what contributed code to include in each distribution. They were therefore able to decide what to release when. Even if the open source community or corporate communities would like to participate in the development of an open source program, it is possible for the governing company to turn down this aid, for instance by not including contributed patches or bugfixes developed by the communities. Given that existing versions can be shared freely among interested users, an open source product cannot be killed quickly or immediately by pulling it off the market, as can happen with closed source products. However, open source products can be starved through not including improvements or updates into the codebase and thereby die a slow death. The result would be a program that ceases to evolve and meet the changing needs of its users. Software that does not evolve becomes less and less satisfactory to its users over time (Lehman, 1980), and sooner or later results in the user abandoning the program in favour of a more up-to-date program. In such a situation, the only means of keeping the program "alive" is the forking of the original to continue development on a competing version.

Scenario 2: Commercial extensions

The second possible outcome for the MySQL codebase was that Oracle would change MySQL's business model to an open core model by adding commercial extensions. Widenius is among those who are not favourable to the open core approach. In fact, during one interview he shared that, in his opinion, "open core is not open source". Even prior to its acquisition by Sun, MySQL's management team had been trying to implement an open core model, but Widenius had been able to block such attempts. If My SQL adopted an open core model, development of the codebase could continue, with community and corporate contributions still accepted for consideration, but not all of the resulting product would be open source.

Common to both scenarios discussed is that the solution to maintaining the openness and further development of the code was to fork the program. In a scenario in which the owners of the code restrict its development, a fork would be necessary to be able to continue developing the code. In a scenario in which the owners of the code switch to

including proprietary extensions, a fork would be necessary to maintain a version in which all future additions to the code remain open source. While Widenius did not know at the time which of these would happen, or even if they would happen at all, he viewed the possibility as enough of a threat to the future openness of the code to warrant a fork. Hence, MySQL was forked and MariaDB born in order to ensure that the program would be continued as Widenius intended: constantly evolving and entirely open source.

Although Oracle would have known that a fork was a possibility, Widenius believed that Oracle simply considered it exceedingly unlikely that anyone would be willing to invest heavily enough into a fork to be able to make it successful. While forking the code itself is relatively easy, gaining development, marketing, consulting and services, etc. to support the project is a mammoth task, one requiring no small investment in time and money. Indeed, at the time of the interviews, Widenius had already invested several million Euros in the continued development of the codebase. Due to the sale of MySQL to Sun, Widenius was financially in a position that made it possible to invest money in developing the Maria-branch and the subsequent MariaDB fork. Furthermore, existing developers who already were experts in the code base were interested in working with him, a factor which enabled MariaDB to continue developing with little to no loss of time for the training of new talent. While a fork of the magnitude of MariaDB would technically be possible without money for development, in practice it would be a significantly more challenging endeavour to rely on volunteers working in their spare time[9].

Research question 2: why was the fork not instigated by earlier conflicts?

There were several conflicts during MySQL's existence before the actual fork, among them were the push for competing business models, internal disputes and disagreements regarding the quality of the product.

Competing business model interests. The original (and all subsequent agreements) MySQL shareholder agreement included a section that requires the consent of the founders Widenius and Axmark to change MySQL's license. Over the years, the MySQL management team made multiple attempts to change the MySQL business model to include commercial extensions. However, Widenius always rejected these requests given his strong support for the openness of the project. When MySQL was sold to Sun, the shareholders agreement was made void and Widenius no longer had the authority to stop such changes. Shortly after the acquisition by Sun, in the spring of 2008, MySQL management announced that it would include commercial extensions in the project. Widenius believed that Sun was unaware of this move, so therefore made no move to create an entirely open fork at that time. When Widenius confronted the managers at Sun regarding this change, they assured him that they were unaware of the plans. Soon after, the announcement was retracted[10], with MySQL management noting

[9] As a point of reference, Widenius estimates that to learn the MariaDB or MySQL code base in its entirety would take 2-3 years when studied under the tutelage of someone already knowledgeable in it.

[10] When asked to hypothesize about what he would have done had MySQL's management team managed to change MySQL to open core, Widenius notes that there was a group of developers in MySQL who had

that MySQL is and will always be free. During the codebase's time under MySQL it was kept from being changed to an open core model by the shareholders agreement's requirement of Widenius' consent; during it's time under Sun it was kept from being changed to open core by Sun management's view that MySQL should be open source.

Internal disputes. The Maria storage engine, out of which MariaDB evolved, was a topic of dispute, both regarding its name and the choice of the engine itself as a product. MySQL had used a storage engine called InnoDB. After Oracle, at the time one of MySQL's biggest competitors, purchased InnoDB, MySQL started looking for a replacement. They searched for a replacement over a year with no luck, so Widenius led a team of developers working part-time to develop a new storage engine. Meanwhile, the management team at MySQL believed that the storage engine Falcon, which MySQL had acquired, was the better choice to focus on, rather than waiting for Maria to be completed. In Widenius' opinion, the Maria engine's development received insufficient manpower and support from the MySQL project. Additionally, the naming of the Maria engine sparked further disagreement[11]. During his time at Sun, Widenius' primary obstacle was not direct conflict, but unresponsiveness. He notes that, while initially eager to "help make Sun a better open source company", those he contacted in the organization seemed to ignore his attempts to help. Widenius eventually felt that the best way to continue to develop the MySQL code base was to leave Sun. Even having left Sun, taking many key MySQL developers with him, the goal was not to fork the codebase. On the contrary, he considered leaving Sun to be the best way to focus on the continued improvement of MySQL through improving his experimental version with the goal of implementing the improvements in the original (by selling the improvements to Sun).

Quality of the product. Widenius felt Sun had released a MySQL version lacking in several key areas. While open about his discontent with the release, his goal was still an improved MySQL, not a fork.

Faced with each of these challenges Widenius never planned to fork the MySQL codebase. These findings further support to the notion that there is a high threshold for competitive forking, but raise the question of why the fork was finally started after avoiding it for so many years.

The freedom factor hypothesis

The absence of a fork during the earlier conflicts highlights the active avoidance of competitive forks ingrained within the open source community. Due to this competing

---

formed, ready to spit off into a separate entity to continue work on an open source version of MySQL. However, they would have attempted to work with, rather than against, MySQL, with the goal of achieving the best code possible. In other words, even then they would have attempted to avoid a fork. When asked if he would ever consider joining forces with Oracle to improve the code together, Widenius noted that he would do it "immediately" if the openness of the code were guaranteed.

[11] When it was ready for release, MySQL marketing told Widenius that it should be renamed after a bird, to fit with the Falcon engine, before he can release it. Widenius notes that his reply was "try and stop me", and he released it as Maria.

fork-averse culture, the presence of conflicts alone are insufficient to account for the motivational drive behind at least some competing forks.

Earlier studies have discussed factors that enable and increase participation. It has been noted that managerial actions not supported by the developers can lead some developers to abandon a project, but there is a lack of knowledge regarding the factors that instigate a fork. The MariaDB case shows us a set of circumstances under which a contributor goes from contributor to competitor, rather than the more studied transition of contributor to non-contributor. This transition only occurred after Widenius felt there was a credible threat to the future openness of the project. In order to more fully understand the motivations behind this transition, we propose the freedom factor hypothesis: *limiting either developers' freedoms to contribute to a project, or the freedom inherent in the current license, increases the likelihood of a fork.*

The first part, the freedom to contribute, covers concern over the ability to contribute to the code. There is some support for this hypothesis in the "community forks" of projects, which are sometimes born to enable the community to contribute code more freely, without having to abide by the decisions of the sponsor regarding what is accepted into the program. Furthermore, in extreme cases where a project has been abandoned, it is not unusual that a fork of the program is started to breathe new life into the project (Nyman and Mikkonen, 2011; Robles and González-Barahona, 2012). The second part, the freedom of the license, concerns a change toward a more commercialized software license.

Conclusions

The open source movement was born from the hacker ethic of freedom and sharing. This philosophy is still present in the open source community, and can motivate developer disengagement or even a fork if these freedoms are seen to be hampered or threatened by moves to restrict the evolution or distribution of the program. Competitive forking, where a developer may go from collaborator to competitor, is considered to be the most extreme expression of this desire for freedom and sharing. While the right to fork is guaranteed by all open source licenses, our findings offer further proof of the commonly held notion that actors in the open source community go to great lengths to avoid starting competitive forks. Disagreements among the developers has previously been considered to be a leading cause of forks in open source projects. Conversely, analysis of the events leading up to the MariaDB fork shows that the impetus behind the fork was a perceived threat to the freedom to contribute to the code, its wellbeing, and the future openness of the code base.

This study shows that, in the world of open source software, ownership without trust is exceptionally fragile. Our findings suggest that managers/organizations must convey an unambiguous guarantee of the future openness of the code in order to achieve the greatest potential community contribution. Programmers must feel able to contribute code to a continuously developing program, which they feel confident will remain open source. If these criteria are not met, the likelihood of a fork increases, as such a situation will be unacceptable for at least some programmers. The results are explained through introducing the concept of the "freedom factor hypothesis": limiting either developers'

freedoms to contribute to a project or the freedom inherent in the current license increases the likelihood of a fork. Further study is needed to validate this hypothesis.

Avenues for future study

Open core, business models & licensing. In sponsored open source projects, the hacker mentality and values co-exist with corporate interests and needs. This co-existence can work as long as the needs are not in significant conflict. We posit that the difference between open source and open source with proprietary extensions, or open code, is a significant one. Unfortunately, extant research does not draw a sufficiently clear line between open source and open core and is therefore of limited use in understanding the causal implications of this form of licensing. Further research into clarifying the open source community's views of open core would offer important insights into a topic that is of great significance to both generating income and the attracting and maintaining of contributors. In his blog, Widenius notes that "You can't buy an open source project with money; the currency in open source is trust." There are open core projects that have significant contributor communities and those that do not. Is the difference one of trust? What is the source of that trust? A further significant future avenue of research here would be to find out how large a group is required to precipitate a competing fork.

The business models and project types of forks vs. parents. While Oracle can generate income through the dual-licensing of MySQL, MariaDB no longer owns the necessary code to do so. This creates a challenge in the financing of its development efforts. Future studies could examine the business models of forks vs. their parents to see how successfully forks manage to compete. Is it predominantly through a services model, or are there other approaches? Furthermore, do such forks commonly move from sponsored projects to community, or foundation-run projects, as in the case of MariaDB? Such future research could inform future managers on how to more effectively manage a project, as well as future developers considering a fork.

Motivations for switching. While anyone can start a fork, without developers, customers and community it will be short-lived. MariaDB has been gaining support among all these groups. Why are they switching over? Some cite concerns over the future openness of the MySQL codebase, others a desire to be closer to the open source spirit of things, while still others note the technical superiority of the MariaDB codebase. There is still much research to be done in this area to get a clear picture of both the motivating factors and their implications.

Categorization/taxonomy. Competitive forks are commonly thought to be caused by disagreements among the developers. A clearer understanding of what types of disagreements there are, and their motivations, would be valuable information to better understand the phenomenon.

References

Bacon, 2009. *The Art of Community*. O'Reilly, Sebastopol, CA. Available at: http://www.artofcommunityonline.org/downloads/jonobacon-theartofcommunity-1ed.pdf, accessed 23 June 2013.

Bonaccorsi and Rossi, 2006. Comparing motivations of individual programmers and firms to take part in the Open Source movement. From communities to business. *Knowledge, Technology, & Policy*. Winter (2006), Vol. 18, issue 4. pp 40-64.

Ciffolilli, 2004. The economics of open source hijacking and the declining quality of digital information resources: A case for copyleft. *First Monday*, Vol. 9, Number 9, September. Available at: http://firstmonday.org/ojs/index.php/fm/article/view/1173/1093

Dahlander and Magnusson, 2005. Relationships between open source software companies and communities: Observations from Nordic firms. *Research policy* 34 (2005) 481-493.

Dahlander and Magnusson, 2008. How do firms make use of open source communities? *Long range planning* 41 (2008) 629-649.

Daffara, 2011. Open Source License Selection in Relation to Business Models. *Open Source Business Resource*. (February 2011).

Fogel, 2009. *Producing open source software: how to run a successful free software project.* O'Reilly, Sebastopol, CA.

Gamalielsson and Lundell, 2012. Long-Term Sustainability of Open Source Software Communities beyond a Fork: A Case Study of LibreOffice. Hammouda et al. (Eds.): OSS 2012, IFIP AICT 378, pp. 29–47, 2012.

Grad, 2002. A Personal Recollection: IBM's Unbundling of Software and Services, *IEEE Annals of the History of Computing*, Vol. 24, No. 1 (Jan–Mar 2002), pp. 64–71.

Haefliger, von Krogh, and Spaeth, 2008. Code Reuse in Open Source Software. *Management Science,* Vol. 54, No. 1, pp. 180-193.

IBM Archives: 1969, available at: http://www-03.ibm.com/ibm/history/history/year_1969.html, accessed 20 June 2013

Krueger, 1992. Software Reuse. *ACM Computing Surveys*, Vol. 24, No. 2, pp. 131-183.

Lehman, 1980. On Understanding Laws, Evolution, and Conservation in the Large-Program Life Cycle. *Journal of Systems and Software* 1: 213–221.

Lerner and Tirole, 2002. Some Simple Economics of Open Source. *Journal of Industrial Economics*, 52 (June 2002) 197-234.

Levy, 2010. *Hackers: heroes of the computer revolution*. O'Reilly, Sebastopol, CA.

Markus, 2007. The Governance of Free/Open Source Software Projects: Monolithic, Multidimensional, or Configurational?" *Journal of Management and Governance*, 11 (2), 151-163.

Meeker, 2008. *The Open Source Alternative*. Wiley, Hoboken, New Jersey.

Naur and Randell, 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.

Nyman and Mikkonen, 2011. To Fork or Not to Fork: Fork Motivations in SourceForge Projects. Open Source Systems: Grounding Research, IFIP Advances in Information and Communication Technology Volume 365, 2011, pp 259-268.

Nyman, Mikkonen, Lindman, and Fougère, 2011. Forking: the Invisible Hand of Sustainability in Open Source Software. Proceedings of SOS 2011, available at: http://tutopen.cs.tut.fi/sos11/papers/SOS11_proceedings.pdf#page=9

Nyman, Mikkonen, Lindman, and Fougère, 2012. Perspectives on Code Forking and Sustainability in Open Source Software. *Open Source Systems: Long-Term Sustainability*, 274-279.

Nyman and Lindman, 2013. Code Forking, Governance, and Sustainability in Open Source Software. *Technology Innovation Management Review*, January 2013, pp 7-12.

O'Mahoney, 2007. The governance of open source initiatives: What does it mean to be community managed? *Journal of Management and Governance*, 11 (2), 139-150.

Phipps, 2010. Open core is bad for you. Computerworld, June. Available at http://blogs.computerworlduk.com/simon-says/2010/06/open-core-is-bad-for-you/index.htm, accessed 18 June, 2013.

Raymond, 1999*a*. The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary. O'Reilly, Sebastopol, CA. Available online at: http://www.catb.org/esr/writings/homesteading/cathedral-bazaar/

Raymond, 1999*b*. Homesteading the Noosphere. Available at http://catb.org/~esr/writings/homesteading/homesteading/, accessed 20 June 2013.

Raymond, 2001. How To Become A Hacker. Available at: http://www.catb.org/esr/faqs/hacker-howto.html, retrieved 20 June 2013.

Robles and González-Barahona, 2012. A Comprehensive Study of Software Forks: Dates, Reasons and Outcomes. Open Source Systems: Long-Term Sustainability, IFIP Advances in Information and Communication Technology Volume 378, 2012, pp 1-14.

Stallman, 1999. The GNU Operating System and the Free Software Movement. In: DiBona, Ockman and Stone (Eds.), *Open Sources: Voices from the Open Source Revolution*. O'Reilly. Available online at: http://oreilly.com/openbook/opensources/book/, also available at: http://www.gnu.org/gnu/thegnuproject.html, accessed 23 May, 2013

West, 2003. How open is open enough? Melding proprietary and open source platform strategies. *Research Policy* 32 (7), 1259-1285.

West and O'Mahoney, 2005. Contrasting Community Building in Sponsored and Community Founded Open Source Projects," Proceedings of the 38th Annual Hawai'i International Conference on System Sciences, Waikoloa, Hawaii, p. 196c.

West and O'Mahoney, 2008. The Role of Participation Architecture in Growing Sponsored Open Source Communities. *Industries & Innovation*, 15, 2 (April 2008): 145-168.

Wheeler, 2009. F/LOSS is Commercial Software, *Technology Innovation Management Review*, February 2009.

Yin, 2009. *Case Study Research (Fourth Edition)*. Sage, Thousand Oaks, CA.

PAPER 4

Hackers on Forking

Nyman (2014) Hackers on Forking. *Proceedings of the 10th International Symposium on Open Collaboration* (OpenSym 2014). ACM, 2014. ISBN: 978-1-4503-3016-9, doi>10.1145/2641580.2641590.

Hackers on Forking

Linus Nyman
Hanken School of Economics
Arkadiankatu 22
00100 Helsinki
+358 9 431 331
linus.nyman(at)hanken.fi

ABSTRACT

All open source licenses allow the copying of an existing body of code for use as the basis of a separate development project. This practice is commonly known as forking the code. This paper presents the results of a study in which 11 programmers were interviewed about their opinions on the right to fork and the impact of forking on open source software development.

The results show that there is a general consensus among programmers' views regarding both the favourable and unfavourable aspects that stem from the right to fork. Interestingly, while all programmers noted potential downsides to the right to fork, it was seen by all as an integral component of open source software, and a right that must not be infringed regardless of circumstance or outcome.

Categories and Subject Descriptors

K.6.3 [**Software management**]: *software development, software maintenance*

Keywords

Open source, free software, fork, code forking, code reuse

1. INTRODUCTION

All open source licenses permit the copying of an existing body of code to use as the basis for a separate development project. This practice is commonly known as code forking. The right to fork is a topic that has long been debated among practitioners and researchers [3] and the debate regarding the advantages and disadvantages of forking began even before the term "fork" came into popular use[1]. The practice of forking has been both complimented and condemned, described as a "Good Thing" [9], a "Bad Thing"[2], open source software's "cardinal sin" [25], and a vital part of open source software (e.g. [23]).

Code forking appears to be both loved and feared – the good, the bad, and the ugly all in one. Despite the conflicting and often strong views on forking, it is a topic that has seen little research. Indeed, Gamalielsson and Lundell [4] note that "there is clearly a need for increased knowledge about how OSS communities are affected by a fork."

Forking has the potential to affect open source software development and governance on many levels, including the software, community, and business ecosystem levels [14]. How programmer communities are affected by a fork will depend in no small part on how they view the fork. Thus, there would seem to be merit in gaining a better overall picture of how programmers[3] view forking. What do they view as the merits and drawbacks of forking? What makes a fork acceptable or unacceptable to the community? Given the strong opinions forking elicits, are there any moral obligations or codes of conduct that must be followed in order to gain community acceptance or favour when beginning a fork?

The aim of the current study was to better understand developers' views on forking, with an emphasis on what programmers consider the benefits and drawbacks of the right to fork, and what kinds of forks they consider favourable and unfavourable. How a fork is viewed by the wider open source community will directly affect its possibility of acquiring developers and attaining sustainability.

The paper is structured as follows: first, we cover previous discussions of, and research into, forking. Then we introduce the research questions and methods. After this the results of the study are presented. The paper concludes with a discussion of the implications of this research along with its limitations and some avenues for future research.

2. BACKGROUND

Code forking can be seen as healthy for the overall software ecosystem, offering a kind of 'survival of the fittest' where only the best code will survive [1, 23]. However, a fork is likely to be

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the Owner/Author.

Copyright is held by the owner/author(s).

OpenSym '14 , Aug 27-29 2014, Berlin, Germany
ACM 978-1-4503-3016-9/14/08.
http://dx.doi.org/10.1145/2641580.2641590

[1] See Nelson's (1993) discussion "Shattering – good or bad?": https://groups.google.com/forum/#!topic/gnu.misc.discuss/G7pwuPhnbEM

[2] See Jargon File: http://www.catb.org/jargon/html/F/forked.html

[3] The terms "programmer", "developer", and "hacker" are used interchangeably in this paper. The term hacker is thus not meant to imply any negative connotations (relating to exploiting weaknesses in a computer system or network, or indeed any other negative connotations).

harmful for the individual project affected [1, 23]. The harm is not due to the mere existence of a fork, but rather the loss of users, duplication of effort among programmers (e.g. [1, 11, 18]), and compatibility issues between forks ([21], see also [23]). A further disincentive is the challenge of attracting a sufficient pool of developers to aid in the further development of a fork, particularly if the original already has a significant community [23]. Programs under a copyleft license have further disincentives to forking: if a program is forked and improved, developers working on the original program may be able to incorporate those improvements, thus creating a disincentive to forking from both a development viewpoint (e.g. [9]), as well as a business-oriented viewpoint (e.g. [24]).

Thus there is a strong social pressure to avoid forking (e.g. [8, 18, 25]). Previous research has categorized the avoidance of forking a project as a community norm among open source software developers [22], and it has been noted that developers require a compelling reason to switch to a competing project [24].

Using a more restrictive definition of forking, it has previously been noted to be avoided. However, the importance of the right to fork is still underlined by many. The potential to fork a program has been called "the indispensable ingredient that binds developers together" [1], as 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. Thus, the potential to fork is also a significant element in the governance of open source programs (e.g. [10, 23, 25]; see also [3, 4]), as no one has a "magical hold" [1] over any project since anyone can fork any project at any time. The right to fork can furthermore be seen as "a vital safety valve" [9] in case existing developers stop working on the project, or decide to stand in the way of progress. Similarly, regarding corporate actions and acquisitions, should a company try to "hijack" [6] the source code by making it proprietary, or otherwise work against the best interests of the program and its community, forking provides the community a way to protect their own vision of the project [10, 12, 14, 16, 17].

While forking is discussed in a number of textbooks and academic articles, there are few studies whose focus has been code forking itself. A small group of papers examine forking on a conceptual level, focusing on its relevance to the sustainability and governance of open source software development [14, 16, 17]. This author and Mikkonen [15] analysed the developer-stated motivations of forks hosted on the SourceForge repository, noting that the majority of forks appear to be motivated by pragmatic concerns, and to be of a non-competitive nature. Indeed, an in-depth analysis of 220 forks showed that forking: occurs in every software domain; has become more frequent in recent years; very few forks merge with the original project; and that many forks are of a "friendly" rather than competitive nature [20]. Competitive forks do exist, and may be motivated by a concern for the future wellbeing of the codebase under its present leadership [12].

Through the analysis of the 9 most forked JavaScript programs hosted on GitHub, Fung, Aurum, and Tang [2] introduced the concept of social forking. Social forking includes community, software, as well as other artefacts taken from the original project in the creation of the new. Gamalielsson and Lundell [3 and 4] analysed the LibreOffice fork from OpenOffice, finding that it is possible for a fork to prosper and achieve sustainability over time, and for a software community to outlive open source software projects.

While a fork has traditionally been defined as an instance in which a group of developers split into competing groups that go on to develop the program in different directions, Nyman [13] notes that the definition of a fork has changed in recent years, with several different interpretations in existence, but that still maintain the common thread of taking an existing program and modifying it. Furthermore, subsequent with the rise of popularity of hosting and revision control services with a broader use of the term[4], 'fork' has come to be used in an increasingly broad fashion. Indeed, in addition to being used by some as a synonym with branch, fork may also now be seen used among developers to indicate reusing existing code in the creation of a program that may target a significantly different user group than the original, developed by hackers with no affiliation with the original project.

3. RESEARCH QUESTIONS

The research questions focus on two broad areas of interest: programmers' views on the concept of forking in general and on the practice of forking. These two broad areas of interest can be divided into three topics:

1. What do programmers consider to be the benefits and drawback related to the right to fork? (Research questions 1 and 2)
2. What constitutes an acceptable and unacceptable fork? (Research questions 3 and 4)
3. How should forking be done, i.e., what, if any, are the moral obligations and proper codes of conduct in regards to forking? (Research questions 5 and 6)

3.1. What are the Benefits of the Right to Fork?

Existing literature has noted benefits like keeping a community more tightly knit, faster progress due to less divergent efforts, and the ability for the community to take control of their own fork of a project, should such a need arise. Are there any benefits developers consider significant that have escaped mention in previous literature?

3.2. What are the Drawbacks of the Right to Fork?

Generally, there is more focus on the negative aspects of forking, including issues such as redundancy of effort and confusion among users regarding which version to use. What do programmers see as the main drawbacks of the right to fork? Are there any perceived drawbacks not noted in previous literature?

3.3. When is it Acceptable to Fork?

Open source licenses guarantee the right to fork. Perhaps not surprising given its potential negative impact, several authors have noted the social pressure *not* to fork. However, previous research (e.g. [12, 15, 20]) has identified at least some circumstances in which a fork is considered acceptable. What kinds of circumstances are required for this social pressure not to fork to be outweighed by the perceived benefits *to* fork? Given that there are situations in which a community accepts or calls for a fork, can we identify any common characteristics of these situations?

[4] See GitHub: https://help.github.com/articles/fork-a-repo.

3.4. When is it Not Acceptable to Fork?

While several challenging aspects of forking have been noted, can one more precisely document what the conditions are under which the social pressure not to fork would be particularly strong? Are there any kinds of forks that are not considered acceptable under any circumstances?

3.5. Are There any Moral Obligations When Forking?

The large amount of forks documented in recent studies appears to contradict the notion of an open source norm not to fork. However, given the notion of the hacker ethic (e.g. [5, 7]; see also [18, 19]), it seems plausible that some norms surrounding a practice as pivotal as forking would exist, perhaps in the form of some unwritten yet commonly held rules or guidelines as to what obligations a programmer has, should they decide to fork a program. Do such guidelines exist? And if so, what are they?

3.6. What is the Proper Code of Conduct When Forking a Program?

Like the previous question, this question addresses a situation in which one has already decided to fork a program. How should one then proceed? Are there any undocumented rules or common practices regarding such a situation? If programmers could decide how other programmers went about forking, what kind of actions or behaviour would they hope for or expect?

4. RESEARCH APPROACH

4.1. Data and Data Gathering

Before beginning the data acquisition, it was important to become familiar with the writings and thoughts on forking found in academic literature, practitioner texts, and other sources of knowledge on forking (developer blogs, online news sites, etc.). Based on this knowledge, an interview guide consisting of open-ended questions was compiled. During the winter of 2012, a pilot study was conducted consisting of 4 semi-structured interviews. The interviewees all fulfilled the criteria that they all either were, or had previously been involved in open source software programming. During these interviews, the interview guide was tested and modified. The interviews were conducted in person, and were recorded and transcribed.

Eight additional interviews were conducted between 2013 and 2014. The interviewees were chosen using snowballing, with the criteria that: 1) they needed to have been involved in at least one open source software project; and 2) they needed to currently make a living from programming, be studying programming, or have at some point previously had made their living from programming. The interviews were conducted in Finnish, Swedish, and English, depending on the preference of the interviewee. Roughly half of the interviews were conducted face-to-face, the rest using Skype. The interviews were semi-structured, allowing for additional questions as needed, as well as giving the respondents the opportunity to express all thoughts and insights they felt relevant. All of the interviews were recorded and transcribed to aid in the data analysis. The shortest interview was 13 minutes and the longest was 41 minutes.

The extended time during which the interviews were conducted allowed for long periods of reflection of the results. The focus of the analysis was primarily to identify all unique opinions brought up in regard to each research question. The secondary goal was to examine these unique answers to see if they lent themselves to grouping. Where it seemed helpful, the answers were grouped into overarching themes. Upon completion of the first version of the article, it was sent to a subset of interviewees. This was done both to allow the interviewees to check their quotes, and to give them a chance to voice their opinions on the points brought up by other programmers.

A majority of the programmers were from Finland, the rest were from Italy, Russia, and the Ukraine. The interviewees ranged in age from their early twenties to their early fifties. One interviewee was female, the rest were male. The programmer with the least experience was involved in his first open source project at the time of interview. The most experienced programmers had several decades of experience, and included a developer who had worked primarily on the Linux kernel, a developer employed by the Apache Foundation, a developer who was a founding member of both MySQL and MariaDB, and a Linux pioneer who has furthermore both authored as well as contributed to numerous other open source programs.

4.2. Unifying the Definitions of "Fork"

Due to the potential for differences among interviewees regarding their views of what, exactly, the term fork entails [13], all interviewees were first asked a number of questions regarding their definition of the term fork and what it entailed. This was to ensure that all interviewees would be discussing a similar phenomena. Of all of the interviews, one programmer had a definition of fork that was unique enough that the answers given during that interview are not included in this study. Thus, the data for the study originally consisted of 12 interviews, of which 11 were included in the study. For the purposes of this paper, the definition of fork used is a somewhat broader version of Robles and González-Barahona's definition [20]. In this study a fork is when one or more developers start an independent line of development based on the source code of an existing project. Projects that will or may attempt to submit their changes back to the original are also included in the definition used herein.

5. FINDINGS

5.1. What are the Benefits of the Right to Fork?

When asked about the benefits of the right to fork, programmers commonly shared that forking is something more than a mere right guaranteed by open source. Instead, they saw it as an integral part of the very concept of open source software. As one programmer noted, "*I don't know if one really could see open source as open source without the right to fork*." Another programmer stated that if developers can't fork, "*then first of all, it isn't free software, it isn't open source anymore."* One programmer discussed the topic in greater historical depth:

"At its root it is a large philosophical question: the whole idea of open source takes as its starting point that you can take source code and do whatever you want with it, entirely regardless of what the original developer or current community wants to do with the code. And in that sense it is one of the pillars of open source, this possibility to fork."

Thus, the first point to note is that the right to fork, regardless of any specific benefits it offers, is commonly seen as a cornerstone of open source.

Moving on to specific benefits brought up regarding the right to fork, one developer, with several decades experience in open source software development, noted that forking *"is what makes it*

possible to take a project in different directions or [to ensure] that the product doesn't die." This developer also brought up a benefit for companies: that they can use the right to fork to solve problems through forking a program and then customizing it to meet their specific needs. If broadly interpreted, the three main points of taking a program in a different direction, keeping a program alive, and benefits for corporations – or, indeed, non-profits or any other interested party – cover the entire list of benefits stated by all interviewees combined. However, to better understand and convey the thoughts of the participants, it is useful to divide the stated benefits into further categories. The benefits brought up by the interviewees can thus be grouped into three overarching categories: preservation, experimentation, and customization.

Preservation

A rather clear-cut benefit is that forking makes it possible to continue a project that has been abandoned[5]. As a developer from the Ukraine noted, *"Sometimes the original developers will just give up, and if there's no way to just continue – for example no one has access to that repository – then you can just fork the project and start a new [one] with a new set of developers."*

Experimentation

This category represents forking as the ability to try new things; either with the intention of merging successful changes back with the original, or simply as a personal exercise. A clear benefit of experimenting on a personal fork is that one cannot cause any unintentional harm to the original project. One developer noted that, if one considers what is often called branches to be forks, then *"As a part of the workflow [a fork] is a vital part. That you can work in peace on your own branch of the program until your feature is ready to be shown to the others."*

A fork may also be necessary in case the project leaders do not share one's views regarding the feasibility of the proposed changes. An experimental fork can then be used to create a proof of concept. As an Italian developer noted:

"If you really don't like something, or if you think that something else has to be done, you always have the chance to try it yourself, even if you are alone […] and everyone else is against you – why not? You can always try it out and then if it happens that you were right, you can always go back in. So I think it gives a lot of freedom to try out something."

Customization (including forking as a "safety valve")

One central benefit is the possibility to customize a piece of software to better suit one's needs, i.e., to take a project in a new direction. As one developer noted, without the right to fork, *"whoever started the project and defined its goals gets to specify how and when it is used. And in my opinion that is kind of a bit against the whole idea of open source."* Different motivations for customizing a project were noted, among them personal and corporate needs, and disagreements among developers – the kinds of situations for which the right to fork has been called a "safety valve" [9].

[5] Indeed, studies by both Nyman and Mikkonen [15], and Robles and González-Barahona [20] found several cases of such forks.

Personal and corporate needs were of a "scratch an itch" –type. Something close to what was needed already existed, but some modifications were required for it to meet the needs of the developer, or the company they were working for. In discussing such cases, many programmers noted that they commonly involved small tweaks or additions to existing programs. However, a large-scale, continuing development effort based on an existing program was also a possibility, though commonly of a kind covered in the third subcategory, discussed next.

The ability to use the right to fork as safety valve was noted to be beneficial in several different circumstances. One possibility is that a project, while not abandoned, may have stagnated significantly enough that a fork can be perceived as beneficial. As one developer noted:

"We have noticed several times that programs – when they reach a certain age… there comes a certain inertia, and forking may have offered the possibility for a new start. You can leave out the old ballasts, and in that sense many projects that have started as forks may in the end have turned out to work much better [than the original]."

Furthermore, even with actively managed projects, a fork can be beneficial. One example that was frequently brought up was if there was significant disagreement among developers. Cases in which developers disagree on the project's direction, where the difference in question was one of technical aspects relating to the program, were commonly seen as an instance in which it is positive that one can fork, with many explicitly noting that a fork in such a situation may be an entirely favourable thing.

"If the community gets into arguments, or disagree on what direction to take the project, then I think it's fine that they go in different directions [i.e. fork the program]. It might be in the software's best interest; that more is accomplished than [if] they just focus on arguing."

In addition to internal disputes that could lead to, or even necessitate, a fork, programmers also noted its relevance in cases in which an original developer is perceived to mismanage a project. Two example quotes follow.

"First and foremost, [the right to fork] reduces tyranny. If I have created a program and am king and ruler there, then the fact that everyone can fork it means there's a limit to what I can decide about. […] A second aspect is that it gives power to those who didn't start the project, which is almost the same as the first [point] but slightly different. It means you aren't bound by what someone else wants, which is a pretty central concept in open source [and] free software."

"I think there are a lot of good examples where a forge has been forked and the forked version has just become a lot better than the original. Like… if a project is maintained by somebody that is very single-minded and doesn't really take into account the opinions of others, but the code that they have produced is, like, of high value, I think a fork could be a very good asset there."

5.2. What are the Drawbacks of the Right to Fork?

Common themes brought up were the negative effects a fork could have on users, developers, and the project itself; commonly, the reasoning behind all three groups of answers was related to problems of fragmentation.

Confusion among users

Developers reported that one drawback to forking was the confusion and extra work required to keep abreast of new forks that users experience. In this way, the programmers were both discussing external users and themselves as users of a program.

"You create problems for the users regarding what they should use. For users, a fork is never a good thing. Unless it is done to keep the project alive."

Similarly, even for developers, who one could assume are more informed about software in general, multiple forks means extra effort when researching a project of potential interest. One developer described it thus:

"Well the main thing is of course this fragmentation, it's kind of obvious. I mean if you look at GitHub, some of the projects, you start searching for some project and then you find ten instances of that project and then you would like [wonder], which one do you pick? Then you have to study all ten and see which one is well maintained. Sometimes you want to pick the original because it might have a bigger following. Sometimes the original might have been abandoned, so that would be bad choice, so you actually should pick one of the more recent forks. And it could be that sometimes it was forked precisely because the original was abandoned and no one could contribute to the original project."

Duplicated work among developers

Developers also noted as a drawback the dividing of development efforts resulting from multiple versions of a similar project developed in tandem; something many noted to be the most significant of the drawbacks. As one programmer noted, *"I think the only really big drawback from forking is that then people are contributing to different projects."*

Similarly, discussing situations in which the fork occurs from within a project, one developer noted the dividing of resources to be a drawback, and added that a further *"perhaps even more important"* drawback was arguments between the two groups.

"You can have several years of arguments between the projects, and that can lead to the developers becoming unmotivated, and the users giving up, and the press writing unpleasant things, and things like that. That doesn't usually lead to anything particularly pleasant anymore, and everyone suffers."

Compatibility concerns

A further point brought up was the impact of a fork regarding compatibility. Even in cases where compatibility and co-operation are strived for, over time it was said to be hard to accomplish, with incompatibility, and the resulting duplication of efforts, an unfortunate result. Furthermore, the challenge for third-party software was brought up: cases where a programmer, or company, wants to maintain versions of their program that are compatible with both the original and the fork. The further the forks stray from one another, the more time and resources would be required to remain compatible with both.

Threatening to fork as a negotiation tactic

As forking may potentially have a strong negative impact on a development project (e.g. through a loss of users and duplication of development work), it was noted that the threat of forking can be used as a negotiation tactic:

"Negative part [of forking] is that it's a strain, in a way, to the project. And in some cases it can also be used as a weapon: 'If you don't do what I want…'. And that's the point, especially in companies. The company can always say that 'Yeah, it's open source – do whatever you want. [But] if you don't do what I want, I will fork and take all my force with me'."

Financial challenge

Few developers brought up financial challenges due to the right to fork. This is perhaps not surprising, as the challenge of developing software that is freely available is found in all open source projects, and not exclusively related to forking. As one developer described this challenge: *"The drawback is that it is hard to generate an income in a situation where you can take a copy. I try to sell you an apple; you take a copy of it and eat [the copy]. Well… then you have to kind of think: where does the income come from, then?"*

5.3. When is it Acceptable to Fork?

One of the developers was of the opinion that it was acceptable to fork *"if the original developers weren't doing their job"*, offering by way of example a case in which an open source and proprietary program were being developed in tandem, and then the developers had ceased development of the open source version, focusing their efforts entirely on the proprietary version instead. The programmer commented on such a fork, noting: *"I think it's entirely OK to do that. It keeps the software as free software."*

All of the other programmers interviewed were of the opinion that it is always acceptable to fork. This was surprising considering that they were asked not when it was allowed, or legal, to fork, but rather when it was *acceptable t*o do so. While they could all give examples of problematic aspects related to forking, as well as examples of forks that they felt were unnecessary or would result in problems for the parties involved, they all nonetheless noted that a fork was always acceptable. One Italian developer put it well, *"it's always acceptable. Whether I like it or not."* Another developer raised the question: *"Why would you make it open source if you were against forking?"*

In order to clarify their answers, follow-up questions were asked regarding the kinds of situations under which the developers would consider forking a program they were involved in, and what kinds of forks they would hope could be avoided. This was done to better understand their opinions on what kinds of motivations they considered valid, or even positive, reasons for forking. A basic notion was that there should be a clear point, or reason, to the fork.

"If there is a clear point why people should be contributing to multiple projects then I think it's the only valid reason for forking a project. If there's no real reason why it should diverge from the original, I don't think there is really much point or benefit to be had from forking."

While the developers shared a number of clear justifications, the common thread among them was that they were all situations in which the developers felt that a fork would lead to a better program. As one developer put it: *"I always see the end user as the point, so… If you see that [by forking] we have the chance to do something good, you should."*

Another common notion was that any fork that is significantly different from the original is a sensible motivation.

"If you believe that the new version will get both users and developers, then of course it's sensible [to fork]. […] The idea is, the point of the fork is, that through it you are actually doing something new, and that it has some meaningful difference."

A situation in which a developer is hindered from making changes they consider meaningful due to divergent views of project goals, or an unresponsive leader, were also commonly brought up. Example quotes from three separate developers follow.

"If the project is for example big – they have a lot of users – they have their own priorities in terms of what to change there. So if you send in your changes there and they wouldn't accept these changes, then you just have to have your own kind of version; your own fork of the project, where you have your own changes."

"Well, if the program doesn't solve the original problem and you can't convince the group working on it that it should be taken in that direction, then you don't have any other choice than to either fork or to choose another program. And in such a situation a fork could be useful."

"If there's a significant difference, or differences in views on in which direction the project should be going, I think that's a good reason why one should be forking a project. Because if the project maintainer doesn't take into account a majority of the projects or the users' of the project's views on in which direction the project should be going, I think it's a good case where somebody should fork it and start directing the project in a way where others could be benefiting more from it."

As one Finnish developer sought to highlight, forking is always acceptable but not always beneficial or practical.

"In the short run forking might be easy – you accomplish a lot in a short time. But in the long run, you lose all the benefits of participating in the activities of a larger community. So in that sense it isn't so much a moral as a practical question."

5.4. When is it Not Acceptable to Fork?

Naturally, the answers to this question closely mirrored those of the previous question. All but one of the programmers noted that it is always acceptable to fork. However, as one Finnish programmer noted, *"That one has the right to fork doesn't exclude there from being situations in which a fork would be problematic"*. Through follow-up questions I attempted to determine what makes a fork undesirable or problematic.

Again, the views closely tracked those in the previous question. Where any fork that is in the best interest of the program was considered a valid motivation, any fork that was not in the best interest of the code was viewed unfavourably. The examples given can be broadly categorized under three headings: in breach of an agreement, personal differences, and malicious forks.

In breach of an agreement

An example of a type of fork one programmer noted to be unfavourably viewed had to do with a situation in which a group of actors would have agreed not to fork. The programmer noted that it would *"perhaps not be that acceptable […] if one had – through either cooperation, common standardization, or some other common endeavour – decided that some specific open source program should be governed by a non-profit organization, in the interest of keeping it neutral."*

A somewhat related answer by another programmer was regarding not when it was acceptable to fork, but rather what kind of fork a license allows, and that one shouldn't breach license stipulations.

"What isn't ethical is if the licensor decides that this [software] must be kept free, [and] it's modifications must be kept free, and then you fork it and make it closed."

Personal differences

Most developers stated that they viewed forks that were motivated by personal conflict unfavourably. Two examples from the interviews follow.

"In my opinion forks that are based on personal dynamics aren't really that sensible; they kind of weaken the community."

"Many times [a fork is] about people colliding and not features colliding, and that shouldn't be, but… we are human."

Malicious forks

A number of developers brought up the theoretical problem of starting a fork with no other agenda but to compete with the original. That is to say, a fork that did not seek to modify or improve the program in any way, but only sought to create competition. However, most developers also noted that they couldn't think of any examples of when such a fork had actually happened.

"Well of course it's bad if, like, the original project would be good and then someone just goes and forks it to compete with it. But, to be honest, I can't recall seeing that sort of thing happening, where it wouldn't have been justified to fork and, like, diverge from the master. So I mean if that would actually happen I think that would be a bad case but as I personally haven't seen a case like that I don't know if it even exists. I mean do people fork just to compete if the original is a good project? I don't know…"

"Maybe you could come up with some nasty reasons, like someone wants to sabotage the project and just creates extra forks for confusion, or just out of spite, and just draw away some parts of the community towards another project, but… I don't know if that happens. […] It basically depends on the purpose: if the purpose is OK then forking is OK."

Another example quote emphasizes the imperative among developers to state that forks, while potentially problematic, are always allowed:

"If there is some form of ill will [towards the original] at the base of the fork then I would say it starts being problematic. But even then I think that it, from a legal perspective, is clear that they have every right to [fork] it."

One developer discussed such malicious forks as "sabotage attempts", offering by way of example that he had seen a few instances of single individuals forking a popular operating system and attempting to start an argument among the community which would cause the developers to spend their time arguing rather than coding. However, he noted that such attempts *"maybe succeed for a week or so, but then [the developers] go back to doing what they do."* The developer also discussed what he considered a more serious form of sabotage fork, one that would most likely be linked to financial interests and set in motion by a commercial company with a competing product.

"The first thing is that you achieve a situation in which everyone knows that there is a fork, and you stir up an argument so that

everyone knows that there is something… some problem with the original project, even if there isn't. This is a tactic I expect some large company at some point will use."

One developer's answer offered insight into why so few forks are viewed in a negative light despite the potential downsides to forking: *"Most of the forks – from the outside, it looks like: 'Why? Why don't they all join together and be friends?' From the outside. But when you read up about the reasons and the discussion on the forums, it becomes clear. […] I can't remember any example when I really looked into [a fork] and it would be just… completely senseless."*

5.5. Are There any Moral Obligations When Forking?

Contrary to expected, in all cases but one these interviews did not yield explicit lists of issues programmers considered significant to either remember or consider when forking. In fact, one developer began his answer by noting that he needed some time to consider the question, as he had never before given it any thought. In light of the insights gained from previous answers, particularly that of programmers' strong feelings of the right to fork as integral to open source, it is perhaps not surprising that no such unwritten yet widely adopted "moral obligations rulebook" exists[6].

However, there was one issue that all hackers brought up: crediting the original contributors. This seemed a self-evident point. Indeed, the aforementioned programmer who noted that he hadn't pondered what moral obligations one may have, noted that one should credit previous contributors, but added that the point was "*kind of obvious*." Some connected the obligation to give credit with the stipulations of the license, others with a moral obligation to do so. Example answers included:

"Just what the license says. They need to credit the author if the license requires it but I think that's it."

"What is important is to give credit to the original project, and meticulously document what has come from whom – who has contributed these different parts. To give credit where credit is due."

As mentioned previously, there was one notable exception. This was from a developer who began his answer by noting that the answer *"depends entirely on how one views the freedom of the program. I am what you would call a free software person rather than an open source person."* The programmer then went on to note that:

"From a free software perspective, one has the moral responsibility to see to it that those who use the program aren't hurt. So for instance that one maintains the fork as free software, and sees to it that, if possible, one keeps both versions compatible with one another […] so that, if possible, one cooperates with the original project. […] If one can do these kinds of things, then I think one has the moral imperative to do them. But the important [thing] is to make sure that the users – not the developers, the users – don't suffer."

5.6. Is There a Proper Code of Conduct if One Wants to Fork a Program?

Only one of the developers, the one who noted he was a free software person rather than an open source person, noted a number of particular actions that he felt should be taken regarding the actual process of starting a fork. He began by stating: *"One should be honest and clear about why one is doing a fork. That means that the big Internet – they won't have to guess, and find scandals where they don't exist."* He continued by noting that the fork should be done *"in such a way that the original does not suffer unnecessarily."* This included practical issues aimed at avoiding confusion and problems among developers and users, with examples being setting up a new mailing list, website, server to host the code, and choosing a new name for the project. *"By and large,"* he concluded, *"one should behave like a human being."*

While other hackers didn't offer views regarding a code of conduct when forking, something that was brought up by several programmers was what to do *before* one decides to fork: to ensure that a fork is absolutely necessary. This point was brought up by both developers who had forked programs as well as those who had not. Examples from the interviews follow.

"Well, usually you start by seeing if you can avoid the fork: talk to the developers and say 'we would like to do this'. And then, if you can't agree, then you say 'OK, we'll fork'."

"My feeling would be that, if the project is active and has a community, it makes sense to talk to them to see if your version would really be different. Because if you don't do that, then it might… there are these drawbacks – that it creates this, kind of, incompatibility and, it kind of makes the whole landscape more complex. So you have to make sure that it's worth it. So it makes sense to talk to the original developers and see that, 'Can't we cooperate?'"

A further point that was brought up by several programmers was the importance of maintaining some form of communication with the original so that one can, as one interviewee put it, *"Share the parts of the program that can be shared"*. Indeed, one programmer, in discussing a fork that had been made from a project that he had spent the better part of a decade working on, noted that, for many years they were able to share code. In discussing the fork, he only bemoaned a decision made by the fork project's leader many years after originally forking, to resolve a technical issue in such a way that they would no longer be able to share code with one another.

Regarding communication before a fork, one developer offered a significantly different viewpoint, light-heartedly noting that *"I would personally dislike it if every person that would want to fork some project [of mine] would be contacting me. Just a lot of spam. I would have to set up some email account that would auto-reply 'yes' and… there wouldn't be much value to be had from that"*.

While this position may seem to contradict earlier points, it seems likely that the approach programmers' would consider favourable could be relative to the proximity of the goals of the original and the fork: the less planned overlap there is between the two, the less important it is to initiate and maintain significant contact with one another.

[6] There are, however, indications that even among those without an explicit list, an implicit list may still exist. Further research would be needed to confirm this, but it could include themes like: thou shalt not infringe on the right to fork; agreements shall be binding; personal differences shall be moot; and, don't start a fork out of spite.

6. CONCLUSIONS: SUMMARY OF RESULTS

What is good/bad about the right to fork?

First and foremost, developers consider the right to fork to be integral to the very concept of open source software. Beyond the general view of the right to fork as an important open source cornerstone, the right to fork is considered positive for several reasons. Among them are the possibilities it offers to: preserve abandoned programs; to experiment with and customize existing programs; and to both minimize tyranny and resolve disputes regarding development efforts through giving the involved parties the opportunity to develop their own versions of the contested program.

While considered vital, the right to fork can still be problematic. Examples of related concerns are confusion among users, and extra work among developers, including both duplication of efforts and increased work if attempting to maintain compatibility. Furthermore, particularly if the fork is from within the project, the resulting decrease in developers is a concern. The right to fork also has a significant effect on the ability to generate income for a project.

What kinds of forks are viewed favourably/unfavourably?

The kinds of forks viewed favourably are those where there is a reason to believe the fork will lead to a better program. Examples given of such scenarios included: that the program better serves (all or some) users, that the original is no longer being developed or development has stagnated, and that disagreements among developers significantly hamper or halt its development.

The kinds of forks that hackers would prefer to be avoided are forks that are based on personal issues (e.g. rather than differences of opinion regarding which direction to take a development in), and forks that are not so significantly different that they couldn't have remained one single program. Purely malicious forks, created merely to sabotage or confuse, are also frowned upon as a concept, but such forks seem exceedingly rare.

How should one go about forking? Are there any moral obligations or is there an (unwritten) code of conduct?

The sole rule that seems to guide forking overall is that one must give credit to those that have worked on the program previously (something also stipulated by license). However, the value of being in contact with the original developers before one decides to fork, to see if the fork is in fact a necessity, was underlined by several developers. This may lead to avoiding the fork and thus keeping more developers working on the same project. In cases in which forking is nonetheless necessary, the projects may still be able to benefit from one another's efforts by sharing code.

7. DISCUSSION, LIMITATIONS, AND FUTURE WORK

While not all programmers gave answers that were identical in content, no programmer's answers were counter to views expressed by the others. No kinds of forks were viewed as positive by some yet negative by others; no issues about forking were viewed as positive by some and negative by others. Thus, there was a general consensus among all of the programmers interviewed.

The findings of this study largely support what can be found in practitioner literature and earlier studies. Gamalielsson and Lundell's [3, 4] findings, that a fork can be sustainable over time and that an open source software community can outlive a project, fit seamlessly with the interview answers gathered for this paper. Particularly in the event of a program's governing body being perceived as acting against the interests of the community at large, a fork not only seems likely, but it may generate more wide-spread support among developers than is enjoyed by the original development.

The data also offers further support for the findings of this author's earlier paper [12], that a fork is considered valid in cases where there is the possibility of a decrease in openness of a program, either through changes in licensing or through the community's hampered ability to contribute. The views expressed by the interviewees also corroborate earlier conclusions drawn by this author and Mikkonen [15], that programmers principally fork code in order to achieve pragmatic, development-related goals, rather than in order to compete with existing communities.

The interviewees' permitting stance overall towards any fork may seem to contradict existing literature about the social pressure to avoid forking. However, this discrepancy seems likely to find its explanation in the broad scope of extant interpretations of the term fork. In popular use the term has grown to encompass a greater breadth of actions than only competitive forks, yet existing literature commonly uses a much more narrow interpretation, that of an internal split among developers due to irreconcilable differences [13]. Thus, these findings do not so much contradict earlier statements regarding programmers' avoidance of forking as they do highlight the evolving interpretation of the meaning of the word fork. Illustrating this point, one of the developers interviewed, having read an early version of this paper, noted that on GitHub the number of times a project has been forked is used as one measure to rank its popularity. Furthermore, he noted that developers actively encourage others to fork their programs.

Very early on in conducting the interviews it became clear that more interviews would likely garner new ways of expressing similar points rather than new viewpoints. While I initially suspected that this similarity among answers may have something to do with a common "hacker ethic", based on the data gathered, the imperative to avoid a fork seems to be less an extrinsic community pressure and more an intrinsic sense of pragmatism shared by all developers.

A final interesting point to note is that the more strict views with regards to what is acceptable and not acceptable in regards to forking were offered by a developer with an affinity for free software, rather than open source software. Though the differences in views were not great, where they did exist the main emphasis of those differences, perhaps not surprisingly, was on the maintained freedom of the code.

7.1. Limitations and Future Work

There were no interview questions that focused specifically on what developers think about variations in open source licenses. Thus, no data was gathered specifically on forks from open source projects that are later made proprietary. Arguably, if developers had strong opinions about such forks, they would have voiced them even given the questions not specifically noting them, as was the case with one of the interviewees. Furthermore, several developers discussed the benefits for companies in that they can fork and modify existing programs. Nonetheless, future work could more specifically address developers' views on proprietary forks of open source software programs.

A further limitation of the study relates to the pool of interviewees and its limit in scope. It would be interesting to include more developers representing a wider distribution of nationalities. However, given the homogenous nature of the interviewees' answers despite them coming from four countries, it is uncertain that a modest increase in interview subjects or backgrounds would have had a significant effect on the outcome. Future work could broaden the sample, including a wider mix of countries and backgrounds. The similarity of the opinions among interviewees in this study makes it challenging to suggest where differences of opinion might exist, or be particularly pronounced; which is something that further study could also reveal. However, the data does seem to indicate that a deeper analysis of the potential differences of opinion between free software versus open source software enthusiasts might offer at least one further interesting avenue of research.

The current study does not attempt to classify opinions relative to their importance. A quantitative, survey-based analysis, based on the information gained here, could uncover information about such issues. Such a study could also shed light on what variables may correlate with certain opinions: do programmers of a certain age, background, or work-history have more similar views than those from other groups? What about programmers who consider themselves more aligned with either the term free software or open source software? While the data from this article suggests that identifying significant differences of opinion is not a certainty, it would be interesting if such differences were uncovered.

8. REFERENCES

1. Fogel, K. 2006. *Producing Open Source Software*. O'Reilly, Sebastopol, CA.
2. Fung, K.H., Aybuke, A., and Tang, D. 2012. Social Forking in Open Source Software: An Empirical Study. In *Proceedings of the CAiSE'12 Forum at the 24th International Conference on Advanced Information Systems Engineering (CAiSE)* (Gdańsk, Poland, June 28, 2012).
3. Gamalielsson, J. and Lundell, B. 2012. Long-Term Sustainability of Open Source Software Communities beyond a Fork: A Case Study of LibreOffice. In: *Proceedings of the 8th IFIP WG 2.13 International Conference*, *OSS 2012* (Hammamet, Tunisia, 2012).
4. Gamalielsson, J. and Lundell, B. 2014. Sustainability of Open Source software communities beyond a fork: How and why has the LibreOffice project evolved? *Journal of Systems and Software* 89: 128-145
5. Himanen, P. 2001. *The hacker ethic, and the spirit of the information age.* Random House, New York, NY.
6. Lerner, J. and Tirole, J. 2002. Some Simple Economics of Open Source. *The Journal of Industrial Economics*, Vol. 50, No. 2, pp. 197-234.
7. Levy, S. 1974. *Hackers*. O'Reilly, Sebastopol, CA.
8. Meeker, H. 2008. *The Open Source Alternative*. John Wiley & Sons, Inc. Hoboken, NJ.
9. Moen, R. 1999. *Fear of Forking essay*. Available at: http://linuxmafia.com/faq/Licensing_and_Law/forking.html
10. Moody, G. 2009. Who Owns Commercial Open Source – and Can Forks Work? *Linux Journal*, 23 April, 2009.
11. Moody, G. 2011. The Deeper Significance of LibreOffice 3.3. *Computerworld UK*, 28 January, 2011.
12. Nyman, L. 2013. Freedom and forking in open source software. *Proceedings of the Nordic Academy of Management Conference* (Reykjavik, Iceland, 2013).
13. Nyman, L. 2014. When is a Fork a Fork? On the Definition(s) of Code Forking. Forthcoming.
14. Nyman, L. and Lindman, J. 2013. Code Forking, Governance, and Sustainability in Open Source Software. *Technology Innovation Management Review*, January, pp. 7-12.
15. Nyman, L. and Mikkonen, T. 2011. To Fork or Not to Fork: Fork Motivations in Sourceforge Projects. In: Hissam, S. (Ed.), IFIP Advances in Information and Communication Technology, vol. 365. Springer, Heidelberg. pp. 259-268.
16. Nyman, L., Mikkonen, T., Lindman, J., and Fougère, M. 2011. Forking: the invisible hand of sustainability. In: *Proceedings of the OSS 2011: Towards Sustainable Open Source, Sustainability workshop* (Salvador, Brazil, 5 October, 2011).
17. Nyman, L., Mikkonen, T., Lindman, J., and Fougère, M. 2012 Perspectives on Code Forking and Sustainability in Open Source Software. In: Hammouda, I. (Ed.), IFIP Advances in Information and Communication Technology, vol. 378. Springer, Heidelberg. pp. 274-279.
18. Raymond, E. 1999. *The Cathedral and the Bazaar*. O'Reilly Media, Inc., Sebastopol, CA.
19. Raymond, E. 2001. *How to Become a Hacker*. Available at: http://www.catb.org/esr/faqs/hacker-howto.html
20. Robles, G. and González-Barahona, J.M. 2012. A Comprehensive Study of Software Forks: Dates, Reasons, and Outcomes. In: Hammouda, I. (Ed.), IFIP Advances in Information and Communication Technology, vol. 378. Springer, Heidelberg. pp. 1-14.
21. St. Lauren, A. 2004. *Open Source & Free Software Licensing*. O'Reilly, Sebastopol, CA.
22. Stewart, K.J. and Gosain, S. 2006. The Impact of Ideology on Effectiveness in Open Source Software Development Teams. *MIS Quarterly,* Vol. 30 No. 2, pp. 291-314. June 2006.
23. Weber, S. 2004. *The Success of Open Source*. Harvard University Press, Cambridge, MA.
24. Wheeler, D. 2007. *Why Open Source Software / Free Software (OSS/FS, FLOSS, or FOSS)? Look at the Numbers!* Available at: http://www.dwheeler.com/oss_fs_why.html
25. Ågerfalk, P.J. and Fitzgerald, B. 2008. Outsourcing to an Unknown Workforce: Exploring Opensourcing as a Global Sourcing Strategy. *MIS Quarterly* Vol. 32 No. 2, pp. 385-409/June 2008.

Appendix 1. Interview questions

The interviews were semi-structured, allowing for follow-up questions when needed. Examples of follow-up questions are given as bullet points. The interview began with three control questions, each with possible follow-up questions. The first question was about how they got into programming, followed by a question about their open source background. These questions were both to confirm that the interviewees met the criteria established for interviewees, as well as to establish some further details of their level of experience with programming in general and open source in particular. The third question was about their definition of a fork.

The questions were as follows:

1. When, in your opinion, is it appropriate to use the term "fork"?
    - Does it matter to the definition whether I plan to merge the program back with the original?
    - Does it matter to the definition whether I plan to modify the program?
    - Is it a fork if a functionality has been reused as part of the new program, but it is buried/invisible inside it, or at least not prominent?
        - What if the functionality is the entire forked program vs. only part of it?
    - Does the term require community involvement? Is it a fork if I take code for a personal program that I never intend to share with anyone else?
2. What are the benefits of the existence of forking as a practice?
3. What are the drawbacks of the existence of forking as a practice?
4. When, or under which circumstances, is it acceptable/OK to fork a program?
    - Have you ever forked a program, or been involved in a forked program?
    - Under what kinds of circumstances would you yourself consider forking a program in which you were participating?
    - Can you think of any examples of forking where you felt it was good that a fork had happened?
5. When is it not acceptable/OK to fork a program?
    - Under what kind of circumstances would you consider it a bad thing that a fork happened?
    - What kinds of forks do you hope could be avoided?
    - Can you think of any instances of forking which you felt it was a bad thing that a fork had happened?
6. When someone forks a program, are there any moral obligations towards anyone? The original developers? The community? Others?
7. Is there a proper code of conduct when one wants to fork a program, and if so, what is it?

PAPER 5

When is a Fork a Fork? On the Definition(s) of Code Forking

Nyman. Unpublished essay.

When is a Fork a Fork? On the Definition(s) of Code Forking

Linus Nyman, Hanken School of Economics

ABSTRACT

All open source licenses allow an existing body of code to be used as the basis for a separate development project, a practice commonly known as code forking. This paper presents results from a study in which 13 programmers representing four nations shared their views on forking. The study finds that there can be significant differences between programmers' definitions of a fork. Based on the findings herein, researchers should be mindful of the existence of the differences in the definition of fork when designing and conducting research about code forking.

Keywords: Code forking, fork, open source, code reuse, branch

1. INTRODUCTION

There is a small but growing number of studies whose focus has been specifically on the phenomenon of code forking, the process of copying a project's code base to use in the creation of a separate development. Areas of focus in previous research have included: conceptual papers focusing on the effect the right to fork and the practice of forking have on the sustainability and governance of open source software development (Nyman, Mikkonen, Lindman, and Fougère, 2011 & 2012; Nyman and Lindman, 2013); the motivations behind forks (Nyman and Mikkonen, 2011; Robles and González-Barahona, 2012, Viseur, 2012; Nyman, 2013); outcomes of forks (Robles and González-Barahona, 2012); case studies into community-related aspects as well as the long-term sustainability of forks (Gamalielsson and Lundell, 2012 & 2014); and how programmers view the right to fork and the practice of forking (Nyman, 2014).

Earlier research has noted some inconsistencies regarding the use of the term fork among developers, which suggests the existence of multiple definitions of the term (Nyman and Mikkonen, 2011; see also Nyman, 2014). This could be problematic for researchers in that programmers may be referring to different phenomena when discussing forking, or taking part in a survey about forking. This paper presents the results of interviews with developers regarding what constitutes a fork. The goal of the paper is to examine if there are significant differences among developer interpretations of the term. The paper is structured as follows: first, some background information is presented on extant definitions of a fork, followed by a discussion of methods, and then results. The paper concludes with a discussion of the findings, their implications, and avenues for future research.

2. BACKGROUND

Numerous definitions of forking have been used in the literature. At its most brief, a fork has been defined simply as being a version of an existing open source project (e.g. Moen, 1999).

A fork is sometimes defined as the result of irreconcilable differences among a developer team, leading to two different versions – implying that part of the definition is a requirement that the new version be started by some of the original developers.[1]

Incompatibility between the original program and the new program is sometimes included in the definition of a fork and often noted to be a largely unavoidable result of forking (e.g. Raymond, 1999; St. Lauren, 2004; Weber, 2004; Meeker, 2008).

Robles and González-Barahona (2012) offer the most detailed definition of a fork among existing literature, defining a fork as occurring when "a part of a development community (or a third party not related to the project) starts a completely independent line of development based on the source code basis of the project." They include four further criteria for something to be called a fork. It should have: 1) a new project name; 2) a branch of the software; 3) a parallel infrastructure;[2] and 4) a new developer community (disjoint with the original).

While it can be noted that extant definitions of a fork are similar, they are not always identical. This may in part be due to some definitions being more specific than others. However, there are also some explicit differences worthy of note, with two key differentiating features being the inclusion of incompatibility as a defining characteristic of a fork and the prerequisite that for something to be called a fork, it must originate from among the developers of the original program.

3. RESEARCH QUESTIONS

The goal of this paper is to examine developer views on what constitutes a fork, and to compare these views with one another. The two research questions investigated herein are:

Research question 1: *What are the key identifying characteristics of programmers' definitions of a fork?*

Research question 2: *Are there significant variations among programmer definitions of a fork?*

4. RESEARCH APPROACH

Before beginning the data acquisition, it was important to become familiar with the writings and thoughts on forking found in academic literature, practitioner texts, and other sources of knowledge on forking (developer blogs, online news sites, etc.). Based on this knowledge, an interview guide consisting of open-ended questions was compiled. During the winter of 2012, a pilot study was conducted consisting of 4 semi-structured interviews. The interviewees all fulfilled the criteria that they all either were, or had previously been involved in open source software programming. During these interviews, the interview guide was tested and modified. The interviews were conducted in person, and were recorded and transcribed.

[1] See the Jargon File: http://www.catb.org/jargon/html/F/fork.html.

[2] Web site, versioning system, mailing lists, etc.

Nine additional interviews were conducted between 2013 and 2014. The interviewees were chosen using snowballing, with the criteria that: 1) they needed to have been involved in at least one open source software project; and 2) they needed to currently make a living from programming, be study programming, or have at some point previously had made their living from programming. The interviews were conducted in Finnish, Swedish, and English, depending on the preference of the interviewee. Roughly half of the interviews were conducted face-to-face, the rest using Skype. The interviews were semi-structured, allowing for additional questions as needed, as well as giving the respondents the opportunity to express all thoughts and insights they felt relevant. All of the interviews were recorded and transcribed to aid in the data analysis.

A majority of the programmers were from Finland, the rest were from Italy, Russia, and the Ukraine. The interviewees ranged in age from their early twenties to their fifties. One interviewee was female, the rest were male. The programmer with the least experience was involved in his first open source project at the time of interview. The most experienced programmers had several decades of experience, and included a developer who had worked primarily on the Linux kernel, a developer employed by the Apache Foundation, a Linux pioneer who has additionally both authored as well as contributed to numerous other open source programs, and a developer who was a founding member of both MySQL and MariaDB.

To answer the research questions, a detailed understanding and analysis of the programmers' views was important. Specifically, it was necessary to identify the principal elements of a fork, i.e., at a minimum, what needs to happen for something to be considered a fork? To uncover this, interview questions first focused on when, in the programmer's opinion, it is appropriate to use the term "fork". Based on their initial answers, several potentially significant points were targeted with questions including:

- Does it require modification to be categorized as a fork?
- Does it require a split among developers that originally worked together on the project?
- Is it relevant how much code is taken, e.g. is it a fork if only a few lines are taken, or if a functionality is taken, or does it need to be the entire codebase?
- Does it matter to the definition whether one intends to merge changes back with the original?
- Is it sill a fork if one only intends to use it for personal use, with no plans for distribution?

5. FINDINGS

5.1 Programmers' definitions and views on key characteristics of a fork

When asked when it is appropriate to use the term fork, most developers gave rather broad definitions. Further specificity was then sought through additional questions. This section will begin with some sample quotes from their views on when it is appropriate to call something a fork, followed by a presentation of their views on specific characteristics and their relevance to the definition of a fork.

5.1.1 Programmers' definitions of a fork

Most of the developers offered quite similar preliminary definitions of a fork. Much like the previous definitions of a fork discussed in Section 2, some definitions focused on a

fork being a modified version of a pre-existing project, from which a new, active project emerged. Some example quotes follow:

> "I would say that [a fork is] when you take an existing program and not just create a new version of it, but start a whole new project that is separate from the original. […] If I take the Linux kernel and make some changes to it, it isn't a fork. But if I put it up on a server and [say] that this is now my kernel and I am going to continue developing it and I don't care what Linus Torvalds does, then I have forked."

> "*A fork is […] when the project takes two different – or three, or whatever – directions. […] There must be a difference. Essential difference. On the code.*"

> "*A fork is where you take the whole codebase and start developing it as a separate unit in whatever direction you want.*"

A further element some developers highlighted was that the versions must remain separate from one another.

> "Fork usually means that a program goes in two separate directions, and that those roads never again merge."

> "A fork is when the project is split. Not that you just copy a repository. You can set up your own repository in Git. Because of the distributed nature of Git versioning system, essentially you are always cloning a repository. So, just cloning a repository – I wouldn't call that forking. It has to be really you're starting like… it's separate, and you aren't constantly merging your changes back, it's kind of diverging."

Similar points noted by some were that the fork needed to co-exist with the original, with both continuing to be developed by separate groups.

> "I think it's accurate to talk about a fork when you get several groups maintaining [their own versions of] the code."

> "In my opinion fork implies that something exists, and continues to be more or less actively developed, and then another group comes and takes the same cake, in practice makes a copy of it, and tries to start taking it in a perhaps somewhat different direction."

Some answers were not explicit about the need for a separate project to emerge from the fork.

> "I would pretty much describe a fork [as] anything that's cloned from the master or the original and gets any sort of modification done. I mean… pretty much the term that GitHub[3] uses for a fork."

While most of the developers gave generally homogenous definitions centred around topics familiar from earlier literature, one developer gave a definition that was altogether different in its scope and specificity of demands:

> "[A fork is] when you get the entire repository. Which…. In the modern sense of things it also includes history. So, let's say in the old school when companies made releases that you would get source for, […] it was just a fixed snapshot, so that is not a fork. However, if it's like a… if it's a full repository with tests and everything else, and history, then I think that constitutes a fork."

Through follow-up questions, the developer in question was asked about LibreOffice and MariaDB – commonly considered forks of OpenOffice and MySQL, respectively.

[3] GitHub (https://github.com/) is a popular hosting service and revision control system launched in 2008.

While the developer used many terms in discussing these programs (e.g. a natural continuation or evolution of an earlier project), they were not considered forks.

A further developer began his answer by first discussing how the term has become something different:

> "Distributed revision control systems have made [the term fork] an entirely new thing [...] and GitHub and these kinds of services, where you can go to a website and click a button to 'fork this project', and then you have your own fork of it there. And, well, the purpose of this kind of thing isn't directly to create some new product, but rather that you mutate the code according to your needs and then maybe at some point do a pull request to the original."

The developer went on to contrast this with, and discuss, some of the better-known open source forks, including libc, X.org, and the BSD forks. He concluded his answer about the definition of a fork by noting, with a light-hearted laugh, that there's plenty of breadth of interpretation to it. In then discussing his own views on the definition of a fork, he noted that both a program that had been entirely rewritten could be considered a fork, as well as a program that had had no code modified whatsoever, but where significant focus had been put on branding and marketing of the software as a new, different product.

5.1.2 What are the key characteristics of a fork?

In addition to asking programmers about their definition of fork, they were also asked about several characteristics and to what extent they were required for a project to be considered a fork. The topics and sample responses were as follows:

Does it require a split among developers that originally worked together on the project?

No programmers were of the opinion that the term fork could only be used in cases where there was a split among developers resulting in two or more programs. Thus, according to the developers interviewed, a fork could come from either a split between previously co-working developers or from a developer or group of developers not at all affiliated with the original project. As one developer put it, "*I see it as code-based [rather than people-based]. If someone takes the code and development is born around it, then it is a fork of that code.*"

Does it require modification?

The most commonly noted feature of a fork was modification, i.e., that a program must be modified after having been copied (forked) from an original.

> "[I]n my opinion it's not really a fork if they don't do anything to it. It's just a clone from a snapshot of the original."

> "Have you changed [the code]? If you have changed it then it's a fork. If you just take it as part [of your program], a library or something like that, then it isn't a fork because you haven't changed it."

> "As soon as you start doing something with the code it is a fork."

However, one developer noted two extremes that could both be considered forks:

> "If [you] don't change a single line of code, but you rebrand [the program] and develop its sales, is that a fork or not? Well, viewed form the outside it's a fork, right? And then also, you [can]

> take some library [...] and note that the code is so bad that you go through it line by line and rewrite it completely. Then you rewrite all of it and at some point there is no longer a single line of code [left from the original]. These are both forks."

Does amount matter? Does it matter if you copy only a few lines of code, versus a functionality, versus the entire code base?

Here, views were mixed. While the majority noted that there needs to be at least a complete functionality (many developers used a library as an example) for a project to be considered a fork, one developer noted that even a few lines of code taken from one project and implemented in another can be considered forked from the original program. Another developer was of the opinion that the goal, rather than the amount of code copied, was what was truly important in deciding whether or not something should be called a fork. When asked if copying a functionality for a different project should be considered forking, he noted that, in his opinion, the goal was likely to be so different from the original program's that he didn't consider it forking. In such an instance he noted, "*You are just taking open source and doing something else. That's fine, but it's not forking*."

Does it matter if one intends to merge back with the original?

Here, the most common view was that it could accurately be described as a fork during the time between having been forked and having been merged back with the original. As one developer noted, it is a fork "*if you don't know that it is going to merge.*" However, some preferred the term branch, specifically if one knows it will merge back with the original project.

Does the term require community involvement? Is it a fork even if it is only for personal use, with no plans for distribution?

Responses to this question were mixed. A number of developers responded with a light-hearted or tongue in cheek comment about the lack of practical relevance of any answer.[4] Indeed, few seemed to have given the issue any thought. As one developer noted:

> "I suppose it is [a fork] but... but then again who... like the only person that would know about the fork would be yourself so...".

When discussing significant elements of the definition of a fork, another developer noted, "*usually you would say that one also has to distribute [the program], because otherwise other people won't know it's a fork.*"

5.2 Are there significant variations among the definitions?

All of the definitions of fork given by the programmers interviewed had significant overlap. At the core of what developers consider to be a fork is when a code base is copied in part or in its entirety and used as the basis for, or implemented into, a separate project. However, many developers also considered the term fork appropriate even when done as part of developing one program, without the intent of diverging from it. Furthermore, it was also noted that something could be considered a fork even if no code is reused, as well as if all of the code is reused but left entirely unmodified.

[4] One developer began his answer by noting the similarities between that question and the question of "if a tree falls in the forest and no one is there to hear it, does it still make a sound?"

Most developers were of the opinion that an internal split is not necessary for a project to be called a fork. No developers highlighted compatibility problems between the original and the fork as a necessary characteristic for defining something as a fork.

At one end of the spectrum of views expressed, copying and modifying a few lines of code was seen as fitting the term fork. At the other end of the spectrum, something was considered a fork only if it, in addition to the source code, also included all available tests, historical data, and other information from the original project. Furthermore, one developer noted that rewriting the entire codebase can be seen as a fork, but that so can reusing a program where the code remains unmodified but the software is externally rebranded. Given the striking difference between the extremes of the examples given, it is clear that significant variations among programmers' definitions of a fork do exist.

6. DISCUSSION, IMPLICATIONS, AND FUTURE WORK

6.1 Discussion

The findings inevitably raise at least two questions. Firstly, why is there such variance among definitions? And secondly, what categories of definitions, if any, can be identified?

Why are there different interpretations?

Discussing the issue of defining a fork, one developer noted that he didn't think programmers themselves were very clear on its definition, and that "*there's probably a lot of opinions [regarding] what a fork is.*" Indeed, when asked about specifics and details regarding what constitutes a fork, developers sometimes pondered the question, seemingly forming their opinion as they answered.

Some of the veteran developers discussed what may be a key insight into understanding how these differences in definitions have come about: the rise in popularity of programs that define fork in a very broad context.

> "The definition [of a fork] has actually changed over the past maybe five years, particularly due to Git and other similar [programs] that have kind of redefined the term."

Upon a brief examination, it appears that at least (if not specifically) GitHub has a very broad definition of a fork. Indeed, one of the younger developers interviewed noted their definition to be "*pretty much the term that GitHub uses for a fork.*" Another veteran developer noted that, particularly since the rise of GitHub and similar software, fork had become "*a fairly amorphous term*". Regarding forking, GitHub[5] states:

> "At some point you may find yourself wanting to contribute to someone else's project, or would like to use someone's project as the starting point for your own. This is known as 'forking'".

While starting a new project based on an existing one is quite a different endeavour than merely contributing to an existing project, GitHub defines the term fork broadly enough to encompass both.

[5] https://help.github.com/articles/fork-a-repo, accessed 12 April, 2014.

In addition to the recent effects GitHub may have had on the definition, language scholars note that language itself changes over time, a phenomenon called semantic change. Indeed, the data gathered for an earlier paper co-authored by this author (Nyman and Mikkonen, 2011) consisted of motivations reported by developers as to why they had forked programs they hosted on the source code repository SourceForge (sourceforge.net). The data gathered included forks from a timespan of 11 years, from late 1999 to the end of 2010. Thus, the majority of forks analysed were from before GitHub's founding. Nonetheless, there seemed to be a plethora of cases that couldn't conveniently be categorized as a fork, when defined as it commonly is in the literature. The broadening of interpretation of the term fork, then, is a phenomenon that is likely to have been hastened by, but simultaneously be older than, the emergence of GitHub.

What categories of definitions can be identified?

This study did not gather enough data to speak with any kind of certainty to this point. However, what data there is suggests there to be to be two main categories, but with a broad enough spread as to be more reminiscent of a continuum with groupings close to either end. Alternatively, the definitions offered could be described as being along a cone or funnel, with narrow definitions at one end, and broad definitions at the other.

At the one end is what one could call the narrow, or classical interpretation of a fork. At its narrowest, a fork is defined as a split among developers, who for whatever reasons decide they can no longer work on the same project, and they split up into two different projects (to produce software that either immediately is, or over time becomes, incompatible with the original). While few if any programmers' definitions fit this category unerringly, quite a few definitions fit within a broader area of this narrow end. Specifically, when a fork is seen to not need to stem from a split among the original developers of a project. Common to these definitions was that a fork can come from anywhere, any individual or group, and results in a new project with goals that are at least to some extent different from the original.

At the other end is a category we might call the broad interpretation. Perhaps the most significant difference is that this definition has expanded to include elements like clones and branches, which were previously given their own definitions.[6] Thus, while a fork may result in a new project, it may also simply be part of the workflow of a project, done by a person or group with no plans for creating a new, diverging project. Furthermore, in its broad interpretation, copying a project for one's own personal inspection or use is also considered a fork.

6.2 Implications

The answer to the question of when a fork is a fork seems to be: that depends on whom you ask. The implications of these findings are that future research with a focus on code forking must be mindful of the very real risk of different interpretations. Specifically, studies should be designed to minimize variations in the interpretation of fork in interviews and questionnaires. Otherwise, researchers run the risk of collecting data on somewhat different phenomenon erroneously categorized under the same name,

[6] However, even this alone is not enough to explain the findings in their entirety, as one programmer offered a definition neither synchronous with early definitions nor with that of GitHub. A greater sample size would allow speculation as to whether this one case would be an anomaly overall, or a further general shift of the meaning of the term. The findings of this paper suggest the former to be the case.

potentially invalidating comparisons between multiple studies or even multiple study participants.[7]

6.3 Future work

The limited amount of programmers interviewed means that the results do not lend themselves to generalization regarding specific numbers or percentages of developers who view the definition of forking in a particular way. Future research could attempt to add clarity to the extent of adoption of the definitions uncovered in this study, as well determine if there are other views not mentioned herein. Even allowing for some overlap between categories, some definitions do not fit perfectly in either category. Such a study could also test the suitability of the categories suggested herein, as well as add clarity to it.

One aspect that surfaced in the interviews, but that wasn't addressed in the questions to developers, was the question of forks that are effectively complete rewrites. Mozilla, for instance, is commonly considered a fork of Netscape, yet arguably no actual code was (ultimately) reused in its writing.[8] Thus the notion that a requirement for a fork is that the original code must be copied and modified may, in fact, also be considered expandable to merely the original idea being copied. It would seem plausible that even those of a narrower definition of fork would accept the notion that such a rewrite could be considered a fork, at least in a case in which members of the original project start the new project, or there is an initial attempt to reuse code, or both.

One of the challenges future work in this area is likely to face is that the meaning of the term appears to be in flux. Will the definition continue to broaden? The answer to this will likely depend in no small part on what tools become popular, and how those tools use and define the term fork.[9] A further important issue may be how respected and prolific members of the open source community define forks. Unchecked, the term fork may continue to change over time. Therefore, research may only allow for the capturing of a snapshot of its meaning among a limited group at a specific point in time.

Achieving a unified definition of a fork, commonly accepted among researchers and practitioners alike, would seem a significant goal. However, given the differences that already exist, it may not be a realistic one. Were one to attempt such a goal, one must first decide a starting point. Previous definitions among academics (e.g. Robles and González-Barahona, 2012) and practitioners (e.g. Raymond, 1999) alike offer useful distinctions between forks and fork-like phenomenon.[10] However, how would one go about achieving widespread adoption of these among practitioners? Taking the common interpretation among practitioners as a starting point for a standardized definition is no less problematic, as this study suggests that no comprehensive, commonly accepted definition among practitioners exists. A more realistic goal is a

---

[7] Indeed, while the definitions used by Gamalielsson and Lundell (2012 & 2014) and Robles and González-Barahona (2012) are compatible (though the research focus is different), the approach seen in Nyman and Mikkonen (2011) of a fork as anything the programmers proclaims to be a fork may, in light of the findings of this study, be problematic, as the studies are thus likely to a greater or lesser degree to be addressing related yet potentially different phenomena. Viseur (2012) did not specify the definition or selection criteria used in selecting forks.

[8] See, e.g., http://www-archive.mozilla.org/roadmap/roadmap-26-Oct-1998.html.

[9] Will GitHub's popularity endure? How will other tools that have yet to be released define a fork?

[10] E.g. pseudo-forks, fragmentation, distributions, and branches.

unified definition within academia. If we accept this as a goal to strive for, what should that definition include?

A definition that attempts to encompass all, or as many as possible, of the variations of the term found in use among practitioners seems counter-productive to the practice of researching open source. The interpretations covered in this study have included the term fork used to describe actions done as part of a workflow, with no intention of splitting off into separate projects, as well as reusing a few lines of code. Including under the heading of "code forking" activities that are effectively code reuse or a common method of distributing work in an open source project makes the definition unhelpfully broad. Furthermore, as the example of a rewrite seen as a fork infers, a fork may not be definable solely through the analysis of how code is reused.

If we accept that the broader definitions encountered in this study are unhelpfully broad, we can explore further the narrower definitions offered. Indeed, here we can see that there is much similarity among the definitions offered by those of the narrower interpretation of a fork and the definition put forth by Robles and González-Barahona (2012). Their definition is somewhat more detailed in its requirements than what is commonly found among practitioner views, specifically when taking into account the four additional criteria given beyond the general definition. Furthermore, these additional criteria might exclude some events that could usefully be described as forks. However, their definition would nonetheless seem the most fertile of available starting points in the pursuit of a definition that would both be helpfully narrow, as well as find some overlap between both researchers and practitioners. In fact, calling it a starting point may be unnecessarily harsh, as even uniting around their definition as-is is likely to exclude only a relatively small number of events that even said authors might agree could beneficially be called forks, and could be commonly accepted as exceptions to the rule. Thus, it would be beneficial if academia could unite around the definition of fork offered by Robles and González-Barahona, either as-is, or with some small modifications to allow for the inclusion of exceptional cases that nonetheless could fittingly be called forks.

Previous work (e.g. Nyman and Mikkonen, 2011; Robles and González-Barahona, 2012; Nyman, 2013 & 2014) would seem to suggest that, regardless of the definition used, forks encompass a wide variety of motivations. A further goal worthy of consideration is to explore a sub categorization based on (or at the very least including) motivational prefixes, as have already been used on occasion, e.g. friendly forks (Robles and González-Barahona, 2012), experimental forks (Nyman and Mikkonen, 2011), and competitive forks (Nyman, 2013). This approach might furthermore alleviate some of the problems of the existing breadth of interpretation of the term fork.[11] Indeed, even given a unified definition of when a fork is a fork, the addition of a descriptive prefix would offer some immediate insight into the kind of fork in question, as well as provide potentially interesting starting points for comparisons among and between groups. With the increased ease with which forks can be created, and the many related options offered by version control systems and other software, it may also be relevant to include some technical sub categorization to definitions of a fork. Future work, preferably co-authored by several researchers interested in the phenomenon in order to aid reaching a common view and its widespread adoption, could explore these notions further.

[11] However, even such sub categorizations are not likely to be free from difficult definitional boundaries. E.g., what are the specific criteria for a fork to be considered to compete with the original? While perhaps unproblematic in the case of MariaDB, a drop-in replacement for MySQL, other forks may retain some elements or functionalities that compete with the original, with other new elements that do not.

7. REFERENCES

Fogel, K. (2006) *Producing Open Source Software*. O'Reilly, Sebastopol, CA.

Gamalielsson, J. and Lundell, B. (2012) Long-Term Sustainability of Open Source Software Communities beyond a Fork: A Case Study of LibreOffice. In: *Proceedings of the 8th IFIP WG 2.13 International Conference, OSS 2012* (Hammamet, Tunisia, 2012).

Gamalielsson, J. and Lundell, B. (2014) Sustainability of Open Source software communities beyond a fork: How and why has the LibreOffice project evolved? *Journal of Systems and Software* 89: 128-145.

Meeker, H. (2008) *The Open Source Alternative*. John Wiley & Sons, Inc. Hoboken, NJ.

Moen, R. (1999) *Fear of Forking essay*. Available at: http://linuxmafia.com/faq/Licensing_and_Law/forking.html.

Nyman, L. (2013) Freedom and forking in open source software. *Proceedings of the Nordic Academy of Management Conference*. Reykjavik, 21-23 August.

Nyman (2014). Hackers on Forking. *Proceedings of the 10th International Symposium on Open Collaboration* (OpenSym 2014). ACM, 2014. ISBN: 978-1-4503-3016-9, doi>10.1145/2641580.2641590

Nyman, L. and Lindman, J. (2013) Code Forking, Governance, and Sustainability in Open Source Software. *Technology Innovation Management Review*, January, pp. 7-12.

Nyman, L. and Mikkonen, T. (2011) To Fork or Not to Fork: Fork Motivations in Sourceforge Projects. In: Hissam, S. (Ed.), IFIP Advances in Information and Communication Technology, vol. 365. Springer, Heidelberg. pp. 259-268.

Nyman, L., Mikkonen, T., Lindman, J., and Fougère, M. (2011) Forking: the invisible hand of sustainability. In: *Proceedings of the OSS 2011: Towards Sustainable Open Source, Sustainability workshop* (Salvador, Brazil, 5 October, 2011).

Nyman, L., Mikkonen, T., Lindman, J., and Fougère, M. (2012) Perspectives on Code Forking and Sustainability in Open Source Software. In: Hammouda, I. (Ed.), IFIP Advances in Information and Communication Technology, vol. 378. Springer, Heidelberg. pp. 274-279.

Raymond, E. (1999) *The Cathedral and the Bazaar*. O'Reilly Media, Inc., Sebastopol, CA.

Robles, G. and González-Barahona, J.M. (2012) A Comprehensive Study of Software Forks: Dates, Reasons, and Outcomes. In: *Proceedings of the 8th IFIP WG 2.13 International Conference, OSS 2012*. Hammamet, Tunisia.

St. Lauren, A. (2004) *Open Source & Free Software Licensing*. O'Reilly, Sebastopol, CA.

Viseur, R. (2012) Forks impacts and motivations in free and open source projects. *International Journal of Advanced Computer Science and Applications*, Vol. 3, No. 2.

Weber, S. (2004) *The Success of Open Source*. Harvard University Press, Cambridge, MA.

EKONOMI OCH SAMHÄLLE
Skrifter utgivna vid Svenska handelshögskolan

ECONOMICS AND SOCIETY
Publications of the Hanken School of Economics

252. NADER SHAHZAD VIRK: Explanations for Finnish Stock Returns with Fundamental and Anomalous Risks. Helsinki 2013.

253. TAMARA GALKINA: Entrepreneurial Networking: Intended and Unintended Processes. Helsinki 2013.

254. JOHANNA ARANTOLA-HATTAB: Family as a Customer Experiencing Co-Created Service Value. Helsinki 2013.

255. ARGYRIS ARGYROU: Developing Quantitative Models for Auditing Journal Entries. Helsinki 2013.

256. JAKOB STORÅ: Earnings Management Through IFRS Goodwill Impairment Accounting: In the Context of Incentives Created by Earnings Targets. Helsinki 2013.

257. LILIA HYTTINEN: Pharmaceuticals' Strategic Interactions: Three Essays in Corporate Finance. Helsinki 2013.

258. TOMMY OLIN: Det sociala kapitalets inverkan på företagande: En studie i förutsättningarna för företagande i Purmo 1945 – 1976. Helsingfors 2013.

259. JONNA LOUVRIER: Diversity, Difference and Diversity Management: A Contextual and Interview Study of Managers and Ethnic Minority Employees in Finland and France. Helsinki 2013.

260. FRANS SAXÉN: Essays on the Economics of Retailing: Payments, Finance and Vertical Restraints. Helsinki 2013.

261. HILAL ANWAR BUTT: Asset Pricing in Small Sized Developed Markets. Helsinki 2013.

262. PAUL CATANI: Misspecification and Bootstrap Tests in Multivariate Time Series Models with Conditional Heteroskedasticity. Helsinki 2013.

263. HELI HOLTTINEN: Cultural Ideals, Practices and Value Propositions in Consumer Everyday Value Creation. Helsinki 2013.

264. MIKKO VESA: There be Dragons! An Ethnographic Inquiry into the Strategic Practices and Process of World of Warcraft Gaming Groups. Helsinki 2013.

265. HENRICH NYMAN: Service Profitability: An Augmented Customer Lifetime Value Approach. Helsinki 2013.

266. HANNA-RIITTA HARILAINEN: Managing Supplier Sustainability Risk. Helsinki 2014.

267. JACOB MICKELSSON: Customer Activity: A Perspective on Service Use. Helsinki 2014.

268. MIKAEL LAAKSO: Measuring Open Access: Studies of Web-enabled Innovation in Scientific Journal Publishing. Helsinki 2014.

269. HANNA KIEHELÄ: Dimensionality of the Consumer Perceived Value of Product Colour. Helsinki 2014.

270. LINDA TALLBERG: Processing Puppies: An Animal Shelter Ethnography. Helsinki 2014.

271. PIA HELLMAN: The Effect of Communicating E-service Benefits on Consumer E-service Adoption. Helsinki 2014.

272. PENG WANG: Four Essays on Corporate Finance of Chinese Listed Firms. Helsinki 2014.

273. DHANAY MARÍA CADILLO CHANDLER: The Role of Patents in Latin American Development: 'models of protection' of pharmaceutical patents and access to medicines in Brazil, Chile and Venezuela. Helsinki 2014.

274. CARLOS A. DIAZ RUIZ: Market Representations in Action: Foundations for the Performativity of Representations in Marketing. Helsinki 2014.

275. IRA HAAVISTO: Performance in Humanitarian Supply Chains. Helsinki 2014.

276. SALLA PÖYRY: Essays on Financial Market Frictions and Imperfections. Helsinki 2014.

277. HELENA LIEWENDAHL: What Motivates Employees to Live up to Value Promises: An Employee Discourse. Helsinki 2014.

278. ALEXEI KOVESHNIKOV: Micro-Political Perspectives on Multinational Corporations: Legitimation, Stereotyping and Recontextualization. Helsinki 2014.

279. FRÉDÉRIC DÉLÈZE: Essays in Quantitative Analysis of the Effects of Market Imperfections on Asset Returns. Helsinki 2014.

280. KAI HUOTARI: Experientializing – How C2C Communication Becomes Part of the Service Experience. The Case of Live-tweeting and TV-viewing. Helsinki 2014.

281. RICHARD KEDZIOR: How Digital Worlds Become Material: An Ethnographic and Netnographic Investigation in Second Life. Helsinki 2014.

282. MARIA FORSS: Fortbildning är mer ”fort” än ”bildning”. En kritisk granskning av fortbildning för sjukskötaren. Helsingfors 2014.

283. MAGNUS BLOMKVIST: Essays on Market States, Firm Growth and Corporate Finance Decisions. Helsinki 2014.

284. ANNAMARI TUORI: Doing Intersectional Identity Work: Social Categories, Inequalities, and Silences. Helsinki 2014.

285. SANNE BOR: A Theory of Meta-Organisation: An Analysis of Steering Processes in European Commission-Funded R&D ‘Network of Excellence’ Consortia. Helsinki 2014.

286. RINAT MUKMINOV: Deposit Markets, Lending Markets and Bank Screening Incentives. Helsinki 2015.

LINUS NYMAN

UNDERSTANDING CODE FORKING IN OPEN SOURCE SOFTWARE: AN EXAMINATION OF CODE FORKING, ITS EFFECT ON OPEN SOURCE SOFTWARE, AND HOW IT IS VIEWED AND PRACTICED BY DEVELOPERS

Open source software is everywhere. From phones, tablets, TVs, and game consoles to less self-evident examples like cars, washing machines, and the International Space Station. However, what makes open source software remarkable is not where it can be found, but rather what can be done with it. One of the most astounding rights guaranteed by all open source software licenses is the right to fork the source code. In other words, the right to copy any program, either in part or in its entirety, and use that program to create a new, modified version of it.

The right to fork has an enormous impact on both the development and governance of open source software. Despite its significance, code forking has seen little academic study. This dissertation examines the right to fork, its impact and significance, and how it is viewed and practiced by developers.

The study draws on data consisting of hundreds of forks, interviews with open source software programmers, and an in-depth analysis of the birth of the MariaDB fork. This dissertation is relevant to anyone seeking a greater general understanding of how open source works and why it is considered a superior software development model. It may also serve as a useful resource for firms seeking to harness the power of open source software. Furthermore, it offers important insights to those who want to better understand how code forking is practiced and viewed by developers.

This study finds that forks are primarily started for non-competitive reasons, with unique features or goals that distinguish them from their parent projects. Competitive forks are rare but do exist, with some motivating factors being to ensure the freedom of the code and the community's ability to contribute to it. Furthermore, though developers may not always agree with the forking of a project, they nonetheless consider the right to fork to be of vital importance, and a cornerstone of free and open source software.

In many ways, open source can be thought of as a return to how software was developed before the emergence of proprietary licensing. The same freedoms of development and sharing that thrived back then can be found today in the open source community. Indeed, in many ways the right to fork is synonymous with freedom: the freedom to explore and experiment, the freedom to benefit from the work done by others, and the freedom to keep any project relevant and vibrant even when faced with leadership decisions that are deemed unsupportable. In short, the right to fork is open source software's guardian of freedom and watchdog of meritocracy.

ISBN 978-952-232-274-6 (printed)
ISBN 978-952-232-275-3 (PDF)
ISSN-L 0424-7256
ISSN 0424-7256 (printed)
ISSN 2242-699X (PDF)

EDITA PRIMA LTD, HELSINKI



HANKEN
SCHOOL OF ECONOMICS

HELSINKI
ARKADIANKATU 22, P.O. BOX 479
00101 HELSINKI, FINLAND
TEL +358 (0)9 431 331. FAX +358 (0)9 431 33 333

VAASA
KIRJASTONKATU 16, P.O. BOX 287
65101 VAASA, FINLAND
TEL +358 (0)6 3533 700. FAX +358 (0)6 3533 703

PUBL@HANKEN.FI
HANKEN.FI/DHANKEN