Palacký University, Olomouc

Faculty of Arts

Department of Sociology, Andragogy and Cultural Anthropology

KNOWLEDGE PRODUCTION IN FREE AND OPEN SOURCE SOFTWARE DEVELOPMENT

Dissertation thesis

Author: **Tomáš Karger**

Advisor: **Doc. PhDr. Dušan Lužný, Dr.**

Olomouc

2015

I declare that this thesis is my own original work. Where other people’s work has been used, this has been properly acknowledged and referenced in accordance with departmental requirements.

Olomouc, May 27th, 2015 …………………………………………..

I would like to express my gratitude to those, who influenced the shape of this work:

Doc. Dušan Lužný for providing me with opportunities to grow,
Prof. Libora Oates-Indruchová and Prof. Gerlinda Šmausová for their guidance,
Dr. Jan Kalenda for all the discussions,
my family and friends for their support and understanding.

Annotation (CZ)

Svobodný software představuje formu kooperativní produkce, která je založena na redukci transakčních nákladů, jíž je dosaženo pomocí neformálních způsobů organizace a využívání internetové infrastruktury. Cílem práce je zachycení dynamiky vědění v projektech svobodného softwaru, aby tak mohla popsat roli, kterou vědění hraje v této konkrétní formě kooperativní produkce. Přístup ke svobodnému softwaru je založen na konceptualizaci programování (ústřední aktivity ve vývoji softwaru) jako aktivity založené na specifických typech vědění. Dále rozlišuji mezi věděním a informacemi, což mi umožňuje formulovat problém dekontextualizace. Předpokládám, že konfigurace aktérů a artefaktů, jenž se promítá do kognitivních procesů, má podobu sítí. K jejich analýze je pak využit jazyk sociologické teorie označované jako Actor-Network Theory, konkrétně koncept mediace specifikovaný jeho čtyřmi významy: kompozice (composition), překlad (translation), delegace (delegation) a zneprůhlednění (black-boxing).

Metodologicky je práce ukotvena v přístupech technické etnografie, etnografie infrastruktury a multi-sited etnografie. Sběr dat proběhl v rámci terénní práce (která zahrnovala zúčastněné pozorování a analýzu dokumentů) v projektu svobodného softwaru – působil jsem v roli autora dokumentace vyvíjeného softwaru. Na základě pozorování popisuji programování jako druh praxe, v rámci které je jednání stabilizováno, sestavováno a delegováno (prostřednictvím zkompilovaných programů) na množství míst (počítače uživatelů). Softwarové nástroje v tomto procesu slouží k transformaci nestabilního průběhu práce do standardizovaných jednotek, jenž jsou delegovány na veřejná místa a jsou uzpůsobeny tak, aby bylo možné je zahrnout do jiných kompozic. Taková konfigurace pak značně redukuje transakční náklady kooperativní produkce.

I když ale licencování typické pro svobodný software záměrně a systematicky tlumí práva (na přístup, modifikaci a redistribuci) tradičně vázané na vlastnictví tím, že je připisuje každému, činnosti, které tato práva definují, jsou v praxi vykonávána jen malou skupinou aktérů, jenž jsou nositeli určitých typů vědění. V důsledku toho se pak zdá, že v tomto prostředí existuje těsný vztah mezi vlastnictvím (na praktické úrovni) a věděním. Toto zjištění na první pohled potvrzuje předpoklady entuziastických očekávání, vztahujících se k potenciálu kyberprostoru a digitálních technologií obecně, že vědění se stalo nejvýznamnějším faktorem produkce. V této práci se ale snažím ukázat, že vědění je stále umístěno v rozsáhlejších systémech produkce, jenž mají materiální povahu a že požadavky, které tyto systémy ukládají aktérům, limitují jejich domněle volnou interakci.

Table of Contents

Preface

The original topic of my dissertation research was quite a different one. It should have been a theoretical work about the concept of self-organization (or emergence, spontaneous order) and its use in the social sciences. Having a technological background in my education, I was fascinated by the language of Cybernetics and Systems Theory (Niklas Luhmann’s theory in particular) and by the images of spontaneously emerging order. The reader can still trace these influences in number of footnotes through the work. It was only during the first year of my Ph.D. studies that I realized that I could put my education to use in a different way. I could do participant observation in a field where not many sociologists or anthropologists would feel at home. This went hand in hand with the fact, that I found two fields to be repeatedly listed as empirical examples of self-organization: science and free and open source software development. The former has been researched for quite some time now with a substantial body of literature on record. But the latter has only about a decade and a half on its record with a significantly smaller body of literature. This was an area I could make a good contribution to. Moreover, I was a user interested in the latest development in free and open source software since high school (where I studied electrotechnical engineering), which provided me with a rough picture of the field’s most basic dividing lines.

All of this led to my decision to officially change the topic of my dissertation, leaving me with a lot to catch up on but also with an advantage of my prior everyday knowledge which, of course, must have undergone a thorough reflection. Eventually, my findings led me to a rather critical position towards the images associated with self-organization as I uncovered their limited relevance and the utopic valence they carry. There is a little irony in the fact that one of the points of this work is to show the limitations of the images which initially fascinated me and have drawn me to the topic. But I take this as a sign that I did not succumb to my initial preconceptions. How much this work will transform the knowledge in the relevant fields of study remains to be seen. However, a different transformation has already taken place, that of its author.

Introduction

For some, software development is an activity obscure enough that it should nicely connect to the anthropological tradition of studying exotic cultures covered with mystery. Others may find it as boring as reading through telephone books – a different kind of information infrastructure. However, reading and writing of source code (which is the primary activity behind software development) has been, by many influential individuals, recently pronounced to be a new form of literacy.[1] The atmosphere induced by this assumption has spread considerably. Software developers tend to criticize it[2] while the officials endorse it – to name the most glaring example, Barack Obama became the first US president to write a line of source code.[3] Without trying to position myself in the discussion about the legitimacy of the literacy status of programming, I want to point out that the spread and significance of activities associated with software development have risen considerably. From the times when computers could be afforded by few and the knowledge necessary to operate them was held by even fewer to the times when GitHub, a web service designed to share source code belongs among the top 100 most visited sites on the Internet.[4]

This development went hand-in-hand with another process – the shift in which free and open source software (as a movement or as a software development methodology) established itself against the traditional proprietary model. It could suffice to reiterate the glaring difference between the 1970s and the present day, pointing at the scale of involvement of significant players of the technological industry in open source software.[5] However, there is a more illustrative way of showing the gradual establishment of free and open source software.

In 2001, the Microsoft CEO Steve Ballmer famously stated that "Linux is a cancer that attaches itself in an intellectual property sense to everything it touches."[6] For a long time, the

1 See the Code.org initiative which revolves precisely around this assumption and is supported by the likes of Bill Gates or Mark Zuckerberg.

2 *Coding is not the new literacy*. Blog post of a well known software developer relayed by Slashdot, a popular user curated news site. Published: 2015-01-26. Accessed: 2015-04-30. Available at: http://www.chris-granger.com/2015/01/26/coding-is-not-the-new-literacy/.

3 *President Obama Is the First President to Write a Line of Code*. Article published by The White House Blog. Published: 2014-12-10. Accessed: 2015-04-22. Available at: https://www.whitehouse.gov/blog/2014/12/10/president-obama-first-president-write-line-code.

4 *How Github Conquered Google, Microsoft, and Everyone Else*. An article in the Wired magazine. Published: 2015-03-12. Accessed: 2015-04-21. Available at: http://www.wired.com/2015/03/github-conquered-google-microsoft-everyone-else/.

5 Among top contributors to the development of Linux Kernel, the hallmark of open source software, there are companies such as Intel, Samsung, IBM, or Google (Corbet, Kroah-Hartman, & McPherson, 2015, p. 11).

6 *Microsoft CEO takes lunch break with the Sun-Times*. Interview published by Chicago Sun Times. Published: 2001-06-01. Accessed: 2015-04-22. Available at:

Microsoft company was seen as an arch enemy of free and open source software. This relationship occasionally culminated in statements like Ballmer's or, from the other side of the barricade, pokes by Linus Torvalds: "Really, I'm not out to destroy Microsoft. That will just be a completely unintentional side effect."[7] This was in 2003. One decade later, in October 2014, a Microsoft CEO Satya Nadella says that "Microsoft loves Linux."[8] In February 2015, Microsoft releases its important .NET framework on GitHub under an open source license.[9] And in April 2015, Mark Russinovich, one of Microsoft's top engineers publicly states that open sourcing Windows, the company's core product, is "definitely possible".[10]

These statements mark a shift during which the open source approach to software development rose from a challenger to the one who sets the tone. Microsoft, a company that has embodied the success of proprietary approach to software development, is the one who has to catch up, as Nadella openly admits.[11] Indeed, free and open source software became ubiquitous in the world of digital technology, in large part because other companies like Google learned to build their business models around this type of software development.[12] The shift was also recognized in the world of Linux distributions, where Ubuntu, one of the most popular distributions had a long standing bug in its issue tracking database (filed in 2004, it was actually the first bug in the database) which was labeled "Microsoft has a majority market share".[13] The bug served as a mission statement – that Ubuntu was intended to provide an alternative which could possibly end the dominance. Eventually, the role played by Ubuntu in the shift was not essential, but the shift took place nevertheless. Mark Shuttleworth, the founder of Ubuntu, closed the bug in May 2013, noting that Microsoft no

http://www.linuxtoday.com/infrastructure/2001060100920OPMS.

7 *The Way We Live Now: Questions for Linus Torvalds*. Interview published by The New York Times Magazine. Published: 2003-09-28. Accessed: 2015-04-22. Available at: http://www.nytimes.com/2003/09/28/magazine/the-way-we-live-now-9-28-03-questions-for-linus-torvalds-the-sharer.html.

8 *Why Microsoft CEO Satya Nadella Loves What Steve Ballmer Once Despised*. Article published by the Wired magazine. Published: 2014-10-21. Accessed: 2015-04-22. Available at: http://www.wired.com/2014/10/microsoft-ceo-satya-nadella-loves-steve-ballmer-despised/.

9 *.NET Core is Open Source*. Blog post on the Microsoft Developer Network website (msdn.com). Published: 2014-11-12. Accessed: 2015-04-22. Available at: http://blogs.msdn.com/b/dotnet/archive/2014/11/12/net-core-is-open-source.aspx.

10 *Microsoft: An Open Source Windows Is 'Definitely Possible'*. Article published by the Wired magazine. Published: 2015-04-03. Accessed: 2015-04-22. Available at: http://www.wired.com/2015/04/microsoft-open-source-windows-definitely-possible/.

11 *Why Microsoft CEO Satya Nadella Loves What Steve Ballmer Once Despised*. Article published by the Wired magazine. Published: 2014-10-21. Accessed: 2015-04-22. Available at: http://www.wired.com/2014/10/microsoft-ceo-satya-nadella-loves-steve-ballmer-despised/.

12 For example, Google's Android, currently the most popular mobile operating system, is based on Linux.

13 *Microsoft has a majority market share*. Bug in Ubuntu's issue tracker at launchpad.com. Published: 2004-08-20. Accessed: 2015-04-23. Available at: https://bugs.launchpad.net/ubuntu/+bug/1.

longer had a dominant market share in computing platforms.

With this introduction, I do not intend to argue that free and open source software development is superior to the other development models or that it is the future of computing. My intention was merely to show the relevance of this topic – that while the practices and cultural aspects of free and open source software may seem unusual, they do not involve just a few hobbyists at the periphery of the computing industry. It is now heavily involved in developing and maintaining the information infrastructure that became so important during the last decades. The subculture became heavily intertwined with current capitalist practices, changing the nature of both in the process. And it is the aim of this work to describe free and open source software development precisely in this context.

The text is structured as a research report. In the first chapter, I give a review of research on FOSS relevant for this text and formulate the problem I attempt to approach. In the second chapter, I elaborate software and programming from a theoretical perspective and relate them to knowledge. Subsequently, I work out a conceptual grid that I use in my analysis and reformulate the problem at hand in terms introduced by this chapter. In the third chapter, I develop a methodological approach (including sampling criteria) that matches the assumptions of previous chapters. In the fourth chapter, I describe the most significant elements and practices of a FOSS project and use the conceptual grid to make visible the characteristics of mediation and resource allocation. In the fifth concluding chapter, I assemble the entities dismantled by the analysis into an overall picture of a FOSS project and attempt to relate the findings to the problem outlined in the first chapter.

1. FOSS and Utopian Virtualism

Free and open source software development began to be systematically researched by the turn of the millennium. According to Christopher Kelty, who has been studying it consistently during the first decade after the turn, it can be defined as:

> software whose source code (the code humans read and write) is made freely available (generally on the Internet, without restriction) through the use of a special copyright license. The software is copyrighted by its creator and then distributed under one of several standard licenses that allow the licensee to use the software, to distribute it, to copy it, and even to modify it for his/her own purposes. Some licenses require that if the software is re-distributed, any changes need to be released under the same license used to offer it in the first place (this is variously referred to as reciprocal, recursive, or viral). The most famous of these licenses is the GNU General Public License created by the Free Software Foundation. (Kelty, 2004, p. 501)

Many of the characteristics that Kelty describes in his definition were inherited from the practices established around UNIX, a very successful operating system from the 1970s. Its success is attributed to fact that it could run on affordable computers, that its source code was distributed together with its binaries and that its license permitted modifications to source code and even sharing of those modifications among licensees (Söderberg, 2008, p. 15). Retrospectively, we[14] can see that the main characteristics of the development model central to free and open source software was present already in practices related to the predecessor (direct or indirect) of many of today's widespread operating systems including a variation of BSDs (Berkley Software Distribution), Linux-based distributions or Mac OS X.

However, in 1980s, the AT&T company attempted to enforce ownership rights over UNIX which, according to Söderberg, resulted in the informal programmer community established around UNIX becoming skeptical of the existing intellectual property regime (Söderberg, 2008, p. 19). This led Richard Stallman to found the Free Software Foundation (FSF) in 1984 – an organization dedicated to allowing computer users to operate without proprietary programs. The break with the privatized UNIX system is represented by the acronym GNU (GNU is Not UNIX), that is used to label all software and licenses (as in GNU C Compiler, or GNU General Public License) that the FSF produces. The endeavors of FSF included development of a operating system kernel (called GNU Hurd) as a substitution for

14 In this text I employ one rule concerning the use of personal pronouns consistently: I use "we" when I take into consideration the reader following my description or argumentation; "I" is used in every other case, often indicating that I take responsibility for particular decisions or the unfolding of the text in general.

UNIX. However, the work has been significantly delayed due to licensing issues with the Mach microkernel, which was to be released by the Carnegie Mellon University under a suitable license and thus was proposed by Stallman to be used as a basis for development.[15]

Another reaction to privatization of UNIX came from the researchers of Berkeley University who participated heavily on UNIX development. They resorted to removing every line of code from UNIX that AT&T claimed and replaced them with their own code. The result is known as the Berkeley Software Distribution (BSD) and is still actively developed in several versions. However, in early 1990s, AT&T sued Berkeley University for infringement, which led to a trial that was eventually lost by AT&T, but that, in the meantime drove developers away with fear that their work could end up being claimed by the company (Söderberg, 2008, p. 24). Instead, the developers started contributing to another kernel project written from scratch by Linus Torvalds and licensed purely under the GNU GPL. This project, today known as Linux, established a strong position during the rest of the 1990s and became (together with other successful projects such as the Apache web server) the hallmark of the new software development model (Kelty, 2004, p. 503). The conclusion that Söderberg draws from this historical development is that Linux succeeded "not because it was backed by the highest concentration of capital, but to the contrary, because under the GPL it had the *purest absence of private property relations* [emphasis original]" (Söderberg, 2008, p. 24).

However, in spite of the success of several projects, free software was predominantly perceived as hostile to private businesses – a result of the value system developed by Stallman and the FSF, which considered free software a moral standard and which was very critical of anyone using or developing proprietary software. To capitalize on the success of free software projects, Eric Raymond and Bruce Perens founded the Open Source Initiative in 1998 to redefine the existing development model with this new term. In doing so, they attempted to play down the moral and political associations that free software was bundled with and instead to emphasize the practical advantages of the development model (Kelty, 2004, p.

15 Source: https://www.gnu.org/software/hurd/history.html.

503).[16][17] Since then, there has been a spread of business models revolving around open source software.

But the significance of FOSS goes beyond successful software development projects. Broadly speaking, Kelty grasps Free software as a movement with several defining characteristics: sharing source code, emphasizing and conceptualizing openness, using copyleft licenses and collaborative practices (Kelty, 2008, p. 14). Within those loose boundaries, there are varying practices that can be deemed conventional or experimental. This leads Kelty to postulate a system of thresholds discovered by collective experimentation within the movement. However, sometimes the experiments consist in attempts to apply conventional FOSS practices to other areas of production. In these cases, Kelty speaks of "modulations" of FOSS practices (Kelty, 2008, p. 16). As Kelty notes, FOSS values and practices have spread to or inspired other realms of life in recent years (Kelty, 2008, p. 2). These include hardware design and manufacture (Open Hardware[18]), science (Open Access[19]), media (Creative Commons[20]), knowledge management (Wikipedia[21]), visual arts (Processing language[22]) or even ecological engineering (Open Source Ecology[23]). What all these initiatives have in common is, according to Kelty, that they use Internet as a key infrastructural element while attempting to reorient knowledge and power (Kelty, 2008, p. 16). Taken together, this historical development exemplifies the "cultural significance of free software" (which is the

16 At that time, this also meant going against a widely influential premise formulated by Fred Brooks. In his book, *The Mythical Man-Month*, Brooks argued that: "Cost does indeed vary as the product of the number of men and the number of months. Progress does not. Hence the man-month as a unit for measuring the size of a job is a dangerous and deceptive myth. It implies that men and months are interchangeable. Men and months are interchangeable commodities only when a task can be partitioned among many workers with no communication among them (…). This is true of reaping wheat or picking cotton; it is not even approximately true of systems programming." (Brooks, 1995, p. 16) From this argument, Brooks deduced the famous Brooks' Law: "Adding manpower to a late software project makes it later." (Brooks, 1995, p. 25) After formulating the premise, Brooks argued that, for teams of software developers, the only organizational form which can assure efficiency and conceptual integrity is the one modeled after surgical teams where problem solving is reserved for one person while everyone else provides necessary support (Brooks, 1995, p. 32). However, the FOSS software development model is based on exactly opposite model of organization involving volunteer association, work self-assignment and occasional contributions. Therefore, at that time, the FOSS model existed as an unexplained alternative.

17 In this text I use the expression "free and open source software" (FOSS) to denote both branches of the movement represented by the Free Software Foundation and the Open Source Initiative respectively. I can allow myself to amalgamate the branches, because most of the time, I am not concerned with their value differences (and where I am, I differentiate between them), but with practices associated with them, which, as Kelty notes, are common: "for all the ideological distinctions at the level of discourse, they are doing exactly the same thing at the level of practice" (Kelty, 2008, p. 14).

18 http://www.ohwr.org

19 http://www.doaj.org

20 http://www.creativecommons.org

21 http://www.wikipedia.org

22 http://www.processing.org

23 http://www.opensourceecology.org

subtitle of Kelty’s book).

Inside the FOSS movement, the predominant personal identity is that of a hacker.[24] The corresponding verb “hacking” designates work that, as Pekka Himanen claims, is tied to a specific ethic. Himanen describes the hacker work ethic as being primarily based on passion and opposed to what has been with reference to Max Weber’s work called the Protestant work ethic (Himanen, Castells, & Torvalds, 2001, p. 6). Work is seen by hackers as intrinsically interesting, inspiring, and joyous. On the other hand, Protestant ethic perceives work as a calling – work is a duty which is an end in itself and must be done as well as possible. This rendering of the hacker work ethic resonates with several later studies which emphasize the key role of intrinsic motivation for volunteer involvement. For example, one of the findings in a study by Sonali Shah is that while initial contributions to software development projects often serve to satisfy a need for improved functionality of the software, many of those who stay involved do so because they enjoy the work (Shah, 2006, p. 1010). Correspondingly, Stephanie Freeman claims that although life situation of contributors vary widely, the commonality is that the boundary between work and hobby is blurred in their involvement (Freeman, 2007, p. 73). Furthermore, Margit Osterloh and Sandra Rota identify two institutional preconditions for establishing intrinsic motivation within FOSS projects: enabled self-determination and conditional cooperation (contributing when others are too) (Osterloh & Rota, 2004, p. 291–292). Osterloh and Rota further claim that intrinsic motivation of project members translates into trustworthiness of the project for those outside of it (Osterloh & Rota, 2004, p. 296).

But motivation is not the only determinant of participation in FOSS projects. The demographic characteristics of a typical contributor, as summarized by Söderberg show that FOSS projects are populated mainly by “middle-class males living in the West” (Söderberg, 2008, p. 28).[25] This situation has, according to the author, its origins in the early days of computing when access to computers was highly restricted. However, these restrictions have been considerably lowered as the prices of computers declined. Currently, Söderberg sees the

24 In this context, the term “hacker” has a positive connotation and denotes someone who cleverly takes advantage of a formal or automated system. However, this does not necessarily involve criminal activities. To differentiate themselves, the free and open source software hackers use the term cracking or crackers to denote those who perform hacking with criminal intent.

25 A more detailed, though older, summary is offered by Holtgrewe, who draws on the surveys by Ghosh et al. (2002) and Lakhani et al. (2002): “developers are youngish with an average age below 30 years. They are almost exclusively male (98 – 99%). 60 – 70% are university or college graduates, 20 – 30% are students. Around 80% are IT professionals, which leaves roughly a fifth of amateurs in the sense that they have nothing to do with the IT industry” (Holtgrewe, 2004, p. 10).

main cost in the amounts of leisure time that need to be spent in order to contribute to a project (Söderberg, 2008, p. 28) – a resource that is distributed along different lines than wealth and that, as a result, privileges aggregates such as students or the unemployed.

It is perhaps not surprising that there is a significant gender imbalance in FOSS projects. But this, according to Söderberg, cannot be explained by an active struggle for economic resources, as many of the projects are predominantly volunteer oriented.[26] On the other hand, there is not much preference, across FOSS projects, to actively seek and support joining of underrepresented groups. The projects are declaratively and practically open, but the emphasis placed on meritocracy leads to the position that it is up to the underrepresented groups to exert effort and join the activities (Söderberg, 2008, p. 29).[27]

This leads us back to the topic of values forming the hacker ethic. As such, the hacker ethic can be summarized as adhering to seven basic values (Himanen et al., 2001, p. 139–141):

1. passion – hackers work on tasks intrinsically interesting for them and enjoy their realization
2. freedom – hackers organize their life around creative work and other passions, they oppose routine and monotonous work
3. social worth – hackers are aiming to create something valuable and be recognized for that by their peers
4. openness – hackers allow further usage, development or testing of their creations by anyone
5. activity – hackers prefer active pursuit of passion over passive receptiveness
6. caring – hackers perceive concern for others as an end in itself
7. creativity – hackers respect the imaginative use of abilities and providing new and original contributions.

Some of these values can be clustered into more general areas of conduct. According to Himanen (2001, p. 140), the values of passion and freedom constitute the hacker work ethic, the values of social worth and openness form the hacker money ethic and the values of

26 Again, a more detailed description can be found in Holtgrewe (2004, p. 10): “Between half and 80% of FS/OS developers are volunteers. For the majority, involvement is limited to the extent of a more or less time-consuming hobby. Roughly two thirds of developers spend less than 10 hours per week on FS/OS development.”

27 Although there are exceptions, such as the Outreachy program originating from the activities that took place already in 2006 under the patronage of the GNOME Foundation. The project’s website can be found at: https://www.gnome.org/outreachy/

activity and caring serve as a basis for hacker network ethic, or “nethic”, while creativity, the seventh value, permeates all of these areas. With reference to concrete activities, Katherine Stewart and Sanjay Gosain found four types of values present in F/OSS communities: collaborative values (helping, sharing, cooperation), individual values (learning, technical knowledge, reputation), process values (bug fixing, code quality, status attainment) and freedom values (free information, free software) (Stewart & Gosain, 2006, p. 303). Based on their research, these authors claim that in most cases, these values have positive impact on trust and communication quality (Stewart & Gosain, 2006, p. 303), which means that they are functional with regard to community building and technical performance.

Overall, the hacker ethic seems to imply the abolition of the distinction between work and leisure. Meaning cannot arise from duty bound work or unproductive leisure, it can be found only in the intrinsic value of an activity which an individual is passionate about (Himanen et al., 2001, p. 151). These values are, of course, not uniformly applicable to the movement as a whole so that the actions of every member would be generally determined by them. There are significant differences, most notably for example between the adherents of free software or open source software. But these values indicate the overall spirit the movement as such represents.

We can see that the norms are predominantly concerned with regulating the process of software development. This is indicative of what Kelty emphasizes by calling Free software a recursive public, that is, a public “that is vitally concerned with the material and practical maintenance and modification of the technical, legal, practical, and conceptual means of its own existence as a public; it is … capable of speaking to existing forms of power through the production of actually existing alternatives” (Kelty, 2008, p. 3). In other words, this type of public is able to develop and deploy its own infrastructure, be it technical, legal, or conceptual. Because of this, it can enjoy a significant amount of independence.

In FOSS, the technical recursivity is achieved primarily by preference of FOSS software to be used as tools. Therefore, FOSS projects build on existing FOSS software to develop new programs. As Matt Ratto notes, in terms of FOSS development projects, there is a difference between software as a compiled tool and software as object of work (Ratto, 2007, p. 96). But considering the distinction between mutable and immutable mobiles, introduced to the FOSS studies by Mary Darking and Edgar Whitley (Darking & Whitley, 2007, p. 24), the developed software (object of work) and the used software (tools) differ in the nature of their presence in

FOSS projects. While software tools could be characterized as immutable mobiles which maintain their shape despite the configuration of relations they enter, the developed software could be characterized as a mutable mobile – unstable, situation dependent, or even “fluid” object. This is so because all the information and knowledge necessary to meddle with the developed software is actually and readily available in a given project, while for software used as tool (and developed at another place) this is available only as a potentiality.

The distinction between hacker and protestant ethics seems also to be historically embedded in different spheres of life. Himanen (2001, p. 6) argues, that the historical precursor of hacker work ethic was the work ethic employed in the antique academia with its intrinsic interest in knowledge, search for inspiration and joy of discovery. Protestant work ethic is supposed to have its precursor in work ethic present in medieval monastery and its emphasis of duty fulfillment. The Protestant work ethic, as shown by Weber, was eventually embraced by capitalism, which stripped it from the religious context and preserved the emphasis on duty fulfillment. As the hacker work ethic is fundamentally different from the Protestant work ethic, Himanen (2001, p. 12) argues that its existence and spread poses a challenge for the present-day capitalism.

This is why the hacker work ethic is significant and worthwhile, but it still needs to be elaborated in more detail. First, it is a work ethic so it must be distinguished from the utopian images of life without doing anything. The hacker work ethic is characteristic for preference of tasks, which are found to be interesting, inspiring and for the completion of which the hacker is willing even to go through not so joyful parts (Himanen et al., 2001, p. 19). Furthermore, the hacker ethic involves the belief that the use and optimization of machines should lead to a less routine and machinelike human life. There is an implied emphasis on creativity which can not flourish under the conditions of time pressure and monotonous tasks. Work is seen as a part of continuously ongoing life and workers as multi-dimensional human beings. In this way, the hacker ethic constitutes the image of a worthy life (Himanen et al., 2001, p. 39).

The hacker ethic also emphasizes openness through information-sharing (Himanen et al., 2001, p. 39). This goes together rather well with what is considered the prevalent motivation force – peer recognition. Only when the results of one’s work are traceable and widely accessible can peer recognition work. This characteristic has the potential to collide in certain cases with the concept of ownership which forms the basis of capitalism. This issue is further explored by Gabriella Coleman. She claims that the emphasis put by hackers on making the results of their work available not only to themselves, but also to anyone interested, is evocative of Karl Marx’s critique of estranged labor (Coleman, 2013, p. 13). However, hackers do not follow the line of reasoning of radically leftist critique of capitalism. As Coleman shows, they establish their critique by playing one aspect of liberalism against another by claiming that source code should be associated with freedoms related to speech, not with those related to private property (Coleman, 2013, p. 6). Hence, the central value expressed by Coleman as “code is speech” (Coleman, 2009, 2013, p. 147). This is also reflected in a saying hackers developed to distinguish between the two kinds of freedom: free as in speech/free as in beer.

Johan Söderberg goes as far as claiming that the hacker movement is a part of a broader revolt against commodification of labor (Söderberg, 2008, p. 44) and a continuation of the labor struggle (Dafermos & Söderberg, 2009). In recent years, software development contributed significantly to deskilling workforce in many occupations. And the routinisation is paradoxically starting to affect also the professions related to software development itself. However, Söderberg argues that the knowledge workers have a specific position in the struggle (Söderberg, 2008, p. 46). They can either engage in hacking, that is, using their skills to build viable alternatives, or resort to cracking – using their skills for conducting actions of resistance that could be considered illegal. According to Söderberg, this resistance cannot be undermined by deskilling the workforce – a strategy that could have been applied everywhere else: “At this point, however, Taylorism runs into its own limits. There is no easy way to deprive ‘knowledge workers’ of knowledge and still have them working” (Söderberg, 2008, p. 46).

However, the hacker ethic can not be considered wholly anti-capitalistic. It does not oppose the idea of making profit, it opposes the idea of making profit by constraining specific kinds of information. In fact, there is significant involvement of private businesses in FOSS projects that are strategically positioned. In this regard, Joel West and Siobhán O’Mahony distinguish between autonomous and sponsored communities. These authors claim that the licensing and access to the source code are provided in the same way by both types of communities. They differ, however, in that the governance is more pluralistic in autonomous communities while in sponsored communities, the control exerted by the sponsor prevails. This is outweighed by the assurance of continued existence – autonomous communities that don’t attract volunteers cease to exist. In sponsored communities, the core developers are usually employed by the sponsor – which safeguards continuity (West & O’Mahony, 2008, p. 14–15). On the other hand, companies may benefit from involvement with a community by extending their resource base, as Linus Dahlander and Mats Magnusson suggest in general terms (Dahlander & Magnusson, 2008, p. 638).

Moreover, by combining business and community involvement, sponsored communities are placed at the intersection of formal and informal economies. But in this case, the informality is not associated with downgraded labor, breaching a link that Manuel Castells made in one of his older works (Castells & Portes, 1989, p. 26). Although it may share the status of being undeclared or invisible, which is the case, as Bonnie Nardi and Yrjö Engeström (1999) show, with much of the work in the postindustrial society. In this sense, autonomous FOSS projects may represent a specific kind of informal economy.

Correspondingly, some authors claim that FOSS projects are structured according to a symbolic economy of their own. According to Magnus Bergquist and Jan Ljungberg, the economy is based on gift giving (Bergquist & Ljungberg, 2001, p. 312).[28] Here, software developers are seen as gift givers to those that accept the gifts – users. This constitutes a relationship where software developers gain power by systematically giving away the results of their work. The only way for users to even up their position is to give back by contributing. But this is not only a matter of decision. The presence of peer review for contributions means that the current developers assess and select contributions to be used and therefore, in a sense, select users who will be allowed to give back (Bergquist & Ljungberg, 2001, p. 314). However, as Bergquist and Ljungberrg point out, this relationship works only when the parties share a framework of meaning (e.g. the users know that the software they use was developed by volunteers and appreciate it) (Bergquist & Ljungberg, 2001, p. 314).

Focusing on the developer side of the relationship, we can find more elaborate status hierarchies explored by Daniel Stewart. This author claims that status is based on references that other members give and that in FOSS communities, it is largely based on reciprocity and collaboration (Stewart, 2005, p. 834). In other words, developers tend to give references for those they work with and also give references back when they receive some. Furthermore, as

28 Alternatively, one could construct a similar explanation along the lines of spending attention. The conceptualization of attention as a resource was done already by Herbert Simon (Simon, 1971, p. 40) and the relation between attention economy and the Internet was drawn later by Michael Goldhaber (Goldhaber, 2006; Goldhaber, 1997) or Philippe Aigrain (Aigrain, 1997). However, the concept of attention is too general and underspecified in the sociological or anthropological traditions of thought and so it can’t be readily used.

Stewart argues, the references form a self-reinforcing cycle, which means that the more references of a sort a members receives, the smaller is the probability of receiving references that counter the previous ones (Stewart, 2005, p. 835). However, the most interesting point made by Stewart is his identification of peer evaluation as predominantly endogenous (Stewart, 2005, p. 838). This means that status is derived mostly from endorsement of work undertaken within the community and external forces are not taken taken into account (at least not directly).[29] This makes the community embedded in its own rules but it also provides foundations for its compatibility with a broad range of organizations and worldviews.

Part of the hacker ethic is also its perception of authority. In this way the hacker ethic once again resembles the academic, because one of its key components is that anyone can use, criticize, or develop the objects produced by other hackers (Himanen et al., 2001, p. 68).[30] It is this model of open development and self-correction that is perceived as desirable in contrast with models that keep knowledge constrained and goals authoritatively set. However, this does not mean that the hacker work ethic asserts absence of any kind of structures (Himanen et al., 2001, p. 72). As we have seen, there are structures at least in terms of status hierarchies, but browsing the studies published during the last decade reveals more.

Siobhán O'Mahony and Fabrizio Ferraro studied the process of governance establishment in the Debian Linux distribution. These authors found out that in the long run, the community preferred leaders with organizational competence over ones with purely technical (O'Mahony & Ferraro, 2007, p. 1100). This provides a correction for the description of the status building process – status does not have to be based on endorsement of technical work only. Furthermore, even though the community initially placed many checks on the power of elected leaders, eventually, those that broadened their sphere of influence were preferred (O'Mahony & Ferraro, 2007, p. 1100).

With regard to power and influence, Didier Demaziére et al. distinguish between

29 This tendency can be observed also in other online constituted communities such as Wikipedia (for example, with its restrictions on original research and not taking into account the professional researcher status of some of its contributors (Luyt, 2011, p. 1063; Rosenzweig, 2006, p. 140)), or as John Shiga notes in the case of mash-up communities (Shiga, 2007, p. 97).

30 In this sense, Eric Raymond, the author of a highly influential essay *The Cathedral and the Bazaar* points out that the already mentioned Brooks' Law (a premise, well known among software developers, implying that adding developers to a late software project makes it later) needs to be corrected with the concept of egoless programming: "Gerald Weinberg's classic The Psychology Of Computer Programming supplied what, in hindsight, we can see as a vital correction to Brooks. In his discussion of 'egoless programming', Weinberg observed that in shops where developers are not territorial about their code, and encourage other people to look for bugs and potential improvements in it, improvement happens dramatically faster than elsewhere." (Raymond, 1999, p. 39).

centralized control – which ensures a consistent performance over time and serves as a guarantee that time invested by volunteers will not be lost – and distributed regulation – influence distributed according to the presence (number of contributions or amount of time invested) of individuals in the area (Demazière, Horn, & Zune, 2007, p. 51). These findings point to the fact that power and influence are dispersed among individuals with high levels of involvement – a fact that is used by the communities to label themselves as meritocratic.

However, as Nicolas Ducheneaut shows in his analysis inspired by Bruno Latour, FOSS projects recede from the ideals of openness and meritocracy in situations when newcomers are attempting to join and influence a project. Ducheneaut identified a series of stages that a newcomer goes through – from passive monitoring of the development activities to making substantial modifications to the developed software.[31] But as he notes, most newcomers stop at the initial stages and very few of them reach the advanced ones. Ducheneaut attributes this to several characteristics of FOSS projects. According to this author, FOSS projects represent black-boxes for newcomers – they need to uncover the relationships forming the project network in order to identify how they can interact with it and where could they start with their contribution. Furthermore, Ducheneaut claims that the network naturally resists change, which means that a newcomer has to mobilize human and non-human allies in order to insert himself into a position from which he can make a substantial modification (Ducheneaut, 2005, p. 353–355).

Such situations typically lead newcomers to perform what could be called autonomous learning. In this process newcomers make use of the available sources of information – which in FOSS projects are abundant and which, according to Andrea Hemetsberger and Christian Reinhardt enable re-experiencing rationales and past events. These authors argue that the archived traces left after past interactions (for example in mailing lists) combined with information sources specifically aimed at newcomers (user or developer documentation) form a transactive memory that can be explored independently of the actors that created it (Hemetsberger & Reinhardt, 2006, p. 195–199).[32] This phenomenon is further supported by the self-documenting tendencies in the FOSS culture. Hackers usually produce accounts (typically in the form of blog posts) of the learning processes that they undergo. These records

31 Drawing on Ducheneaut's work Israr Qureshi and Yulin Fang later developed a model of four classes of "joiners" differentiated according to the volume of interaction with core developers (Qureshi & Fang, 2010, p. 223).

32 Already in 2003, Gwendolyn Lee and Robert Cole pointed out that the reuse of mailing list communication and peer review observation is at the heart of the knowledge generating processes in Linux kernel development (Lee & Cole, 2003, p. 644).

are publicly available so that others can make use of them in or develop them further. This leads to a continuous creation and re-creation of learning resources for any topic that is deemed to be interesting or worthwhile. In this sense, learning of one individual can teach others.

Similarly to learning, work in FOSS projects exhibits an emphasis on autonomy. Kevin Crowston et al. found that self-assignment is the most frequent type of work assignment in FOSS projects (Crowston, Li, Wei, Eseryel, & Howison, 2007, p. 6). This finding is further supported by Giampaolo Garzarelli and Ricardo Fontanella, who additionally clarify, that self-assignment is made possible by the modular architecture of the projects, allowing for individuals to work in parallel (Garzarelli & Fontanella, 2011, p. 930–936). Keeping in mind that FOSS projects are often run by volunteers, so that there is little or no leverage to enforce work assignment, this should not come as a surprise. Overall, Crowston et al. characterize the FOSS projects as “self-organized” and compare the coordination mechanisms to those identified by Karin Knorr–Cetina in high energy physics (Crowston et al., 2007, p. 11).[33] Athina Karatzogianni and George Michaelides further characterize self-organized FOSS projects by noting that they typically exhibit a two-tier (core and periphery) structure distinguishing maintainers and occasional contributors and that the overall distribution of the projects follows power law (frequency of an event is inversely proportional to its magnitude), which translates to the fact that there are few projects that attract large numbers of developers while there are many projects that attract only a small number of developers (Karatzogianni & Michaelides, 2009, p. 148–149).

The two-tier structure observed by Karatzogianni and Michaelides points to the issue of participation inequality noted by several authors (Holtgrewe, 2004; Krishnamurthy, 2002; Kuk, 2006; McInerney, 2009). The study by George Kuk is of particular interest here, because it links participation inequality with knowledge sharing. This author claims that besides collaborative interactions, FOSS developers also perform epistemic interactions, that is, place inquiries on each others knowledge. However, these inquiries can be demanding and so they may easily turn from exploration to exploitation. Kuk’s point then is that participation inequality is functional in that it reduces the load of epistemic interaction by restricting it to a

33 Knorr–Cetina herself characterizes self-organization in the following way: “Self-organization, in turn, keeps social relations liquid (and presupposes their liquidity): there is the fluidity of everyone’s readiness to become drawn into temporary engagements with others in voluntaristic collaborations, a fluidity aided by the breakup of forces of individuation and the holistic competence of individuals trained in object circuits.” (Cetina, 1999, p. 179) Here, self-organization consists in work self-assignment made possible by even distribution of knowledge.

narrow group of core developers (Kuk, 2006, p. 1039). Taken to a more general level, this finding supports Ursula Holtgrewe's criticism of sweeping claims that characterize the Internet as an "undifferentiated mass of simultaneous and arbitrary information" (Holtgrewe, 2004, p. 14). Indeed, Holtgrewe points to FOSS development to demonstrate that meaningful action is not drowned in the abundance of digital information.

Moreover, Kuk's claim is consistent with the finding of Georg von Krogh et al., who claim that core developers usually avoid narrow specialization (Von Krogh, Spaeth, & Lakhani, 2003, p. 1230). Their activity spread across number of modules then requires exploration with epistemic interaction as one of its forms. We can expect higher levels of socialization present by occasional contributors as their motivations and actions are aligned with the project only in certain respects. However, these specialized occasional contributions do not cause disturbances in the development process. As Hemetsberger and Reinhardt point out, FOSS projects are able to integrate individual actions with their overall activities, regardless of their nature as general maintenance or specialized contribution. Therefore, these authors characterize FOSS projects as "coat-tailing work systems" (Hemetsberger & Reinhardt, 2009, p. 1003).

On a more general level, Yochai Benkler includes free and open source software development under the umbrella of new forms of peer production. This author claims that the defining characteristics of these forms of production stem from reduced transaction costs.[34] Benkler argues that market transactions differ from non-market social exchange (such as gift-giving) in that the calculations or definitions (calculating prices or drafting agreements, for example) necessary for market transactions place significant burden on all parties. However, non-market social exchange is exempted from these costs because it does not involve explicit calculation or definition (Benkler, 2004, p. 307, 2006, p. 109). These non-market exchanges traditionally reached only a scope of locally and temporally restricted interactions but the advancements made in information technologies achieved in the past decades made possible the rise of what Benkler calls "effective, large-scale cooperative efforts – peer production of information, knowledge, and culture" (Benkler, 2006, p. 5). Given that this new form of production is based on the non-market social exchange, Benkler claims that it not only has a systemic advantage in the form of reduced transaction costs or better allocation and motivation of workforce, but that it also improves the practical capacities of individuals by

34 In this regard, Benkler bases his argumentation on the classical work of Ronald Coase (Coase, 1937, 1960).

opening to them a broader scope of production activities without the restrictions placed by traditional models involving price calculation or strict hierarchical organization (Benkler, 2002, p. 376, 2006, p. 8).

This optimism is intensified in some of the works surrounding the concept of collective intelligence, which has been developed at least since the early 1970s (Wechsler, 1971). Collective intelligence can be defined as the “ability of virtual communities to leverage the combined expertise of their members” (Jenkins in Uspenski, 2013, p. 142). While Uspenski proceeds to a heuristically inspiring distinction between collective intelligence (based on mutual evaluation of meaning) and mass intelligence (based on aggregation of data) (Uspenski, 2013, p. 148), the rest of the discourse is interwoven with utopistic visions of future. For example, Pierre Lévy foretells the coming of planet-wide civilization through collective intelligence based in cyberspace and proceeds further to claim that television will be replaced by omnivision, allowing all humans to watch any place at any time (Lévy, 2005, p. 189, 191). In his most known work, Lévy claims that the historical development which the emergence of cyberspace has triggered implies “a new humanism”, one that promotes individual intelligence to a collective level. From this, “new forms of democracy, better suited to the complexity of contemporary problems than conventional forms of representation, could then come into being” (Lévy & Bonomo, 1999, p. 18). Other authors associate the term collective intelligence with images of “harnessing crowds” (Malone, Laubacher, & Dellarocas, 2010) or “creating a prosperous world at peace” (Tovey, 2008) directly in the headings of their works.[35] This demonstrates the positive valence with which the terms “collective intelligence” or “cyberspace” are charged.

Furthermore, such claims can be also found in constitutive texts of the FOSS movement. The essay *The Cathedral and the Bazaar* from Eric Raymond can serve as a good example:

> That is, that while coding remains an essentially solitary activity, the really great hacks come from harnessing the attention and brainpower of entire communities. (Raymond, 1999, p. 39)
>
> The Linux world behaves in many respects like a free market or an ecology, a collection of selfish agents attempting to maximize utility which in the process produces a self-

35 Furthermore, Francis Heylighen, former physicist who is attempting to formulate a general model of the Internet as a system of collective intelligence (Heylighen, 1999; Heylighen & Bollen, 1996; Heylighen, Heath, & Van, 2004), explicitly formulates his utopistic vision in that the Internet has introduced a cognitive system on a planetary level, a global brain, and that this super-organism can be conceived as a higher level in human evolution (Heylighen, 2002, p. 2).

> correcting spontaneous order more elaborate and efficient than any amount of central planning could have achieved. (Raymond, 1999, p. 40)

The basic claim made by Raymond in the text is that taking advantage of the Internet infrastructure and employing a certain set of cooperative customs leads to establishment of spontaneous order which is more efficient than central planning in that it allows to harness the brainpower of entire communities.[36] Additionally, for Jan Ljungberg, the specific ways of knowledge sharing and work coordination signify the forms of organization of the future (Ljungberg, 2000), while for Georg von Krogh and Eric von Hippel, free and open source software development represent a new model of innovation that should spread to other fields of production (Von Krogh & Von Hippel, 2006, p. 982). Finally, Cory Ondrejka stays within the boundaries of cyberspace and elaborates upon the possibilities of establishing "metaverse", an alternative reality of unmatched complexity (Ondrejka, 2004, p. 81).

These visions are symptomatic of the enthusiastic anticipations[37] of what cyberspace can offer, which are placed under elaborate criticism by David Hakken.[38] Hakken argues that these anticipations are based on the assumption that knowledge has replaced capital, labor and natural resources as central productive forces (Hakken, 2003, p. 9). Indeed, Lévy starts his book on collective intelligence with a claim that the "prosperity of a nation, geographical region, business, or individual regions depends on their ability to navigate the knowledge space" while "power is now conferred through optimal management of knowledge" (Lévy & Bonomo, 1999, p. 1). Correspondingly, Hakken argues that there is a broader tendency of uncritically accepting the "knowledge society" label:

> Rather than carefully articulating their view of the proper way to conceptualize the knowledge revolution and then going on to make their case for it, most performers merely jump on a generally conceded "knowledge society" bandwagon. (Hakken, 2003, p. 9)

As Hakken demonstrates, this tendency is also present in the works prominent authors such as that of Karin Knorr-Cetina (Hakken, 2003, p. 9). Furthermore, if we look at the notable work of Manuel Castells, we can identify the tendency (provided we acknowledge the link between

36 The contraposition of "free market" and "spontaneous order" against "central planning" evidently hints at certain positioning within a political spectrum. However, leading the analysis in this direction would diverge the current text from it purpose.

37 To be sure, there are also pessimistic expectations with regard to cyberspace, as, for example, Mark Davis (Davis, 2013, p. 162) shows. But it seems that these did not gain such momentum with regard to claims about the discontinuity of knowledge-related processes introduced by cyberspace.

38 The utopistic tendencies in Lévy's work were critically noted also by László Fekete (Fekete, 2006, p. 742). A more systematic account of the discussion about the utopistic tendencies surrounding digital technologies can be found in a dissertation thesis written by Jakub Macek (2011, p. 84–93).

mind, cognitive processes and knowledge) in his more radical claims such as the one that: “for the first time in history, the human mind is a direct productive force, not just a decisive element of the production system” (Castells, 2010a, p. 31). While this claim was made in the first part of a first volume of Castells’ trilogy (Castells, 2010a, 2010b, 2010c) on the information age, in the concluding part of the third volume, we can find the following statement:

> The promise of the Information Age is the unleashing of unprecedented productive capacity by the power of the mind. I think, therefore I produce. In so doing, we will have the leisure to experiment with spirituality, and the opportunity of reconciliation with nature, without sacrificing the material well-being of our children. The dream of the Enlightenment, that reason and science would solve the problems of humankind, is within reach. Yet there is an extraordinary gap between our technological overdevelopment and our social underdevelopment. (Castells, 2010c, p. 395)

A notable difference between the two statements is that the former constitutes a claim about the actual state of labor organization while the latter represents an expression of a potential state that could be reached. This ambiguity is further supplemented by claims of discontinuity: “I do believe that there is a new world emerging at this turn of millennium. In the three volumes of this book I have tried to provide information and ideas in support of this statement.” (Castells, 2010c, p. 372) As we will shortly see, these characteristics are typical for a speech mode that is closely related to the utopistic visions I elaborated earlier.

Hakken identifies a speech mode, which he calls the “optative form”, that is indicative of the sweeping claims emphasizing discontinuity in their images of cyberspace.[39] According to this author, the optative form’s predominant characteristic is that it mixes statements about what is and what is hoped to be, it blurs the distinction between present and future (Hakken, 2003, p. 27). In other words, when using the optative form, authors see the future potential of things as their essence. This corresponds to the initial definition that Rob Shields uses in his elaborate work on the term “virtual” – “that which is so in essence but not actually so” (Shields, 2003, p. 2). However, Shields is also wary of the uncritical acceptance of the expectations surrounding the virtual:

> The hype around digital virtuality over the past decade has been more about myth and less about actual cyberspaces. As a fad and myth, virtualism is itself virtual. Symptoms of

39 Further critique of the positions emphasizing discontinuity can be found in the work of Steve Woolgar (2002, p. 17) or Marylin Strathern (2002, p. 311). Both authors claim that the dichotomy between the virtual and the actual (or “real”) does not constitute a mutually exclusive binary opposition. On the contrary, they argue that the virtual and the actual are mutually co-extensive.

virtualism include exaggerated expectations of anything described as ‘virtual’, and unrealistic expectations that digital technologies will solve social problems. The boom in technology stocks and enthusiasm for virtual reality hinted at the ongoing expectations of the virtual. In line with its historical definitions, it carries a certain promise of positive potential or virtue. Portrayed as enabling a human virtuosity beyond the limits of the body or gravity, the legacy of the baroque echos through the claims of Silicon Valley entrepreneurs. (Shields, 2003, p. 15)

Already in his older work, Hakken claims that the images around the so-called computer revolution should be bracketed as a myth in the anthropological sense (Hakken, 1999, p. 18). The logical implication of this position is a call, made by Hakken, to examine in more detail the knowledge-related processes taking place in the cyberspace (Hakken, 2003, p. 29). This work aims to answer this call in a specific sense: informed by this critique, the aim of this work is to explore closely the processes of knowledge production in free and open source software development, an area of practice which Benkler deemed to be typifying the new form of cyberspace enabled peer production. For this purpose, Hakken offers three ways of conceptualizing the knowledge related changes taking place in cyberspace:

1. Quantitative growth in knowledge,
2. Change in its quality/character, or
3. Change in its social functions, perhaps involving quantity, quality, or both. (Hakken, 2003, p. 29)

The present work focuses on the third option by attempting to show how the social function of knowledge is altered in the environment of FOSS projects by its interrelatedness with practicing of rights that were traditionally associated with ownership. Furthermore, I try to show the knowledge-related limitations of the images, associated with utopian virtualism, of frictionless interaction of individuals spontaneously emerging around problems to solve them. These claims are supported by an elaborate analysis of mediation and resource flows that take place inside a FOSS project. This whole construction stands on a basis formed by thick description of selected incidents I recorded during my participant observation and other involvement in the field.

2. Software and Knowledge

Software is a general designation for the sum of all programs that can be run on a computer. It represents one side of the software/hardware distinction where hardware designates tangible computer components on which software operates. It is common to say that software consists of ones and zeros, that is, of digital information. However, ones and zeros represent the end product that is readable only for machines. When developing software, programmers are not dealing with ones and zeros, they use one of many programming languages to produce a strictly formalized text – the source code. After the source code of a program is written, it is turned into a machine readable binary file consisting of ones and zeros through an act called compilation. Compilation represents an event in which readable and modifiable text is transformed into a solid thing that behaves according to its own logic. It is the act of materialization of an object.

What interests me, however, is what happens before compilation: the process in which humans and nonhumans are organized in a manner that results in an object that can be executed and purposefully utilized by its users. That is to say, I am interested in associational processes that take place during software development. As claimed by Jacob Nørbjerg and Philip Kraft, software production typically involves a "complex mix" of organizational structures, work practices or even politics (Nørbjerg & Kraft, 2002, p. 218).[40] From this perspective, software is relevantly defined by Arne Raeithel:

> Computer science or informatics appears in this perspective as one of the sciences of human self-regulation, mainly concerned with electronic and virtual machines used in this process. Software objects may consequently be seen as predefined constraining contexts ('forms') for sign processes (semioses) mediating between human actors, while at the same time presenting virtual objects and instruments ('means') for self-determined use by the cooperating persons. (Raeithel, 1992, p. 391).

There are two important points in this definition. First, software is seen as a constraining context, a digital environment which determines the options its users have. Any action the user can take has to be pre-conceived by software developers and implemented in a given program. Second, software represents an instrument, a tool that can be used for purposes its developers have not envisioned and that can be combined with other tools to produce unexpected results.[41]

40 This is why ethnography is suggested by several authors for studying it (Dittrich, 2002; Klischewski, 2002; Rönkkö, 2002; Westrup, 2002).

41 These two points can be found also in Christiane Floyd's work (Floyd, 1992, p. 15). The common point is

Software development is carried out by a practice called programming. According to Peter Naur, programming is “matching some significant part and aspect of an activity in the real world to the formal symbol manipulation that can be done by a program running on a computer” (Naur, 1985).[42] However, Naur further conditions programming by knowledge building: a programmer must craft a theory of what is to be matched and in what way.[43] Theory, in this sense, represents a support for action not limited only to know-how, but including also explanations and justifications of what is to be done. The programmer has to be able to explain how the important characteristics of the real world activities are mapped into the program and subsequently justify his decision and choices. In this way, the knowledge needed for software development transcends what is recorded in the final product. This knowledge is needed for every modification of a program in order for those modifications to form an integrated whole (Naur, 1985).

Naur further elaborates a model of three phases a program can go through. First, program life designates a phase when a team of programmers actively develops a program, providing all modifications. Second, program death represents a phase in which the program is no longer actively developed by the team. Third, program revival marks a phase in which a new team of developers take up the development and tries to reconstruct the theory with which the program was originally developed. Naur’s main point is that the theory can be maintained only when the new developers get the chance to work with the original developers and receive the theory from them. According to Naur, “reestablishing the theory of a program merely from the documentation, is strictly impossible” (Naur, 1985). I will further explore this issue as a theoretical problem of decontextualization of knowledge.

In a similar way, Pelle Ehn describes software development as a process of designing a computer artifact. According to this author, design is an activity and a form of knowledge in which artifacts and their use are anticipated and which deals with the distinction tradition/transcendence (Ehn, 1988, p. 161). This means that designing technological objects

that software, while it represents instrumental means to achieve goals, constitutes some kind of constraining context for its users.

42 According to Reinhard Keil-Slawik, a new quality emerges from this process. Sequences of activities that need to be performed to achieve a certain action can be condensed into a single object or operation. These objects and operations can then be further combined without the constraints of enforced sequentiality they previously had. As Keil-Slawik puts it: “prescriptive temporal structures are dissolved by creating physical objects and corresponding spatial structures” (Keil-Slawik, 1992, p. 182). Thus, matching real world activities does not mean mirroring them, it means distilling them into objects or operations that are not limited by the constraints their original models had.

43 In this sense, Catharina Landström et al. write about “forming an understanding” with regard to software development (Landström, Whatmore, & Lane, 2011).

is a process which draws its resources from what is currently available and attempts to overcome it. It represents a situation in which human creativity is needed in order to produce something new using everything that is already there. However, human creativity, albeit important, is hardly the only thing that counts. The technological artifacts already there play a vital role in the process.[44][45]

According to Ehn, the importance of artifacts lies in the fact that they are able to augment or replace human activity and can do so with regard to both communicative and instrumental activities (Ehn, 1988, p. 162–163). With this general characteristic, I hold that Ehn has one particular type of artifacts on his mind, the one that is commonly grouped under the label "tools". But in software development, tools too take the form of software. Therefore, we must consider developers of certain software to be simultaneously users of other software.[46] And as users, they are dependent on the interface of the software-used-as-tool as it determines the options available for exploiting the provided functionality.

Furthermore, one other type of artifacts is introduced by Ehn: "design artifacts". These can be defined as objects (for example descriptions, models, or prototypes) which mediate the design process. Ehn's characterization of this type of artifacts is worth quoting at length:

> The role of design artifacts in the language game of design is as reminders and as paradigm cases for our reflections on existing and future computer artifacts and their use. The use of design artifacts brings earlier experiences to our mind and it bends our way of thinking about the future. I think it is in this sense we should understand them as re-presentations. … I see descriptions or models as design artifacts to objectify experiences, visions, and ideas relevant for communication in the design process. … These kind of artifacts support reflection. … Another category of design artifacts is prototypes, mock-ups, scenarios with role playing, etc. They differ from descriptions and models in the sense that they also allow for involved practical experience, not just detached reflections. (Ehn, 1988, p. 169)

There are three important characteristics that Ehn attributes to design artifacts. First, he shows

44 This is consistent with how Bruno Latour connects his theory with that of Peter Sloterdijk. "Dasein ist design" says the quote in one of Latour's texts. And he further elaborates his position: "There is not the slightest chance of understanding Being once it has been cut out from the vast numbers of apparently trifling and superficial little beings that make it exist from moment to moment" (Latour, 2009, p. 139).

45 The importance of things already there is highlighted by Reinhard Keil-Slawik's claim that preservation of tools (not their construction) is what differentiates humans from animals. As he states: "This is essential, because only then does it become possible to compare a previously built tool with a new one, to communicate about tools, and to use them as a means for education" (Keil-Slawik, 1992, p. 181).

46 In the terminology used by Susan Leigh Star, software tools are boundary objects, that is, objects that have varying use or purpose depending on the location they appear in. Thus, their purpose is ambiguous in general, but clear in particular cases (Star, 2002). Specifically, the dual nature of software as a development target and as a tool is consistent with Star's claim that we can read information infrastructure as a material artifact (tool) or a trace of activities (development target) (Star, 1999, p. 387).

that representations do not have to be necessarily symbolic. In the design process, there are objects that are created specifically with the purpose to represent something, to prove a point. Second, these artifacts provide snapshots of experiences and make them intentionally reproducible so that more time can be allocated at reflection. Third, they also provide new experience through interaction with the latest version of the desired product.

Correspondingly, Mike Reay considers conscious reflection and new experience as two main sources of learning, the activity which dynamizes knowledge. Furthermore, Reay distinguishes two types of knowledge distribution. Horizontal distribution of knowledge is based on distribution of experience in space and time. Vertical distribution, on the other hand, means distribution of knowledge into conscious and unconscious layers differentiated by presence or absence of reflection. Differences in distribution of experience and reflection can lead to stable patterns of ignorance or mis-perception. The barriers leading to this “knowledge insulation” can be overcome only with mobilization of new experience or further reflection (Reay, 2010, p. 100). These processes are key for social arrangements focusing on knowledge production because they are based on constantly overcoming knowledge insulation.

In the previous paragraph I suggest that by providing some sort of experience or reflection, design artifacts are able to transmit knowledge through the process of learning. But to grasp the problem in more detail, we must differentiate knowledge and information just like, for example, Nico Stehr and Ulrich Ufer do. According to these authors, knowledge inherently involves appropriation by a knowing subject, as they put it: “Knowledge always requires some kind of attendant interpretive skills and a command of situational circumstances. In other words, the acquisition, dissemination and realization of knowledge requires an active actor.” (Stehr & Ufer, 2009, p. 9). Information, on the other hand, does not require appropriation and therefore represents something that actors simply have and can pass on whenever they want. This makes information easily transferable (Stehr & Ufer, 2009, p. 9).[47]

In a similar manner Loet Leydesdorff makes a distinction between information and meaning. According to this author, information by itself is in a state of “still-to-be-provided-with-meaning”. Meaning arises in the process of relating information to one another in a context of individual (personal meaning) or supra-individual (discourse) systems of reference.

47 Consider a map as an example. By itself, it is a piece of information that can be passed quickly from one actor to another (and even more so in a digital form). However, it takes an actor who can read all the signs, reference points and directions in order to turn the information into knowledge about location and possible courses of action.

Therefore, meaning is defined in use (Leydesdorff, 2011, p. 393–394). This is consistent with how George Herbert Mead defined meaning, provided that we think of gestures as of transmitted information:

> Meaning arises and lies within the field of the relation between the gesture of a given human organism and the subsequent behavior of this organism as indicated to another human organism by that gesture. If that gesture does so indicate to another organism the subsequent (or resultant) behavior of the given organism, then it has meaning. (Mead, 1972, p. 76)

Therefore, meaning is derived from the ways in which information is used, from action.[48] In this sense, knowledge and meaning are very closely tied as ways of making sense from information. This relationship is further explored by Doyle McCarthy who claims that knowledge is best conceived and studied as culture. This is to claim that various bodies of knowledge operate within culture, "that they contain and transmit and create cultural dispositions, meanings, and categories" (McCarthy, 1996, p. 118).[49]

Applied to the area of software development, the theoretical statements above mean that what is embedded in design artifacts is information which can be easily transmitted but which can also be easily stripped of its meaning.[50] Knowledge involves the ability to utilize information contained in objects and to give them meaning.[51] In other words, the source code of a program by itself represents only information. Knowledge arises from the ability to either use the compiled program as an end-user or read, understand and meaningfully alter the source code as a programmer. This means that knowledge does not reside either in programmer's head or the source code, but can be found only in the interplay of the two. Knowledge is not the content, it is a quality of interaction.[52] The claim, which is made by

48 It is also common to say that meaning is the result of interpretation. As Elizabeth Long shows, this process traditionally associated with reading in private can be made less opaque by explicating the "social infrastructure of reading" by which she means not only the socialization and learning processes that establish the competency of reading, but also the social base of reading groups or other associations devoted to reading, interpreting and discussing texts (Long, 1993, p. 190–191). As a result, software development projects can be seen as a specific case of reading associations.

49 Karin Korr-Cetina's detailed study of epistemic cultures in natural sciences is a case in point (Cetina, 1999).

50 Such decontextualization is documented in a study by Jessica Thurk and Gary Fine who examine how importing pieces of architectural drawings causes errors when not accompanied with the original meaning. This happens despite the standardization with which tools are designed (Thurk & Fine, 2003, p. 115–116).

51 According to Hakken, the process of distilling knowledge from information is emphasized in the modernist knowledge discourse. The other direction, production of information based on situated knowledge is emphasized in the postmodernist knowledge discourse (Hakken, 2003, p. 37–39) . Hakken further claims that these two points of emphasis form a useful dialectic, keeping the focus on both of the processes instead of just one (Hakken, 2003, p. 45). In software development projects, we can see this dialectic at work when a newcomer first appropriates information to gain knowledge and then use it to create more information for others to appropriate.

52 This claim is also supported by Keil-Slawik (1992, p. 169).

Naur, that this quality is best spread by observing (experiencing) the interaction of someone knowledgeable seems obvious. But that does not mean that it is the only way to achieve it.

The problem of transferring knowledge through its embodiment in design artifacts essentially corresponds to the classical issue of reciprocity of perspectives explored by Alfred Schütz. According to Schütz, there is an inherent difference in perspectives among actors because they 1) are located in different spatial distance from the object while experiencing different aspects of it as typical and 2) are located in different biographical situations which are projected into different purposes at hand and systems of relevances (Schütz, 1953, p. 8). These two points also constitute the underlying basis for distribution of knowledge. We can see that the knowledge with which software developers approach a program will necessarily differ from that of its users or newcomer programmers. Members of each of these three groups approach the object with a different purpose at hand, different systems of relevances and different typifications. Assuming, of course, that these three groups are homogeneous in those three regards which does not have to be the case. But let me postulate that the differences will be bigger between those three groups rather than within them. After all, they are defined by a common experience (developing, using, learning to develop) with the object.

According to Schütz, the difference in perspectives can be overcome by two types of idealizations: interchangeability of standpoints and congruency of the system of relevances. According to the first, actors take it for granted that when they change positions, they would be at the same distance from the object and see it in the same typicality as their counterparts. The second idealization implies that actors assume that the uniqueness of their and their counterpart's biographical situations can be set aside to establish common purposes at hand and a common system of relevances (Schütz, 1953, p. 8). These two idealizations show us how the differences resulting from heterogeneous distribution of knowledge can be overcome, that is to say, how reciprocity of perspectives is established.

When applied to the practice of software development we can see that the differences may be overcome to a degree to which interchangeability of standpoints is possible among current developers, new developers and users. First, interchangeability of standpoints can be achieved when current and new developers observe each other's work. This is Naur's original proposition (Naur, 1985). But there is more to this way of establishing this interchangeability. The current developers remember the beginning of their involvement and see newcomers in the light of this experience. On the other hand, new developers can read the project's archives, which, as Andrea Hemetsberger suggests, allows them to re-experience past action (Hemetsberger & Reinhardt, 2006, p. 195). But to make it easier for newcomers to find the right information current developers usually write documentation aimed at developers.

The second way to achieve it is a practice called testing in which developers assume the role of users in order to find issues in the developed software. Testing is used not only to discover hidden errors, but also to make sure that the interface is intuitive enough for users, that the meaning of offered operations is accessible just by looking at the design without the need to cooperate with the programmer who wrote it (Keil-Slawik, 1992, p. 181).[53] Should the interface fail at this, there is a safeguard in the form of user documentation which aims to explicate the meanings carried in the interface.

The third way, this time to establish congruent systems of relevance, leads, as Keil-Slawik (1992, p. 169) suggests, through using common tools. Given that tools specify, often to a great detail, how a problem is perceived and approached, or how work is done in general, using the same tools usually results in standardization of perceptions and solutions to several options. To be sure, there may be disagreements on which option is optimal, but the recognition of available options is usually widespread. And it is in the light of these standard options that systems of relevance overlap.

These are all ways that allow to establish common purposes at hand and shared systems of relevance. In doing so, they resist the loss of knowledge at points where decontextualization would usually take place. The induction of knowledge at distant places is not something effortless, it must be safeguarded by means I now roughly sketched.

As a result of this elaboration, we can see that Naur's consideration of knowledge processes involved in software development was limited. Naur was not occupied by the congruence between developers and users so his aims were different. But still, his consideration did not involve the possibility to re-experience actions.

When Naur conceptualized the three phases a computer program can go through, the Internet was not yet widespread. This is important to note because developing software in this environment introduces conditions that were not previously present and that free and open

53 To achieve this, interfaces usually consist of signs that are designed to remind users of what can be done with the program. To grasp this problem, Ehn reaches for Wittgenstein's concept of language games and claims that the signs used in an interface must fit within the language games of its users. Only then "the signs remind us of what we can do with the artifact" (Ehn, 1988, p. 164). As a result, signs are often designed as representations of the real world actions that are symbolically mapped by a programmer and implemented as functions of the program. A typical example of this approach is the image of scissors which is common for representing the cut function.

source software development takes advantage of. These possibilities are based on two characteristics of digital texts as described by Lorenzo Cantoni and Stefano Tardini. The first characteristic, *persistence*, means that every communication that is mediated by a computer leaves a physical trace making it possibly available for someone else for an unspecified amount of time (Cantoni & Tardini, 2006, p. 44). At first, this seems to apply to all communication, online or offline – even voice makes a physical trace. However, when communicating outside cyberspace, additional effort needs to be exerted (writing or other forms of recording) in order to capture the communication and make it accessible for longer periods of time. Within cyberspace such effort is not needed because all communication takes place already in the form of text (or, less frequently, in other recorded forms).

Similarly, the second characteristic means that digital texts (and other recorded forms) are *reproducible* without the need to exert effort and resources that would be needed when reproducing non-digital media (Cantoni & Tardini, 2006, p. 55). Reproduction requires only computational resources which, once acquired, are abundant.[54] Going one step further, both of the characteristics (persistence and reproducibility) are tightly interconnected with the ability of *automated manipulation*. This ability implies that digital text can be manipulated (recorded, reproduced, searched or edited) in a way that does not require direct and permanent attendance by a human operator. Automation allows human operators to specify instructions before the manipulation process which is then performed autonomously at the computer's own pace.

These characteristics of digital texts make possible what Yochai Benkler emphasizes as transparency of online culture. As an example, this author takes the case of Wikipedia[55] articles. For any given article, all changes made to it are traceable in its history while discussions leading to those changes are also recorded on a separate page (Benkler, 2006, p. 289). Such an endeavor in offline archiving would require large investments in effort and resources, making it slow and cumbersome. But in cyberspace, the archiving procedures can

54 The words "once acquired" are important here. One more important characteristic that Cantoni and Tardini list is that digital texts are directly inaccessible to human senses. This means that a computer (preferably with Internet connection) is required in order to obtain and read digital texts. As the issue of digital divide (see for example Norris, 2001) reminds us, the spread of this infrastructure is hardly universal.

55 Wikipedia, launched in 2001 and explicitly stating influence from the free software movement (see Wikipedia (2015)), can be considered what Christopher Kelty calls a modulation of Free software practices, that is, an application of one or more characteristics of free and open source software development to other areas than programming (Kelty, 2008, p. 16). Roy Rosenzweig (2006), a historian examining the implications Wikipedia has for historiography characterizes it in a similar way by posing the question: "Can history be open source?".

be automated. Originating in 1970s, free and open source software development seems to be the first organized effort to employ these possibilities systematically. As a result, the source code (and its documentation) is by far not the only set of information that is publicly available in FOSS projects. There are numerous other sources of information of which the main are: detailed history of changes made to the source code (resulting from version control systems), recorded communication among developers (located in mailing lists or Internet relay chat archives), lists of issues containing discussions on how to solve them (gathered in databases such as Bugzilla), websites with further information (project website, developer blogs). With such a wealth of information at disposal, we can hypothesize that the dependency of new developers on interacting with the original ones is lowered. Wider pool of information that can be related to one another means more options for endowing it with meaning.

2.1. Network Shaped Knowledge Distribution

If we try to sum up the role tools and design artifacts play for a programmer's knowledge building, we could come up with a wording similar to "material means of thought", an expression used by Edwin Hutchins to describe material dimension of symbol manipulation. Hutchins' line of argumentation starts with the claim that abstract manipulation of symbols is not a process that takes place inside individual's mind. Instead, symbol manipulation should be considered as mediated by cultural and physical objects (Hutchins, 1995, p. 363).[56] This is not to disprove that humans process symbolic structures, Hutchins' aim is to counter the proposition that the cognitive process is *purely symbolic* (Hutchins, 1995, p. 369–370).

Hutchins is not the only author proposing what could be called the central thesis in Theory of Distributed Cognition.[57] Andy Clark and David Chalmers also propose to award

56 The tendency to restrict cognition only to matters internal to the individual results, according to Hutchins, in attributing every cognitive characteristic to the individual mind while ignoring any role externalities could play (1995, p. 173, 356). This assumption has been rendered problematic by studies from Bruno Latour (1986), Karin Knorr-Cetina (1999) or Hutchins (1995) himself.

57 Hutchins work is a part of a broader line of thought represented also by authors such as Jean Lave (1988) or Lucy Suchman (2007; 1987) and applied in fields of study such as human-computer interaction (Wright, Fields, & Harrison, 2000), religion (Lawson, 1999; Reimer, 2005), morality (Magnani & Bardone, 2008), or work (Rogers & Ellis, 1994). Another approach that also refuses to attribute the epistemic credit to the symbol manipulation performed only by the human mind can be found in the theory developed by Karen Barad who claims that interaction (involving apparatuses used for measurement or observation) constitutes an inseparable part of phenomena (Barad, 1998, p. 95). In her more recent work, she brings this claim to its more radical implications, questioning the metaphysical belief that things have independent sets of determinate properties (Barad, 2007, p. 19). Finally, the central thesis of the Theory of Distributed Cognition is reminiscent of the cyborg image classically described by Donna Haraway: "Why should our bodies end at the skin, or include at best other beings encapsulated by skin? From the 17th century till now, machines could be animated – given ghostly souls to make them speak or move or to account for their orderly development and mental capacities. Or organisms could be mechanized – reduced to body understood as resource of mind.

externalities with epistemic credit:

> If, as we confront some task, a part of the world functions as a process which, were it done in the head, we would have no hesitation in recognizing as part of the cognitive process, then that part of the world is (so we claim) part of the cognitive process. (Clark & Chalmers, 1998, p. 8)

This seems to be the main criterion for determining whether an object is a part of a cognitive system.[58][59] To define what a cognitive system is, we can reach for Ronald Giere's grasp of the concept. In his view, cognitive systems are specified by what they produce: knowledge. Therefore, cognitive systems, even though they include material objects, are based on human agency (Giere, 2002, p. 642). We can also use Clark's and Chalmers' distinction between pragmatic action (alteration of the world for its own sake) and epistemic action (alteration of the world in order to augment cognitive processes, for example search or recognition) (Clark & Chalmers, 1998, p. 8) to claim that cognitive systems engage predominantly in epistemic action.

According to Hutchins, the development of material means of thought results in different representations of given problems so that problems once considered difficult can be turned into easily solvable ones (Hutchins, 1995, p. 367). This connects directly to Keil-Slawik's thesis about removing enforced sequentiality of tasks by condensing them into a single operation that can be easily performed by a program. A sequence of tasks can be mapped, represented as a formal function in a programming language, and implemented in a program by adding it to it's source code. After compilation of the modified source code, the operation is available through user interface of the program. There, it represents not only modification of text (source code), but also modification of an object or a tool (interface). Depending on the type of interface[60], this modification can take the form of an added

These machine/organism relationships are obsolete, unnecessary. For us, in imagination and in other practice, machines can be prosthetic devices, intimate components, friendly selves." (Haraway, 2006, p. 144)

58 There is an ongoing discussion concerning the criterion as represented, for example, by Magnus who considers, whether double blind studies are a case of distributed cognition, given that they in principle could not be carried out within a single mind (Magnus, 2007, p. 301). But these are rather discussions of corner cases, which are not very relevant for the present text. Conceptualizing tools as parts of distributed cognition systems is well established (Clark, 2006; DiMaggio, 1997).

59 Francis Heylighen further characterizes distributed cognition systems with self-organization, co-opting external media, network structure, selective propagation of information and production of novel knowledge (Heylighen et al., 2004).

60 In one of his early texts, Hutchins distinguishes between interfaces based on abstract formalism and direct manipulation interfaces based on graphical representation of tasks (Hutchins, Hollan, & Norman, 1985). The former are typically represented by command line interfaces that rely on complex and precise syntax of textual commands while the latter designate graphical user interfaces that rely on operations with graphical elements. To give an example, copying a file in a command line interface can achieved by a command like:

command in command line interface or added element in graphical user interface, each representing the new operation. In cyberspace, each operation that can be performed has to be an operation of some program and therefore had to undergo this generally described process. As a result, action can be defined as coordination of condensed task sequences represented as operations in a software interface. This positions software (along with the underlying hardware of course) to be the material means of thought of programming.

Clark further specifies objects that meet the criterion for attribution of epistemic credit (what I have been up to now calling "material means of thought") as "wideware" and defines them in a way that shows their direct relevance for this text as objects that "act so as to manipulate, store, or modify the knowledge and information that the organism uses to reach its goals" (Clark, 1998, p. 269). The sociological relevance of these claims, as Ronald Giere and Barton Moffat emphasize, lies in the fact that the shape the distribution of actors and objects takes in particular cases originates in existing social structures (Giere & Moffatt, 2003). Furthermore, Hutchins, when dealing with the question of how are the elements of distributed cognition systems selected and included, relies heavily on the cultural context of cognition. Therefore, he proposes what he calls the "hypothesis of enculturated cognition" according to which "the ecological assemblies of human cognition make pervasive use of cultural products. They are always initially, and often subsequently, assembled on the spot in ongoing cultural practices" (Hutchins, 2011, p. 445). In other words, Hutchins claims that cultural practices play key role in organization of the cognitive systems.[61]

In this work, I assume that the social structures and the *assemblies* of cultural practices take the form of networks. Besides the resemblance in terminology, this decision is also supported by the fact that Hutchins' work on distributed cognition is considered by Bruno Latour to be compatible with Actor-Network Theory (Latour, 2005, p. 60). For example, one

`cp directory1/file_to_be_copied.txt directory2/` while in a direct manipulation interface, one can simply drag and drop the file with mouse. The latter interface seems to be faster and more convenient, however the benefit of using textual commands is that they can be easily combined with other commands, creating a new task sequentiality in a script file that can be executed as a whole, thus representing a new condensation of tasks. See Appendix 1 for an example of a script together with its description.

61 There is also a topic within the theory of distributed cognition which roughly corresponds to the classical sociological dichotomy between actor and structure. Giere shows that there is a tendency to attribute agency to structures by speaking of "distributed minds" but claims that this is not necessary and argues for attribution of agency only to human actors (Giere, 2002, p. 642). In one of his early works, Hutchins draws a distinction between evolution and design to distinguish systems that emerged spontaneously (perhaps as a result of some underlying structure) from those that were results of conscious planning (attributable to human actors) (Hutchins, 1991, p. 38). More recently, Eviatar Zerubavel and Eliot Smith elaborated on the possibility of transcending methodological individualism by considering human actors to mutually constitute the entities of distributed cognition systems for each other (Zerubavel & Smith, 2010, p. 324).

of the central claims of Latour's theory, that connections among entities are constitutive for them (Latour, 1994, p. 35), can be seen as one of the more general assumptions behind the theory of distributed cognition, underpinning the effort to attribute epistemic credit to external entities.[62] Furthermore, Hakken introduces the concept of knowledge networking (broadly consistent with Actor-Network Theory) to avoid some of the controversies stemming from taking into consideration both modernist and postmodernist knowledge discourses and uses it to establish a dialectical perspective (Hakken, 2003, p. 45–47).

An organized set of wideware (that is tools and design artifacts) and its human operators hints at the main types of compositional elements that form the networks of FOSS development projects. At this point I draw from Latour's theory which provides a useful infra-language that specifies how could these networks be mapped (Latour, 2005, p. 174). In other words, it is not a substantial theory of what is going on in any part of the social reality, it is a perceptive grid used for observation (Latour, 1996, p. 11). In Latour's approach, to interpret means to add something from the outside, something which has not yet been mapped to the network.[63] Subsequently, a good research account should be the one tracing a network through different locations.

According to Latour, a network is "not made of nylon thread, words or any durable substance but is the trace left behind by some moving agent" (Latour, 2005, p. 132). Networks are not simply "out there" in the sense of material substance connecting their nodes all the time. Not all networks are like computer networks which need cables, routers and switches to be constantly present.[64] Usually networks need to be mapped and visualized to be graspable. To elaborate the concept more, I note that networks have three basic features:

1. A point-to-point connection is being established which is physically traceable and thus can be recorded empirically.
2. Such a connection leaves empty most of what is not connected.

62 Considering this affinity, it is no surprise to find the two theories elaborated side by side in overview works such as that of Norton Wise (Wise, 2011).

63 This stands in contrast to a more traditional approach to interpretation based on subsuming or connecting particular observations with general concepts. As his criticism of theoretical frameworks demonstrates, Latour is strongly opposed to such a way of interpretation. For him, explaining something with a theoretical framework means to make the phenomenon vanish, because the general social forces play a paradoxical role of a necessary cause which, however, always remains invisible (Latour, 2000, p. 6, 2003, p. 3, 2005, p. 102, 2012, p. 138). In this sense, the general social forces are seen by Latour as something that Gregory Bateson called explanatory principles: "an explanatory principle – like 'gravity' or 'instinct' – really explains nothing. It's a sort of conventional agreement between scientists to stop trying to explain things at a certain point" (Bateson, 1972, p. 43).

64 Latour sees the Internet as increasing the material dimension of networks, he claims: "the more digital, the less virtual and the more material a given activity becomes" (Latour, 2010, p. 8).

3. This connection is not made for free. (Latour, 2005, p. 132)

First, we can see that the tracing is done on a material, not conceptual level. As Latour would put it, the world needs to be allowed to “put itself in order” (Latour, 2005, p. 184).[65] But this “putting into order” is nothing mysterious, it is traceable and accountable. As Latour claims, social formations hold together because of graspable entities that operate within them. And it is by tracing the connections that these entities are part of, that we can understand the particular case of order that is established. In this research project it means focusing on the tools and platforms such as Git, Bugzilla, MediaWiki, IRC and trace their origin, how are they used in the studied project and how did they get there in the first place. It also means focusing on the elements within them, that is, commits, pull requests, bug reports, wiki pages and their histories, IRC channels and the conversations within them. Tracing how these entities are put together will allow me to examine how is this specific type of order put together.

Second, networks are made of connections and so they differ from what is commonly called an area, a field, a sphere, or any surface in general. It is for this reason that networks don’t have borders (Latour, 2010, p. 5), they just leave everything that is not part of them unconnected (Latour, 2005, p. 242). As such, Latour claims that networks are “by no means comprehensive, global or systematic, even though they embrace surfaces without covering them, and extend a very long way” (Latour, 2012, p. 118). He then goes to label everything that is unconnected with the word “plasma”. It represents everything “which is not yet formatted, not yet measured, not yet socialized, not yet engaged in metrological chains, and not yet covered, surveyed, mobilized, or subjectified” (Latour, 2005, p. 244). This implies that the characteristics of a network could be arrived at by examining what is not connected just as well as what is. This is not a call to examine everything but a call to include entities that as we understand could be connected but the connection is avoided or are being connected at the

65 This strand of Latour’s thinking is heavily influenced by Harold Garfinkel. According to Latour, both he and Garfinkel share the view that sociology should be a science examining how society holds together (Latour, 2005, p. 13). Although Latour does not use the concept, Garfinkel, in his definition of ethnomethodology explicitly states that it rests on a presupposition that social settings are to be viewed as “self-organizing”. Garfinkel further elaborates the concept as follows: “Any setting organizes its activities to make its properties as an organized environment of practical activities detectable, countable, recordable, reportable, tell-a-story-aboutable, analyzable - in short, accountable” (Garfinkel, 1967, p. 33). However, it seems that Garfinkel is not the only interactionist precursor of Latour’s approach. If we compare Herbert Blumer’s methodological approach (Blumer, 1986) described by Martin Hammersley as a program of naturalistic research (Hammersley, 1990, p. 156), it seems to be compatible in all points that Hammersley lists. In this context, the concept of self-organization is different from its other version rooted in the tradition of cybernetics (Ashby, 1962; Von Foerster, 2003) and utilized most notably as the concept of autopoiesis (originally developed by Humberto Maturana (Maturana, 1980) and Francisco Varela (Maturana & Varela, 1987)) in the systems theory of Niklas Luhmann (Luhmann, 1995, 2014).

time of observation. For this research project, it means studying closely the cases of newcomers who attempt to join the software development project. It also means examining the tools or platforms that the maintainers for some reason avoid using. Elaboration of such cases can bring key insight into the inner workings of the network.

Third, Latour keeps stating that the connections made within networks are "not made for free" (Latour, 2005, p. 132) or that sites have to "levy the means" (Latour, 2005, p. 174) in order to influence other sites. Such statements point to the role resources play in building networks. Indeed, to connect something does not mean just attaching it. It means searching for something, appropriating it, learning how to use it, spending time with it, modifying it, or buying something, creating something, deserving something. In other words, resources are spent on making connections, on making actor-networks more extensive. Without resources, networks will hardly flourish and so for explaining how a certain network operates, it is necessary to shed light on the resources it uses to do so. This applies also to this research project. Servers are necessary to run Git repositories, Bugzilla databases or IRC channels. But servers need to be provided, powered and maintained, people need to be paid and legal entities must represent every attempt for gaining financial resources. Thus, to trace the tools and platforms means also finding out how can they operate, what resources make that possible.

Now I will attempt to describe the difference between two basic types of units that form networks, that is, mediators and intermediaries. The latter are defined by Latour in a following way: "An intermediary, in my vocabulary, is what transports meaning or force without transformation: defining its inputs is enough to define its outputs." (Latour, 2005, p. 39) Thus, intermediaries are regular and predictable, just like machines that function reliably in accordance with expectations of their operators. Mediators are precisely the opposite, their output cannot be predicted from their inputs, every time they have to be considered with all their particular characteristics. Thus, mediators are anything but regular, they are like unreliable, broken or unfinished machines that seem to do whatever they want.[66]

Whether an entity is a mediator or an intermediary is not inscribed into its nature once and for all, but depends on its observed behavior. This means that an intermediary can turn

66 These definitions appear to be analogical to the definitions of trivial (independent of their past states and therefore predictable) and nontrivial machines (dependent on their past states and therefore unpredictable) developed by Heinz von Foerster (Von Foerster, 2003, pp. 309–313). This line of thought is further developed by Ranulph Glanville who considers scientific knowledge to be a cumulative stabilization over examination of in principle unpredictable (because unopenable) black boxes (Glanville, 1982, 2007).

into a mediator at certain point in time (as when an error is discovered in a piece of software leading to unexpected behavior) or from a certain point of view (as when inexperienced user is discovering unknown functions that certain piece of software can perform). This also applies to the transition from mediator to intermediary (as when a bug in a piece of software is fixed or a user learns what to expect).

In this perspective, mediation is a key concept that needs to be elaborated further. In one of his texts, Latour offers four meanings mediation has: translation, composition, black-boxing, delegation. According to this author, translation is a “displacement, drift, invention, mediation, the creation of a link that did not exist before and that to some degree modifies two elements or agents” (Latour, 1994, p. 32).[67] If we start with the premise that a relation between two elements modifies them, it seems logical to claim that connections are constitutive for those elements. Their “essence” is to be found in the links that connect them to others. Those connections allow them to act in ways they would not be able to just by themselves. This is the second meaning of mediation, composition (Latour, 1994, p. 35). Latour goes as far as equating the embeddedness in various configurations of connections with different modes of existence (Latour, 2007a, p. 24). Therefore, interaction can be seen as action which is shared with actors that have different ontologies because they are made of connections from other spatio-temporal frameworks (Latour, 1996).[68] Furthermore, this meaning of mediation can be related to the claim made by Hutchins and Clark, that actors widen their options for epistemic actions by interacting with wideware and by doing so they achieve augmented cognitive results. Programmers use software interfaces in order to perform task sequences that are condensed into easily performable operations. The way these operations are designed and implemented in an interface determine programmer’s options for action.

However, composition may not be visible at first sight. This is a result of the black-boxed nature of mediators or intermediaries (Latour, 1994, p. 36). Their composition is opaque unless effort is made to make them more transparent. As I suggested earlier, compilation, the act of transforming human readable source code into machine readable

67 It seems that the concept of translation is roughly analogical to the concept of emergence used in the systems-oriented tradition of thought (see for example the work of Poe Yu-Ze Wan (Wan, 2011, pp. 69–82)). However, while emergence is associated with the premise that the whole is always more than the sum of its parts, in Latour’s approach, the whole is always smaller than its parts (Latour, 2011, p. 6). This is so because the whole black-boxes its composition by appearing as a single agent.

68 John Law articulates this perspective through the term empirical ontology, which he summarizes in the following way: “It washes away assumptions about pre-given realities and instead asks questions about how realities are done in practices.” (Law & Lien, 2013, p. 3)

binary files, represents a finest act of black-boxing. The contents of a binary file after compilation are not intelligible for human readers and thus an understandable text is turned into a black-boxed thing. Finally, delegation makes explicit what mediators or intermediaries do: they achieve spatial and temporal shifting of action (Latour, 1994, p. 39). They make overcoming distances and durations possible to bring a certain kind of action to a situation. In cyberspace, delegation is carried out primarily by persistent digital text or an interface of a compiled program. They are able to overcome distance and endure time in order to carry information (digital texts) or carry out an operation (interfaces).

To sum up Latour’s perspective: any interaction takes place in a situation which is full of elements that originated somewhere else, at a different time and perhaps were put into motion by some other agency. This means that an observer of an interaction should be led away from it and drawn towards the different places, times and agencies that played a role in putting together the elements of the examined situation. This research strategy is consistent with Dominique Vinck’s approach, which she elaborates under the label “technical ethnography”. According to Vinck, constant effort in both of the areas of production and use of technology is necessary in order to keep the technology existing. The material dimension of a technological object may create the basis for its performance, but it cannot be fully grasped without considering number of actors and intermediary objects organized toward it. This creates the ground for the “objectivity of technology” (Vinck & Blanco, 2003, p. 212). This corresponds to Latour’s claim that “what lasts (…) is generated by what does not last (the constant work of taking it up again)” (Latour, 2011, p. 12) and to his resistance to an idealized form of materialism that take as a reference point the ideas of what things in themselves should be. Instead, Latour directs the attention to the actual processes that result in things as they are (Latour, 2007b, p. 139). Finally, Vinck’s perspective on objectivity of technology echoes the title of one of Latour’s studies: “technology is society made durable” (Latour, 1991).

Furthermore, Latour’s famous statement “follow the actors themselves” is resembled by the perspective employed by technical ethnography, which tries to see “technology in action”:

> Technology in action can therefore be grasped by following and reporting on the action and movement with and by all the intermediaries: tools that are bent and bruised (…), texts that are covered with contradictory and half-finished notes, drawings that never show everything and seem to adhere to tacit conventions, tables of data, and so on. (Vinck & Blanco, 2003, p. 219)

This illustrates the entities this approach focuses on. The general question that technical ethnography aims to answer with such observations is: “What makes something work, and why does it produce the effects it does?” (Vinck & Blanco, 2003, p. 3). The question is twofold and centered around a technological object. First, it aims to trace the elements that were needed to bring together in order for the object to perform. Second, the objective is to make account of other subsequent compositions where the object plays the role of an element, a part. As a result, technical ethnography, as conceived by Vinck, can be used to track the composition of actor-networks.

But Latour’s approach has one more important characteristic. He insists that the observer should abstain from calling in a notion of context or structure and by that adding a third dimension to the interaction being mapped. The observer should move only through the two-dimensional “flatland” of concrete mediators, intermediaries and transformations. This should be done in three steps.

The first step consists in localizing the global. Every element present in an interaction has been put there by other interactions happening in other situations. Or as Latour puts it: “If a site wants to influence another site, it has to levy the means. The tyranny of distance has been underlined again.” (Latour, 2005, p. 174) To explain an interaction then means to trace its connection with other interactions. The micro / macro distinction is replaced by distinguishing sites with different scope of influence.

The second step is redistributing the local. This is where we get to the meaning of the expression “actor-network”. It means that every actor can be decomposed into a network of interconnected mediators, which, in turn, establish the actors subjectivity. The existence of actors relies on the ties they are composed of. “So every assemblage that pays the price of its existence in the hard currency of recruiting and extending is, or rather, has subjectivity.” (Latour, 2005, p. 218) However, these ties not only allow actors merely to exist, but they also make them do things. This is what Latour’s theory of action is based on. The fundamental concept here is that of a plug-in. Plug-ins (e.g. tools or other equipment available in the network) not only make actors competent in given areas of conduct, but also transform them in the sense that they guide their actions.

The third step consists in connecting sites. This step relies on Latour’s conceptualization of information. According to him, “to provide a piece of information is the action of putting something into a form" (Latour, 2005, p. 223).[69] In other words, information is embedded in forms through which it can be handled or passed on. The decisions concerning what to pass on and what not can be observed as decisions on which entities they want to multiply and which to rarefy.[70] Entities which succeed in the multiplication game can be seen as subject to wide agreement and thus might be considered to be standard. The significance of standards lies in their ability to render things comparable and in doing so, facilitating coordination. Comparability and coordination are central to maintenance of ties among different sites and therefore allow for the existence of that which has to be traced.

In this work, I employ the principles by playing down contextual notions such as the free software movement values, hacker ethic, recursive publics or class struggle at the beginning of my analysis. Instead, I focus on the licenses, tools and platforms and study the implications of their design to localize and pinpoint such global phenomena. Furthermore, participant observation allows me to identify the composition of the actor-networks that represent the contributors. Again, the software tools and platforms play a key role here, but also hardware aspects like screen size or availability of server space or the speed of Internet connection. Examining the composition makes visible what is going on in the field. I try to make the black-boxed compositions transparent to grasp the connections that allow actors to act in the ways they do. Subsequently, I assemble my observations in order to see the standards permeating the sites. But not only to identify them, also to examine the processes that lead to some technology becoming standard.

To sum up my theoretical approach, I will now go through the most important points made in this section. I argued that programming, an activity central to software development, requires knowledge not only in the area of how to use programming languages and other tools, but there is also a significant part of knowledge that consists of a theory, or an approach, with which a particular program is developed. An important role in the software development process is played by artifacts. These may be used either as tools that augment or replace human activity, or as design artifacts, intermediary outcomes which allow for further reflection or new experience. Reflection and experience form the basis of vertical and horizontal distribution of knowledge and so, any change in distribution of knowledge (which

69 Or as Hutchins would probably put it, symbol manipulation always has its material side.

70 Through the perspective of Walter Buckley, the basic mechanisms that could be identified in this phenomenon would be positive and negative feedback; the former leading to deviation amplification and structure building, while the latter resulting in deviation counteracting and stabilization (Buckley, 1967, p. 59).

is key for organizations focusing on knowledge production) must come through these processes. However, it is necessary to differentiate knowledge, which involves appropriation by an actor, and information which denotes data that actors possess. Similar distinction can be made with regard to information and meaning, where the latter arises from relating information to one another within a system of reference.

Therefore, within software development, design artifacts consist of information that can be easily stripped of its meaning, in other words, decontextualized. To resist this, reciprocity of perspectives must be established among the parties exchanging design artifacts. I delimited three main aggregates according to their experience with developed software: developers, newcomers and users. I also assume that these groups are internally more consistent than among each other. Based on this, I identify the points where the probability of decontextualization is highest, that is, the exchanges of design artifacts between members of different aggregates. I further suggested several practices such as work observation, consulting archives or documentation, testing, and the use of common tools, that contribute to establishing of reciprocity of perspectives. However, as free and open source software development is an online constituted practice, the conditions in which it operates are specific. I argued that this is mainly because the omnipresent use of digital text which has unique characteristics that allow for storage, reproduction and automated manipulation at the price of computational resources only. I also argued that FOSS software development was one of the first practices that systematically employed these possibilities which, in turn, gave the practices involved in establishing reciprocity of perspectives a new dimension.

In this line of reasoning, tools and design artifacts play central role. It is through them that cognitive processes leading to production of knowledge take place. They constitute the so-called wideware, the parts of cognitive systems that are external to humans. At the same time, the distribution of wideware in particular cases originates in existing social structures. These structures, I assume, take the form of networks which can be assembled as conglomerates of physically traceable connections that are selective and require resources to be maintained or extended. The elements operating within a network can be of two types, reliably transporting intermediaries and unpredictably transforming mediators. What these elements perform can be grasped with the concept of mediation and its four meanings: translation, composition, black-boxing and delegation. By tracing the mediation of identified elements a network can be assembled and described as a form of explanation of a phenomenon.

I will now use the implications of this section to reformulate the problem at hand. The main research question could be formulated along the following lines: given that the design artifacts, which are essential for software development, consist of information, how is it possible that with regard to them, reciprocity of perspectives is established and standard knowledge is produced among varying aggregates of actors? In the following lines, where I will attempt to answer this question, I will assume that the networks which form free and open source software development projects contain elements (wideware) that are formative for the cognitive processes which lead to the induction of knowledge. The practices which I already claimed to be important for establishing of reciprocity of perspectives (work observation, testing, etc.) represent my starting clues, but the strategy of network tracing leads me from them to the elements and resources that make them possible. As a result, the question of how knowledge is produced in free and open source software development can be specified in two ways. First, we must ask how do digital texts and software interfaces (as material means of thought) mediate actions of programmers. Second, we must ask how, or using what resources is the price to make connections paid in cyberspace. Thus, the following analysis will not be limited only to the examination of the aforementioned practices, but will be aimed at exploring also the less visible fringes of the network that make them possible.

3. Methodology

This chapter contains a basic formulation of the methodological approach utilized in this research project. In the previous section, I stated that the shape of systems of distributed cognition depends on existing social structures and that I assume that these structures can be grasped by assembling connections into networks. The method of choice to achieve this is usually ethnography. Indeed, as Richie Nimmo states, there is a significant overlap between Actor-Network Theory and the use of ethnography (Nimmo, 2011). The ethnographic approach, usually perceived as a combination of field research and the resulting written accounts (Packer, 2011, p. 209), offers a suitable variety of tools for detailed scrutiny of social practices in their entirety and context. It has been also suggested by several authors (Klischewski, 2002; Rönkkö, 2002; Westrup, 2002) that ethnography is probably the most appropriate method for studying software development. Furthermore, ethnography has already been used several times for research on free and open source software development (Coleman, 2010, see for example 2013; Ducheneaut, 2005; Kelty, 2008).

According to Isabelle Baszanger and Nicolas Dodier, there are three characteristics distinguishing ethnography from other approaches to the study of human conduct. First, it is the empirical basis of findings, as opposed to purely deductive approaches. Second, it is the openness toward new phenomena as opposed to approaches that start with categories specified in advance. Third, observations are related to the historical and cultural context of the studied environment as opposed to approaches such as conversation analysis or ethnomethodology (Baszanger & Dodier, 2004, p. 13). Ethnography is well suited for making empirical descriptions, being open and tracing wider connections – the characteristics that, according to Latour make a good research account:

> A good ANT account is a narrative or a description or a proposition where all the actors do something and don't just sit there. Instead of simply transporting effects without transforming them, each of the points in the text may become a bifurcation, an event, or the origin of a new translation. (Latour, 2005, p. 128)

The emphasis on what actors actually *do* connects directly to the central concept of technical ethnography – performance. Performance denotes what is produced through human conduct (Vinck & Blanco, 2003, p. 208), it is the outcome of collective action (Vinck & Blanco, 2003, p. 217).[71] Being that, performance is subject to variability due to change in either its

71 Performance in this sense is also elaborated by Jansen and Vellema (2011, p. 171) who try to develop a similar approach called "technography". In doing so, they link performance to the theory of situated action

compositional elements or its compositional structure. The most illustrative definition that can be found in Vinck (2003) is the following:

> “performance” means what is produced in whatever register is used: technico-economic productivity, technical demonstration of the unsuspected possibilities of a machine or product, operator virtuosity (…), beauty of the machine’s movement, turnaround in power relations, identity of a professional group, exalting destructive power, or demonstration of the potential violence of a technology (with weapons, for example). (Vinck & Blanco, 2003, p. 215)

This means that technical ethnography is predominantly concerned with the process of using tools and design artifacts to produce something.[72] The focus is on the process of making, doing, creating, inventing, or coming up with workarounds, while intentions and rationalizations lurk in the background. Technical ethnography presupposes actors that are, in the words of Natalia Rybas and Radhika Gajjala, “creative appropriators of technology” (Rybas & Gajjala, 2007, p. 13).[73]

To make a good description of the use of tools and design artifacts, participant observation is a crucial source of data. According to John Brewer, observation represents a technique of data gathering through contact with everyday lives of the participants (Brewer, 2000, p. 59). Its purpose is to generate data through watching what participants do, listening to what they say and experiencing the conditions in which they act.

During participant observation, the role of the researcher is dual – it contains aspects of both participation and observation. Conceiving them as a binary opposition, Sladjana Nørskov and Morten Rask identify four positions that a researcher could adopt with regard to the duality. In what they call a participant–as–observer role, the researcher becomes a member of the studied group in the sense of participation on group activities (Nørskov & Rask, 2011). The advantage of this role compared to the others (observer, participant, observer–as–participant) is the direct access to experience which is key for understanding what is going on in the field. The need for participation, as opposed to what is called lurking (passive presence only to monitor activity), is emphasized for example by Heike Greschke (2007, p. 18).

(Suchman, 2007).

72 The important role tools and design artifacts play in the process also hints at the importance of existing infrastructure, which is consistent with an ethnographic approach developed by Susan Leigh Star (Star, 1999, 2002; Star & Ruhleder, 1996).

73 By placing emphasis on the technological performance, this type of ethnography is close to focused ethnography, defined by Sarah Wall as an ethnography, which “usually deals with a distinct problem in a specific context and is conducted within a sub-cultural group rather than with a cultural group that differs completely from that of the researcher” (Wall, 2015, p. 8).

Brewer states that it is necessary to distinguish if undertaking the investigation means entering new environment for the researcher or if it allows for staying in a familiar one. In both cases the researcher can either employ one of his existing roles or accept a new one (Brewer, 2000, p. 60). Concerning the research project at hand, I was already familiar to a certain extent with the environment I was about to enter. Prior to conceiving this project I was reading reports on websites focusing on free and open source software, following updates of some of the biggest FOSS projects and using the software. However, I was not actively participating on development in any FOSS project. That is why entering the field meant negotiating and accepting a new role of documentation writer. This came out rather naturally as one of the project maintainers made a call on his blog that anyone interested in this position should contact the team in their IRC channel. When I made the contact, I made it clear that I would do the work as part of my research. No one opposed it, on the contrary, I was given pointers on where to start to understand the architecture of the project. The hardest part of getting into my new role was establishing the work environment, and learning the features of the developed software enough to be able to write the user documentation. This required time and effort both on my part and on the part of the project maintainers who were helpful in solving problems that occurred through the process.

In this research project, the direct experience is of high importance because it gives me an opportunity to use the tools, design artifacts, platforms and infrastructures employed in knowledge building while also experiencing the whole process of establishing work environment, learning how the tools behave and coordinating work on contributions. Contributing to the project also has an ethical dimension of giving back to the project, whose maintainers agreed with my fieldwork in the environment.

To supplement participant observation, I opted for extensive gathering and analysis of documents I encountered not only during my direct engagement in the project (like bug reports or commits) but I also made concentrated efforts to gather documents that were of importance for making important connections in my description. For example, I systematically browsed through old blog posts and wiki pages in order to reconstruct the project’s history and its relationship to a business company as it evolved through time. Furthermore, I also utilized my knowledge of news sources for the domain of free and open source software in general and also for the specific areas of concerned software development projects to scan for any material relevant for analysis.

According to Paul Atkinson and Amanda Coffey, it is necessary to bear in mind that documents do not account for the reality independently of their author's position. As a result, it is necessary to examine their character and utility and not just assume the position of a reader for whom the documents are intended to (Atkinson & Coffey, 2004, p. 73). This includes examining the practices related to how the documents are produced, stored, distributed and used. In this regard, it is useful to apply Norman Fairclough's distinction between the level of text itself and the level of discursive practice (Fairclough, 1992, p. 73), which, for example, involves the relationships between individual texts. According to Atkinson and Coffey, such relationships could traditionally take the form of hierarchical patterns, nesting documents one into another, or sequential patterns, continuing one document with another (Atkinson & Coffey, 2004, p. 67). But, perhaps more importantly, the analysis on the level of discursive practice must take into consideration the agency that could be attributed to the documents – the ways in which they are defining and helping to constitute subjects, objects and domains (Nimmo, 2011, p. 114). That is, for example, whether a document classifies a certain set of licenses as compatible with the FOSS development model, and thus allowing projects that use these licenses to be part of the FOSS domain.

Most documents I deal with are hypertext documents. This type of documents also exhibits hierarchical or sequential patterns, but these are not entirely predefined by the author. As Cantoni and Tardini remind us, the reader is able to make choices on which links to follow (Cantoni & Tardini, 2006, p. 71) and therefore creates sequences based on his interests. However, hypertext documents are typically found on distinct websites, indicating their relevance to a certain general topic. Furthermore, in order to navigate the user, websites are usually hierarchically structured according to topical areas so even if there are hyperlinks that lead outside the website or its topical area, it is possible to examine all the hypertext documents systematically. In this way, I aim to take advantage of the self-documenting nature of the project by analyzing its website as a set of nested documents.

The same applies to maintainer's blogs or other websites related to the project. But there are also other documents which, even though they utilize the hypertext infrastructure, do not constitute hypertext pages in the usual sense. With this, I imply git logs and bug reports as documents. Both represent persistent traces left after a development contribution. In a certain sense, the relationship between bug reports and the corresponding git logs can be seen as a relationship between problems the project faces and solutions its members have devised to fix them. However, the numbers in which these documents exist are overwhelming and because of that git logs and bug reports are analyzed only with regard to incidents (traced through searching or hypertext links) that I selected for further examination.

According to Lindsay Prior, documents enter the human experience as bearers of instructions, obligations, wishes or information. In this sense, documents might be able to project themselves into actions taken by their recipients or extricate from the original intentions of their producers (Prior, 2004, p. 76). In this sense, documents can (but are not limited to) represent design artifacts that mediate the process leading to certain kind of performance (typically bug reports and git commits/logs). In other cases, they provide additional explanations or rationalizations (typically blog posts or wiki pages). The importance of documents is demonstrated by the fact that in certain settings, the right to produce or alter them is a privilege granted only to certain individuals or groups. Write access to the project's website, git repository or to certain administrative actions in Bugzilla is held by maintainers of the project. The content found in these documents is therefore created or at least reviewed by the core members of the team.

3.1. Analysis and Interpretation Through Indexing and Writing

Once gathered, data has to be dealt with through analysis, interpretation and presentation. Data analysis can be defined as a process in which data are organized through selecting and indexing segments. Interpretation involves assignment of meaning to the analytical units and leads to understanding the data. Presentation fixates results of the previous procedures in various forms, most importantly in text. Of course, the separation of data gathering, analysis, interpretation and writing are artificial and serves only explanation purposes in this section.

Analysis represents the first stage in reduction and organization of data, opening space for interpretation and presentation (Brewer, 2000, p. 108). Authors like John Brewer or Matthew Miles and Michael Huberman agree that data analysis is best performed together with data gathering (Brewer, 2000, p. 107; Miles & Huberman, 1994, p. 65). This strategy provides the researcher with continuous stream of analysis results to orientate further gathering so that the researcher focuses on relevant events and gains feedback on operational conclusions. In this research project, fieldwork is done continuously, mostly without the need to travel long distances to reach the field. This creates good conditions for analysis parallel to data gathering.

Collected data were directly typed or copied to plain text files. A markup language called Markdown[74] and its syntax highlighting support in text editors[75] were used for indexing important expressions or denoting incidents so that basic organization of data is achieved immediately after them being gathered. Data are supplemented by a timestamp, a note on their source, label of an incident[76] and highlighted keywords. In ethnographic research, this procedure is usually called indexing (Brewer, 2000, p. 110; Neyland, 2008, p. 126), while the adherents of approaches, where systematic multilevel coding is more common would probably call it pre-coding (Saldana, 2009, p. 16).

No matter how it is called, this procedure is essential for efficient browsing and retrieving of data. Indeed, David Fetterman links the analytical organization of data with more general issues of data storage and retrieval (Fetterman, 2010, p. 117). The important point here is that efficient browsing facilitates further refinement of indexing by allowing quickly going through the data repeatedly. In this way, the analysis bootstraps itself – current organization of data facilitates its further organization. As a result, order is brought to data, a state which, according to Brewer, should be the result of analysis (Brewer, 2000, p. 108).

Repetitive browsing through data also stimulates researcher's imagination. To capture the ideas and descriptions emerging in this process, it is necessary to write memos during all stages of research. I used a separate plain text markdown file to store all memos written in course of this research project. In it, memos are separated by headings, supplemented by keywords while important expressions are still highlighted in the text. In writing memos I usually drew together different parts of data to combine them in an emerging description (while also pointing to the indexes so that the particular pieces of data would be easily retrievable). The first immediate result of writing is that it feeds back into data, validating or suggesting alternatives for their indexing – in the same sense in which Neyland links writing with ordering work (Neyland, 2008, p. 127). But more importantly, and in line with Brewer's (2000, p. 133) suggestions, I used the space for informal writing provided by memos to think through what is going on in the data and to clarify my descriptions. In other words, writing is

74 More information about Markdown can be found here: http://en.wikipedia.org/wiki/Markdown.

75 Syntax highlighting is a standard feature of most plain text editors. Based on the file's extension, the editor determines the language that is used in it and applies corresponding rules to highlight common patterns of syntax. The purpose of this function is to enhance readability and to facilitate quick orientation in the contents of the file. As a simple example of markdown syntax highlighting: words surrounded with asterisks (*word*) – indicating a word in italics – are highlighted with purple (the color is arbitrary in principle, it depends on a concrete implementation in a text editor) font color.

76 Here, incidents mean events like discovering and filing new bugs, discussing severity of bugs, reviewing commits, pushing commits, or testing new features. Comparing incidents among each other provides ground for better understanding of what is common to them. Furthermore, focusing on incidents corresponds to the emphasis on observing performance – what is being done.

essential for sorting and precising of thought. Furthermore, early writing also has the potential to reveal gaps in data (Brewer, 2000, p. 133). These provide incentives for adjusting fieldwork in order to cover them.

Over time, the memos grew into descriptions that I attempted to make as thick as possible. I explicitly draw on the approach of Clifford Geertz, who claims that ethnographic description is *interpretive*, concerned with *discourse*, which it attempts to *fix* in perusable terms, while also being *microscopic*, that is, it confronts abstract terms with concrete settings of the field (Geertz, 1973, p. 20–21).[77][78] There is just one important caveat to the use of this approach here – my descriptions are not wholly discourse oriented, they include orientation on material things and their relations. In this way, my thick descriptions are not entirely consistent with Geertz's theory focused on culture, but incorporate a Latourian influence.[79] That is to say, I use thick description as a way of providing descriptions that include what is going on "behind" the observations (and here comes the twist) – the links, connections or broader situations often represented by material traces.[80]

The generalizations that can be based on such descriptions are of specific kind. They are not empirical generalizations (claims that findings are valid for larger population than is the sample), as Brewer (2000, p. 77) would call them, nor are they what Geertz (1973, p. 26) calls governing laws (rules used for prediction). They are defensible interpretations – they do not predict, but their relevance for future is that they should hold against new cases (Geertz, 1973, p. 26). However, along the lines of Latour's theory, to interpret here means to add something, to add the traced connections and to view situations through the grid of assembled networks (Latour, 2005, p. 244).[81] To be sure, the description of standards allows to make claims outside the strictly defined field, but these are usually more vague than typical empirical generalizations.

77 This microscopic orientation roughly corresponds to Latour's first analytical move, localizing the global (Latour, 2005, p. 173).

78 With reference to another Geertz's work, I aim to avoid the strict empiricism resulting in the production of an "ethnographical telephone book" on the one hand and the omnipresence of grand ideas producing a "historical opera" on the other (Geertz, 1996, p. 23).

79 Perhaps the influence could also be framed as that of a branch of thought currently known as "New Materialisms". Diana Coole and Samantha Frost summarize the approach in the following way: "For critical materialists, society is simultaneously materially real and socially constructed: our material lives are always culturally mediated, but they are not only cultural. As in new materialist ontologies, the challenge here is to give materiality its due while recognizing its plural dimensions and its complex, contingent modes of appearing." (Coole et al., 2010, p. 27)

80 This "behind" corresponds to Latour's second analytical move, redistributing the local (Latour, 2005, p. 189).

81 This kind of interpretation corresponds with Latour's third and last analytical move, connecting sites (Latour, 2005, p. 221).

To give my analysis a more concrete shape, I used three main files to achieve the transformation of data into descriptions: one representing a research journal where all data are stored and indexed, one dedicated to memos and one for writing the more or less finished descriptions – the text itself.[82] Furthermore, I keep literature notes in multiple plain text files, usually one per topical area. These notes are indexed in a very similar way to that of my data, using the same functions of syntax highlighting. This configuration allows me to bring together a variety of sources to build up my ethnography: field notes, gathered documents, memos or other research texts – to draw on a list put together by Neyland (2008, p. 128).

3.2. Research Field

Predominant part of my research takes place online. This means that the information flows necessary to gather data and interact with participants are carried through a technical infrastructure known as the Internet. According to Latour, this digital environment greatly increases the materiality of networks, making them less virtual than before (Latour, 2010, p. 8). This claim seems to be at odds with how the online environment is typically characterized – as bringing virtuality into a world that has been up to that point material. But, as we have seen, that is a perspective emphasizing the potential of the networks. In my work, I see the digital as a result of actual network of ties that had to be put together.[83] In the end, digital information is nothing else than organized values of voltage. But one does not need to go that far. It is sufficient to consider the actual effort that goes into creating a digital artifact – the mobilization of actual people spending their actual time, connected through actual infrastructure. Therefore, in this text, digital is seen not as virtual, it is seen as actual, that is, material, graspable and traceable.

The Internet infrastructure is often characterized as making time and space irrelevant, but such claims hold only in certain sense and need to be carefully specified. The time dimension is displaced by the ability of digital information to persist. This can be illustrated by the opposition of synchronous and asynchronous communication which is elaborated for example by Cantoni and Tardini (Cantoni & Tardini, 2006, p. 44). The tools belonging to the former type of communication are not designed for later retrieval of information and because

82 This setup roughly matches Latour's suggestion for keeping notebooks, especially what he calls the second and third notebooks kept with the purposes of gathering information and writing trials (Latour, 2005, p. 134).

83 An alternative would be to look at the Internet as a distinct kind of space where interaction is made possible, as for example Anette Markham (2004, p. 99) notes. But that would correspond to a different kind of social topology (regions), than what is used through this text (networks). For a more detailed elaboration of the difference, see the work of Annemarie Mol and John Law (1994).

of that both parties have to be present at the same time.[84] On the other hand, tools belonging to the latter type of communication create a persistent trace that can be retrieved at later time and in doing so enable both parties to sustain the communication even when they are not simultaneously present.[85]

Second, space is to some extent rendered irrelevant by the infrastructure's ability to transfer information in real time between any two nodes of the network. However, this does not mean that the space dimension of the world becomes wholly irrelevant. As already mentioned, the problem known as digital divide refers to the fact that the infrastructure is far from being universally present throughout the world (Norris, 2001). If present, the infrastructure provides varying levels of connectivity. As a result, there is a bandwidth limitation which often favors low bandwidth media (such as text) for transferring information (Cantoni & Tardini, 2006, p. 44). Although higher bandwidth media (such as pictures, sound, video) are ever more common, these play only supplemental role in practices like software development as these practices predate the spread of digital high bandwidth media and thus are fully attuned to the nature of digital text. Therefore, most of the traces that this research deals with are of textual nature. In this sense, Maurizio Teli, Francesco Pisanu and David Hakken suggest to see the Internet through a library-of-people metaphor (Teli, Pisanu, & Hakken, 2007, p. 36).

With regard to spatiality, Cantoni and Tardini also speak about the "new syntax" related to the word "here". This word can now have several distinct meanings: it can mean the place where the body of the user is physically present; it can designate the online space the user is active in; it can also denote the desktop space of the computer which the user is using to connect the former location (Cantoni & Tardini, 2006, p. 59). As if this was not enough, Markham notes that the users are typically present in more than one online space at a time (Markham, 2004, p. 105), making the traditional notion of presence limited to only one place problematic.[86]

This issue shows its full extent once we realize that the data gathering techniques of ethnography are traditionally very closely related to presence in the field (i.e. at a certain place). The approach to data gathering must be adjusted appropriately and this is where the multi-sited version of ethnography comes into play. This approach has been developed since

84 A classical example of this type of communication tool is chat.

85 A classical example of this type of communication tool is email.

86 The multiplicity of communication sites that are maintained by varying types of collaboration is also noted by Christine Hine (Hine, 2000 ,115).

1995 when the defining article from George E. Marcus was published. In it, Marcus defined multi-sited ethnography as a form of research that is tracing a certain cultural phenomenon through various settings (Marcus, 1995, p. 669). According to Mark Falzon, this approach rests on the assumption that space is socially produced and on a subsequent realization that ethnographers could also produce a space of their own (Falzon, 2012, p. 4).

However, the conceptualization of space in this approach is far from unproblematic. Joanna Cook, James Laidlaw and Jonathan Mair suggest distinguishing between space (geographical area), place (cultural territory) and field (cultural territory in geographical area appropriated for research) (Cook, Laidlaw, & Mair, 2009, p. 60). According to these authors, the turn to multi-sited ethnography implicated the acknowledgment that cultural territories spanned multiple geographical areas or were part of networks that could not be examined from one geographical area only. In order for the field to correspond to a cultural territory, it has to involve multiple geographical areas (Cook et al., 2009, p. 63).

But Cook, Laidlaw and Mair went beyond what the first generation of researchers promoting multi-sited ethnography proposed and suggested an approach called "un-sited ethnography" (Cook et al., 2009, p. 69).[87] They made one more distinction by insisting that whereas spaces or places are two (or more) dimensional areas, ethnographic field is only a collection of one dimensional lines connecting points of observation. These lines may effectively intersect borders found in spaces or places and in doing so, provide useful data for making comparisons (Cook et al., 2009, p. 63–64). The decisions on what to include in the network-shaped field should be made based on what the research is focused on and what the research questions are (Cook et al., 2009, p. 65). In this regard, Marcus laid out several options of what can be traced through the various settings. These options include following certain people, things, metaphors, narratives, biographies or conflicts (Marcus, 1995, p. 106–110).

Within this research project, the un-sited approach is useful for drawing together the various sites and platforms that are mobilized during the studied instance of software development. There is an IRC channel for synchronous communication, Bugzilla database for bug tracking, Git repository for hosting the version controlled source code, website as a persistent reference point for information and there are also events at which developers from a broader community come together. What ties these sites together is the role they play in

87 Similarly, Hakken proposes non-site bound or trans-sited approach to grasp cyberspace research that takes advantage of the hypertext nature of the web (Hakken, 1999, p. 59).

development of the particular piece of software which are embodied in the frequent references and links to each other. For example, when a bug is filed, a message is automatically generated and appears in the IRC channel. Or when a patch fixes a particular bug, the git log contains the number of the bug as filed in Bugzilla. These public places are accessed by developers from other, private spaces of their bodily locations and personal computers.

The interaction nodes present on online platforms provide the substrate for my points of observation. My field is then, quite literally, constituted by links between these points. Although I have no direct access to the private spaces of the developers, I can get indirect glimpse of it through what they say or through attending events organized around software development in which they take part. That is to say, my approach is un-sited in this particular sense of distributing my observation across several platforms connected to the development of certain piece of software which also provide data to infer what is happening beyond them.

3.3. Research Sample

The FOSS project in which my fieldwork took place was selected according to several criteria. The first three of them are based on my prior experience with the field combined with key concepts and points made above in chapters containing review of prior research and theoretical framework. The criteria were in place to assure that the project was a good representative of free and open source software projects in general, by placing emphasis on what I learned to be standard in this type of software development. The last criterion was present for practical reasons, to make sure that the fieldwork would be manageable for a single person.

Licenses. The source code of the developed software (and possibly other components such as documentation) is licensed under one of the copyleft licenses. The archetypal licenses among these are the GNU public licenses created and maintained by the Free Software Foundation (Lessig, 2006, p. 148). Among these belongs the GNU Lesser General Public License under which the source code of the examined software is licensed. The software documentation is licensed under a Creative Commons Share Alike license, a type of Creative Commons license that preserves the “viral feature”[88] of the free software licenses.

88 The requirement made by certain licenses (typically the GNU public licenses) that all derivative works should be licensed under the same or compatible license is sometimes referred to as the viral feature (Kelty, 2008, p. 179), drawing a picture of pieces of free software software infecting larger bodies of source code with their licensing. However, this feature could also be seen as a counter-measure to practices of massive appropriation and relicensing of the source code by anyone who would aim to profit from free riding on FOSS development efforts (Söderberg, 2008, p. 46).

Tools. The software is developed by using tools typical for FOSS projects. FOSS developers have a tendency to use tools that are also developed as free and open source software, surrounding themselves with objects of familiar nature. In this regard, the choices are especially relevant for the use of version control tools, bug tracking tools and a communication platform. In the project I selected, Git is used for version control – the same tool that is used in the iconic open source project: the Linux kernel. Bugzilla, the bug-tracking system used in the project is also used in development of the Linux kernel or the Mozilla Firefox browser, one of the most famous pieces of open source software. IRC, as a tool for communication is the de-facto standard for synchronous communication in FOSS projects. What is not standard is the absence of use of mailing list in the selected project. This means that synchronous communication is predominant within the project and that there is no archiving of communication (which typically goes hand in hand with using a mailing list). This is something specific about this particular project and it is necessary to take it into account during the research.

Embeddedness. The project is part of a broader ecosystem of other FOSS projects and interacts with those when needed. This means that it recognizes the openness of the development processes of software it utilizes (either as tools or as dependencies) and participates on it at least occasionally for example by filing discovered bugs, providing patches etc.

Size. The project size is key for the amount of communication and other processes that take place at any given point in time. Therefore, the project had to be small enough for information streams to be manageable, but on the other hand it had to be sufficiently big to allow observation of cooperation among several core developers. With this criterion in mind, I searched for a project that had a minimum of four and maximum of ten core developers. These numbers may appear small but we must bear in mind that they apply only to core developers. In most projects there are numerous other participants that take the roles of new developers, occasional patchers, documentation writers, translators, etc.

Another criterion that was important for selecting the project was a sign that I could actually do some work there. This made my entrance into the project similar to more common cases of new volunteers and provided me with the opportunity to do participant observation. Therefore, my entrance to the field happened in several steps. The first direct contact I made with my field was at the GNOME Users and Developers European Conference which is an annual event (hosted in August 2013 in Brno, Czech Republic) focused on development of the GNOME desktop environment. GNOME is a wide project aggregating many smaller projects underneath its label. Therefore, my presence at the event had a twofold purpose – to try to pre-select a smaller project I could do participant observation in and to familiarize myself with the environment by being present at the conference itself and the hackfests (events dedicated to aggregating participants to work together on a selected problem) that took place after the conference officially ended.

Later (in November 2013), through monitoring the GNOME blog aggregator (called Planet GNOME), I discovered that a project called Pitivi needed someone for writing user documentation. It turned out that it was a rather small project aimed at developing video editing software and that it met all of the above listed criteria. I decided to contact the author of the blog post (who was one of the maintainers of the project) using the project's IRC channel so that our conversation would happen somewhere anyone associated with the project is present, in the sense of being able to read it or participate in it. In this contact, I made it clear that I will be contributing to the project as part of my research and I spent some more time in the channel discussing the aims of my research project with several core developers who expressed interest.

From that point on, I became a curious newcomer that required help and explanations at times but that also tried hard to learn the craft and give back by doing something that would be of use to the project. Taken together, my participant observation was spread in a time frame of around a half year, starting at the first day of contact in the IRC channel. However, my passive presence in the project – everyday monitoring of the IRC channel where all of the non-private synchronous communication takes place and browsing through the sites holding records of asynchronous communication – spanned for around a year, until the end of 2014. Part of the interaction with the software developers later on was also their checking of some of the material I wrote (an article discussing my preliminary results) and its subsequent discussion.

4. Dismantling Pitivi, the Video Editor

The development of Pitivi started in 2003 as Edward Hervey's end-of-studies project at the EPITECH engineering school in Paris. Initially, there were 10 students working on the project, aiming to have something usable before graduating. They decided to base their work on an existing FOSS multimedia framework known as GStreamer. This decision allowed for the support of multimedia formats and operations already present in GStreamer to be gradually implemented in Pitivi without re-inventing them. But this also meant that in its functionality, Pitivi relied heavily on GStreamer which was at that time under heavy development and was not considered stable. Therefore, in the first years of development, the focus was aimed at GStreamer and Hervey managed to make it a full time job. After graduating, he was hired by a company called Fluendo in order to work on GStreamer. This lasted two years during which development of Pitivi itself was in a state of limbo. After that, Hervey co-founded multimedia division at a company called Collabora, "in order to improve Pitivi, GStreamer and the GNonlin plugins". In late 2008, Collabora hired two new developers to work on Pitivi and related technologies. This boosted development that stalled during 2005–2007.

At that time the underlying framework for Pitivi consisted of GStreamer and GNonlin. However, it turned out that in order to create a video editor on top of this framework, a lot of additional work had to be done. To solve this problem, and to make the solution re-usable, Hervey created a library called GStreamer Editing Services (GES) in 2009. In 2011, it was officially announced that Pitivi's next version (at that time version 0.15) will be based on GES. But the subsequent version (0.91, released in 2013 and codenamed "Charming Defects") involved also other substantial changes such as porting from GStreamer 0.10 to GStreamer 1.0. The 0.91 release marked a large change in architecture (for an overview of the architecture, see Appendix 2) of the program which brought with it number of issues. The main focus of development at the time of writing is at stabilization that is necessary for releasing the 1.0 stable version.

From this brief account of the project's history, it is clear that underneath its user interface, Pitivi hides a complex architecture of elements it is built with. The user interface (for its picture, see Appendix 3) is a result of combining GTK+ – a toolkit that provides ready-made buttons, widgets or other basic interface elements – and Clutter – a library[89] used

89 A library is a specific type of program that is in most cases invisible to the end-user but which is essential in

to draw timeline (which is a central interface component of any video editor but which is also very specific and so it is not provided by GTK+ as a ready-made element). Thus, GTK+ and Clutter are used to create a convenient interface (also called frontend) through which users can trigger operations that are performed by lower level libraries (backend).

The most recent component of the Pitivi stack[90] is a library called GStreamer Editing Services (GES). This library filled the space between Gstreamer which provides very general functionality and the Pitivi frontend which is designed to apply specific video editing operations. GStreamer is a library that is not intended only for video editing, but for media handling in general and thus, a lot of work must be put into condensing GStreamer functionality into operations that are conventionally used in video editing. To make this work re-usable, it was put into a separate library (GES), which can be used by any video editing program. This effort aims to simplify the work of programmers to an extent that writing a video-editing program should be reduced to creating a user interface that applies operations already provided by GES. The whole backend (which represents the biggest investment of effort when developing a video-editing program) can be simply reused in other video-editing programs, thus doing away with the need to write functionally equivalent software for every new program over and over again. The extent of the achievement is fully understood only when we consider the number of projects that aimed at developing an open source video editor in recent years. In an overview (see Appendix 4) created by one of Pitivi maintainers, there are 54 projects, 9 of which are still active.

The effort that aims to transform what has been initially at the core of Pitivi itself (and thus specific for only this program) into a library that is able to provide functionality to many other programs is indicative of an "upstream first" approach to software development, a convention that forms a cornerstone of the programming theory (in Naur's sense described above) with which Pitivi maintainers operate. This principle was described by one of the Pitivi maintainers at the GNOME User and Developer Conference (GUADEC) 2013:

> It basically means: no hacks. You discuss with upstream such as upstream GStreamer, upstream GTK and everything and you work out solutions in cooperation with them and you don't put some stupid hacks in your application downstream instead of fixing the problem for everybody.

---

that it provides general functionality that is used by programs that interact with users. The existence of libraries allows for modularity (libraries specialized in function may be combined) and reuse (the functions provided by a library may be used by a wide variety of programs) of components in software design.

90 "Stack" is a term that denotes all components that form a working program. In this sense, Glib, Gstreamer, GNonlin, GTK+ or Clutter are all part of the Pitivi stack.

What is interesting here is that in the world of free and open source software, the word “hacking” usually comes with positive connotation (in general, it denotes a creative and clever leverage of formal systems). But in this case, “hacks” are associated with stupidity. It denotes a way of solving problems that is faster and easier but that will eventually result in fragmentation and hindering of development in the wider community. On his blog, the same maintainer likens the difference between the “upstream first” approach and “stupid hacks” to a difference between “being a good citizen” and “doing your own thing in your corner”. This is why many Pitivi developers do not actually work on Pitivi itself, but rather on some of the underlying libraries.

On the lowest level of the stack, Pitivi is based on the Glib library, that provides basic types and algorithms with which Gstreamer, GES, or Pitivi are built. But the basic functionality related to media handling is provided by GStreamer. It is a multimedia-centric library that forms the heart of the Pitivi stack. GStreamer handles multimedia by sending streams of data through series of nodes. Nodes represent spots where data can be altered by various codecs, effects, or (de)muxers which are available in GStreamer as a set of plugins. This means that the elements which make GStreamer useful are packaged independently and may or may not be installed together with it. This seemingly odd design is enforced by licensing issues with various codecs and other elements.

The GStreamer maintainers differentiate plugins into four categories based on source code quality and licensing: base, good, ugly, and bad. The base and good plugins are unproblematic with regard to both licensing and code quality, as the maintainers put it quite humorously:

> A collection of plug-ins you’d want to have right next to you on the battlefield. Shooting sharp and making no mistakes, these plug-ins have it all: good looks, good code, and good licensing. Documented and dressed up in tests. If you’re looking for a role model to base your own plug-in on, here it is.[91]

As indicated by their names, the “ugly” and the “bad” plugins are the problematic groups, each in its own way. Bad plugins simply have bad code quality and can not be relied on. In this sense, they are technologically inferior to the rest of the plugins. By labeling them as bad, GStreamer maintainers renounce their responsibility for their performance and support. They also renounce any commitments to fixing issues that are reported. Performance of these

91 *An explanation of the various plugin modules and how they were split up*. GStreamer documentation page. Accessed: 2014-07-23. Available at: http://gstreamer.freedesktop.org/documentation/splitup.html.

plugins has low priority so the only way that issues can be fixed in this area is when someone else steps in and provides a patch[92]:

> Don't bug us about their quality - exercise your Free Software rights, patch up the offender and send us the patch on the fastest steed you can steal from the Confederates. Because you see, in this world, there's two kinds of people, my friend: those with loaded guns and those who dig. You dig.[93]

Here, the GStreamer maintainers emphasize the "Free software rights" that are granted by the use of free software licenses and that permit modification and redistribution of the code. However, the maintainers expect the contributors to exercise only the right to modify the code, not to redistribute it. The patch should be sent to them for review and redistribution. While this practice may seem as free-riding on the work of others, it keeps development from fragmentation into a confusing number of parallel versions while also providing a trusted source of reviewed official versions. At the same time, the author retains his authorship and is provided with a distribution channel that reaches the widest possible audience. In the context of the "upstream first" principle discussed above, the maintainers expect contributors to be good citizens.

However, the right to redistribute the code is still present and serves as a safety that counterbalances the maintainer's power stemming from the review and redistribution process. If a contributor (or more likely group of contributors) is unsatisfied with the way the current maintainers operate, it is always possible to duplicate the whole source code of the developed program and start maintaining a parallel project. This practice is called forking.[94]

Ugly plugins, the fourth and last remaining category is characteristic by licensing issues. GStreamer maintainers retain their responsibility for fixing issues that are reported against this group of plugins, but note there are difficulties in distributing them:

> There are times when the world needs a color between black and white. Quality code to match the good's, but two-timing, backstabbing and ready to sell your freedom down the river. These plug-ins might have a patent noose around their neck, or a lock-up license, or

92 Patch is a term denoting a modification of source code that is made in order to fix a particular issue.

93 *An explanation of the various plugin modules and how they were split up*. GStreamer documentation page. Accessed: 2014-07-23. Available at: http://gstreamer.freedesktop.org/documentation/splitup.html.

94 Although in theory possible, forking is rather rare and is seen as a last resort (because it means duplication of efforts) in cases where every other option to resolve the differences failed while also placing the burden of maintaining the new project on the initiators. Generally, forks are accompanied by argumentation that justifies the duplication of efforts that developing two parallel versions of a program implies. The argumentation also serves to draw contributors that have to decide if they stay with the original project or become part of the new one.

> any other problem that makes you think twice about shipping them.[95]

To fully understand the extent of problems that licensing issues pose for the GStreamer project, it necessary to note that key functionality is often accompanied by restrictive licenses or patents.[96] As a result, the GStreamer maintainers (and many others) face a dilemma:

> Due to this situation, many companies, including major GNU/Linux distributions, get trapped in a situation where they either get bad reviews due to lacking out-of-the-box media playback capabilities (and attempts to educate the reviewers have met with little success so far), or go against their own - and the free software movement's - wish to avoid proprietary software.[97]

The omni-presence of the problem can be illustrated by a conversation I witnessed on one social network, where a blogger took issue with the choices made for shipping software with the Fedora Linux distribution and suggested an application that is made to easily install the missing software:

> For Fedora new-comers Fedy provides an easy way to install all (almost all!) the software that Fedora cannot ship –for reasons that only Red Hat's legit department knows ;)

In the comments to the post, I found an answer from a Fedora contributor:

> I think the reasons are pretty well known, so not only the RH legal department knows:
> 1) you don't have a right to redistribute the software (the license doesn't allow it)
> 2) technologies in the software are patent protected
> 3) the software cannot be packaged in a way it doesn't violate Fedora packaging guidelines (bundling,…)
> Only the third case is something Fedora can do something about.

The blogger expresses his confusion with a situation, in which software that cannot be shipped (for legal reasons) directly with a Linux distribution, but can be installed on the same Linux distribution afterwards using a specific program. The Fedora contributor attempts to explain the sources of constraint, pointing out that in most cases, there is nothing the

95 *An explanation of the various plugin modules and how they were split up*. GStreamer documentation page. Accessed: 2014-07-23. Available at: http://gstreamer.freedesktop.org/documentation/splitup.html.

96 For example, patented software is necessary in order to process files in the omnipresent MP3 format. Of course, the MP3 format has its alternative in the free OGG Vorbis format, but the use of this format is far from standard. Contrary to licenses which affect exactly defined parts of source code and an ex post change of their terms is very hard if not impossible in most cases, patents are problematic because they affect certain technology in a more general sense and allocate the power to decide into the hands of one formally defined party. Even though the party may permit a free use of the patented technology, there is no guarantee that this will not change in the future, possibly as a result of transferring the patent to another party. In this sense, a call made by Polk Wagner to supplement licenses as ways of organizing open source software development with patents (Wagner, 2003, p. 1031) seems inadequate.

97 *Licensing your applications and plugins for use with GStreamer*. GStreamer documentation page. Accessed: 2014-07-23. Available at: http://gstreamer.freedesktop.org/documentation/licensing.html.

distribution maintainers can do about it. This indicates the dilemma has to be negotiated or explained between developers and users, supporting the statements made by the GStreamer developers.[98]

In attempting to overcome this dilemma, GStreamer maintainers opt for a lesser evil – they choose to use the GNU Lesser General Public License (LGPL) which is a free software license that permits (as opposed to the classic GNU General Public License) distribution together with proprietary software.[99] The proprietary or patented software is then packaged into a separate body of plugins so that the users may decide (according to their needs, local legislation, or their attitude toward using proprietary or patented software) whether they want to install and use it.

However, the problem does not stop with GStreamer, it applies to all applications that use the library to handle multimedia files. As a result, GStreamer maintainers recommend to all developers of applications that use their framework to use the LGPL license too in order to be able to use the functionality contained in ugly plugins. But this is not the only option that developers of other applications have. They may also use the GPL license and supplement it with a clause stating that GStreamer plugins are exempted from the obligations of the license. However, GStreamer maintainers state that using a GPL license with a clause would result in hindrances to sharing source code among projects and therefore recommend the standard LGPL:

> Our suggestion among these choices is to use the LGPL license, as it is what resembles the GPL most and it makes it a good licensing fit with the major GNU/Linux desktop projects like GNOME and KDE. It also allows you to share code more openly with projects that have compatible licenses. As you might deduce, pure GPL licensed code without the above-mentioned clause is not re-usable in your application under a GPL plus exception clause unless you get the author of the pure GPL code to allow a relicensing to GPL plus exception clause. By choosing the LGPL, there is no need for an exception

98 The ambiguity regarding licensing and patents in this area is so deep that Fluendo, one of the companies involved in the GStreamer project has partly built its business model around it: “While Linux OS does provide multimedia functionality in terms of free media players, unlike major paid operating systems like Microsoft Windows and Apple OSX it does not come with licensed codecs pre-installed. Without these codecs, many Linux users and organizations unknowingly violate intellectual property laws, putting themselves and their organizations at risk. Patent infringement has serious consequences and, especially for larger organizations with many users, the cost can be substantial. … This is where we come in. From proprietary codecs to our robust DVD software, Fluendo legally protects organizations and empowers users to engage with multimedia like never before. Experience unmatched playback quality with the peace of mind that you’re adhering to international audio and video patents.” (*The Legal Risk that Linux Users Face*. Fluendo marketing materials. Accessed: 2015-04-09. Available at: http://www.fluendo.com/corporate-linux-users/.)

99 The classical GNU General Public License introduces the necessity to license all derivative works with it, a characteristic that is often called a “viral feature”.

> clause and thus code can be shared freely between your application and other LGPL using projects.[100]

The GPL and LGPL are standard in free software development and their wide use allows for re-appropriation of source code between projects without the need to negotiate licensing conditions. On the other hand, the inclusion of a non-standard clause into one of the licenses introduces a requirement of negotiation and complicates the “free sharing” of source code among projects. Thus, licensing may be seen as a part of the (legal) infrastructure of sharing in the sense that abiding to standards allows for frictionless performance of the intended practices.

It is then no surprise that Pitivi abides to the standard and uses the LGPL to license its source code. It was actually my first task as a documentation writer to add the information on licensing of Pitivi’s source code into the user manual. To get me started, one of the maintainers gave me his to-do list that he kept specifically for user documentation and from it, I selected what seemed to be the easiest task available: adding one sentence containing a link to the licensing page. In FOSS projects, it is common to tag and leave aside easily fixable problems that newcomers can grind their teeth on. These problems are referred to as a “low hanging fruit” in the sense that by contributing their solutions, newcomers get a fast reward and a sense of accomplishment.

However, it is often not clear for a newcomer that there is a database of issues (commonly referred to as “bugs”) in which it is possible to search for specifically tagged entries.[101] Therefore, the most common first step (and mine was no different) on the way to contribute is to join the project’s chat channel and ask how is it possible to contribute.

> Tim: hi I would like to contribute to pitivi, I am good at python and javascript, can somebody point me to the source code and small task to get started, thanks

In most cases, what follows (during the time I was present I witnessed several newcomers trying to contribute to the project) is a negotiation between the maintainers and the newcomer in which the sides attempt to find a fit between the potential contributor’s skills and interests on the one hand and available tasks on the other. To get some sense of how such negotiation proceeds, I will quote one case at length:

100 *Licensing your applications and plugins for use with GStreamer*. GStreamer documentation page. Accessed: 2014-07-23. Available at: http://gstreamer.freedesktop.org/documentation/licensing.html.

101 In Pitivi’s case, the easy bugs are tagged as “gnome-love” because Pitivi uses the database of the GNOME project – a more general project aiming to develop a whole desktop environment forming an umbrella organization for many smaller software development projects.

Steve: After the setup is done and you make sure you can run the dev version, what I would recommend is starting with writing simple tests with dogtail
…
Tim: Steve, ok I will start by writing the tests :-) any other suggestion that would be useful?
Tim: I mean initially to start with small tasks and gradually shift to bigger ones
Steve: That makes sense
Steve: Be aware that video edition is not a completely trivial task :)
Steve: And yeah, the most useful thing IMO would be to just use the software, find bugs, write tests that highlight them, then discuss with us the best way to fix them
Steve: That way you'll start to learn the codebase while being able to contribute code from the get go
Tim: Steve, ah I see! I would definitely follow you and ping you if I get stuck, well thanks for the help :-)
Steve: OK :)
Steve: Tim, I see your [link to a repository containing source code of an audio processing program], nice to see you're interested in audioprocessing :)
Tim: Steve, heh yep that's a recent one :-)
…
Steve: If you're interested in sound specifically, we've got a regression that needs fixing, it's the "auto aligner" feature
Steve: The work needed there is not really low level audio though, more like gstreamer fixes
Steve: ie the one that fixes that won't get to play with raw samples :)
Tim: cool! could you point to the url?
Steve: [link to a file containing a specific part of Pitivi's source code]
Steve: Tim ^
Steve: hm it's quite lowlevel actually
Eric: Tim, we could look for something more easy for a start, depends how used to python, gtk, gstreamer you are, and what kind of task you would like
Tim: Eric: I have been coding in python for quite some time, so it won't be a problem but gtk and gstreamer i will have to look at tbh! but sure that won't be a big task, so suggest something accordingly, thanks btw
…
Steve: Eric certainly has good ideas :)
Tim: Eric, i have been working for Mozilla QA, and writing unit and functional tests for firefox OS and Selenium tests for their websites
Steve: :)
…
* Eric looking for a cool small task :)
…
Eric: Tim, not sure how interesting this might be for you: [link to a bug report]
…
Eric: or this Gtk-related change: [link to a bug report]

…
* Tim has a quick look at the bugs
Eric: If you click pitivi: Show you can see all of them :)
Ted: Tim, well I guess the question is,
…
Ted: 1- do you like to fix bugs, fix tests, or touch features (though we're probably in a feature freeze now)
Ted: 2- do you prefer UI work or backend work?
Tim: hey Ted : i have been working with testing as i said so i would prefer that more and yes I would like to focus at backend
Ted: awesome
… [setting up the development version]
Tim: Steve, yea so now I can get to do some work! what should I start with?
…
Steve: My opinion is tests :)
Steve: Which means getting familiar with dogtail, finding a bug in pitivi, and trying to reproduce it with dogtail
Steve: Ted will be able to tell you more than I about dogtail
…
Ted: Tim, see [link to a project's wiki page containing information on testing] if you hadn't seen it already
Ted: the dogtail tests are "purely UI", but pitivi also has more "theoretical" backend-related tests
Ted: that page gives a quick overview of both
…
Tim: Ted, I will go through [link to a project's wiki page containing information on testing] and try running some tests, also are there any tests that are to be written, so that i can work on them? or will I have to find bugs and write tests for them?
Ted: there's that sub page for a test suite "wishlist" linked inside that page
Ted: but don't let that stop you, there's certainly a ton of stuff that can come out of your imagination that we haven't thought of
Ted: and Roy supposedly had ideas (when I asked him a year ago :) about stuff that needs to be done for the normal (not dogtail) tests
Ted: but hasn't written them on that page yet
Tim: ok I will try to add some tests, here, and keep in touch, about any new things to be added Ted
Ted: cool, thanks :) we'll be happy to help you with pieces of advice etc.
Ted: play around, try the app, see what works and what doesn't, what annoys you, etc.
Tim: but really its 5:41am in my timezone and i haven't slept the night, so will work on it tomorrow, btw thanks a ton Ted and Steve
Ted: good night and good luck!
Tim: thanks!

In his first message, Tim provides the maintainers with information about his skills that could be useful for contribution to Pitivi. Based on that, the first option that Steve provides is writing Dogtail tests. This means that Tim would write automated tests to ensure that once bugs are fixed, they do not appear again. However, a look at Tim's personal repository, Steve discovers that Tim previously worked in the area of audio processing. Thus, Steve suggests an audio related task but by browsing through information about it, reminds himself that the issue concerns lower level libraries that are written in the C programming language (whereas Tim already told him that he knows two different languages: Python and Javascript).

At this point, Eric joins the conversation and links two bug reports that could be suitable for a new contributor with Tim's skills. Ted then attempts to specify Tim's expectations by asking him, what area he is interested in. Through that, they reach automated testing again. Taking it as a valid option, Ted provides Tim with more information on testing by linking to the appropriate page form the project's wiki. At the same time, Ted emphasizes that task suggestions contained in the wiki should not be taken as a final list of tasks available. There is a repeated suggestion to try out Pitivi, use it, play around with it, find out what works and what does not, see what is annoying. By this, the maintainers place emphasis on the contributor's own experience with the developed program and assume that the contributor will be more motivated or find it more interesting to fix issues that he himself discovered or that he finds annoying. This corresponds to the first principle that Eric Raymond lists in his essay *The Cathedral and the Bazaar* (Raymond, 1999), a text that has been formative for the open source movement. In the first principle, Raymond states that "every good work of software starts by scratching a developer's personal itch". Of course, Tim does not have a personal itch with regard to Pitivi (at least not yet) and that's why he asks for guidance. The maintainers readily pull out lists of tasks in the form of known issues and wiki pages, but also try to point the newcomer in a direction, where he could obtain his personal itch to work on.

The second (and often bigger) challenge that newcomers have to face is setting up the development environment. The environment consists mainly of the latest development version of the program (and the libraries it uses). The development version serves as a shared reference point, this time not in the form of source code, but in the form of a running interface. By running the development version, the contributors are able to grasp and use the result of their work. This is necessary for testing the work that has already been done and determining what should be done next (where else to find one's personal itch than in the latest version?). For comparison, I quote here from the same case as the one few paragraphs above:

Tim: hi I would like to contribute to pitivi, I am good at python and javascript, can somebody point me to the source code and small task to get started, thanks
Eric: Tim, /[link to instructions on building the development version/]
Eric: Tim, what Linux distro do you use?
Tim: fedora 19 x86 Eric 64bit
Steve: Tim, pitivi is a complicated project, with many moving dependencies
Steve: For that reason, we have a script that allows contributors to set up the environment automatically, which Eric linked to you
…
Steve: After the setup is done and you make sure you can run the dev version, what I would recommend is starting with writing simple tests with dogtail
…
Tim: so Eric : i ran the script here /[link to instructions on building the development version/] and have pitivi-git dir and [ptv] a virtualenv kind of a thing i guess?
Eric: good
Eric: cd pitivi-git/pitivi; bin/pitivi
Eric: works?
Eric: the [ptv] is just a bash with some environment variables set
Tim: well i have just glib and prefix dirs Eric
Eric: if you run bin/pitivi it should ..fail, can you confirm?
…
Tim: Eric, : i mean in pitivi-git/ i just have glib and prefix , and prefix is empty too after running the script is there anything else that needs to be done ?
…
Eric: I really have to go, see you tomorrow
Steve: bye Eric o/
Tim: bye Eric : i will figure it out :-)
Steve: Tim, did you solve the dependencies as indicated on the website ?
Tim: yes that I need to I think I skipped some steps
Steve: Solve the Dependencies.
Steve: Get this script, save it, make it executable and run it: [link to the script]
Tim: cool! just doing
Steve: just two steps :P
Steve: on f19 it should work, we’ve been developing with f18 - 20
Steve: I skipped 19 but I believe Roy used it
…
Steve: Tim, no problem yet?
Steve: (aliasing make to “make -j4”) makes the whole process faster
Tim: yeah downloading packages
Steve: Should have told you
Tim: Steve, i get [link to a copy-pasted output from Tim’s command line] on running the script, it isn’t able to clone pitvi for me i guess ?
Steve: Thats not the first run right Tim?
Steve: the previous one must have failed

Tim: Steve, yes
Steve: Can you paste the output of the previous one ?
Steve: The run that checked out the libs etc ?
…
Steve: Tim, the file you're showing me is a new run
Steve: You should just remove the newly created directory,
Steve: start the script once again, and paste me the output
Steve: ie remove "pitivi-git"
* Tim follows steps
Tim: it takes too long, all CC CCLD and make on my screen, is it correct, or i have ended up firing something weird ?
Steve: It doesn't take too long
Steve: you're compiling gstreamer, which is a huge beast
Steve: Not the kernel by far but still
Steve: around ~1.5 millions LOC [lines of code]
…
Tim: yea finally something I got and it stopped, should I overwrite ? [link to a copy-pasted output from Tim's command line] ??
Tim: woah! cool I think I got the things right now :-)
Steve: yep, the duck pretty much means "good to go" ;)
Tim: would it take this much time always when i run the script ? Steve
Steve: Of course not
Steve: Once it's built it's built
Steve: you can ctrl + D and rerun the script to get back into the env
Tim: yea! my CPU would have died then :P
Steve: aha don't worry
Tim: Steve, yea so now I can get to do some work! what should I start with ?

Typically, to get the latest development version of a program running, one has to download its source code and compile it. But as Steve puts it, Pitivi is a complicated project – it depends on several underlying libraries, some of which can be used at latest packaged version (therefore no compilation is necessary) and some of which must be compiled from the source code. To simplify setting up of the development environment, Pitivi maintainers created a script that automates the process of checking for the required versions, downloading and compilation of the dependencies and Pitivi itself. However, the script does not take care of everything, there are several issues that the newcomer has to solve manually. It is necessary to install one of the Linux distributions that Pitivi developers typically operate with. The choice of distribution affects what software (and in which versions) is available for the contributor and therefore is crucial for solving dependencies. In this case, the standard distribution used for development is Fedora. This is so, because Pitivi uses the Gimp toolkit (GTK+) for development and so distributions that package the latest versions of GTK+ are preferred. Fedora is tightly integrated with GNOME, to a point where its development cycle is synchronized to that of GNOME. As a result, Fedora is reliable in shipping the latest stable GTK+. This is the main reason why Fedora is used as a standard Linux distribution by the Pitivi developers.

As the quoted conversation indicates, running the script is also not a trivial matter. Tim went through several iterations before he figured out how to correctly run it. While doing so, he exchanged with others the output from his command line that he could not interpret properly himself. By doing so, he tapped into the knowledge that the maintainers pool in the Pitivi IRC channel. The knowledge present there allowed him to treat properly information which would otherwise remain cryptic for him. Tim is not even sure how the correct result of his action should look like – this is demonstrated by him asking if he "ended up firing something weird?". He is assured by the maintainers that what he sees on his screen is normal and that in subsequent runs, it will not take this long. Finally, the running script on his screen stops and he is greeted with a picture of a duck made of ASCII letters. This means that he is "good to go".

```
====================================================================
                   BATTLECRUISER OPERATIONAL
                          >(°)__/
                           (_~_/
                         ~~~~~~~~~~~~
====================================================================
```

However, to contribute to Pitivi, Tim will need more than its running development version. He will need tools[102] to get the work underway. Text editors allow him to browse and edit the source code, Git allows him to manage source code and share it, compilers allow him to turn the source code into an interface, debuggers allow him to find errors, Bugzilla allows him to see the issues reported against Pitivi, Wiki allows him to make drafts and write developer documentation, an IRC client allows him to connect to the Pitivi channel and the list goes on. The common denominator is that the tools are typically developed consistently with the free and open source software model. The reasons for this differ but it seems they follow the reasoning typical for the two branches of the movement – either it is an issue of being self-sufficient with the tools abiding the same moral standards (free software), or it is a practical

102 It should be noted that some tools are needed even to set up the development version. This applies to Git or compilers for example. As a result, the difference between some tools and dependencies may not be initially clear. The clarity comes with distinguishing the roles of developers and users. Dependencies are packaged programs that must be installed for a given program to run regardless if it is run by a developer or an end user. Tools, on the other hand, are not required from end users. Another way of saying this is that official versions of software require only dependencies while development versions require dependencies and tools.

issue of being able to see the guts of the tools and being able to possibly influence the direction of their development (open source software).

Both of these reasons point to the recursivity involved in FOSS projects – the tools that are used in one software development project constitute development projects of their own, while also using a set of tools. A distinct way of demonstrating this recursivity is pointing to the project that develops the version control tool called Git, where Git itself has been used for version control from the very beginning of its development. Linus Torvalds, the originator of Git described the process in a following way:

> You can actually see how it all took shape in the git source code repository, except for the very first day or so. It took about a day to get to be "self-hosting" so that I could start committing things into git using git itself, so the first day or so is hidden, but everything else is there. The work was clearly mostly during the day, but there's a few midnight entries and a couple of 2 a.m. ones. The most interesting part is how quickly it took shape; the very first commit in the git tree is not a lot of code, but it already did the basics - enough to commit itself. The trick wasn't really so much the coding but coming up with how it organizes the data.[103]

In general, we can talk about FOSS projects being "self-hosting" in the sense that the production of one project is used by another. This is what Christopher Kelty is pointing at when he writes about recursive publics (Kelty, 2008, p. 3).

To describe how tools are used in the Pitivi project, let us first pick a very basic representative – text editors. There is a wide variety of text editors to choose from. There are command line editors and there are editors with a dedicated graphical user interface; there are editors supporting a number of modes or modeless editors – to name some categories according to which editors can be classified. Furthermore, text editors are typically highly customizable and extensible. For example, editors can be set up to use dark or light color schemes for their user interface, to highlight current lines, to display line numbers, to highlight the syntax of a certain programming language, to wrap lines longer than 80 characters, to display spaces, to use regular expressions for searching, to use various plugins, or to complement the editor with a terminal and debugger to create a so-called integrated development environment. Needless to say this list scratches only the surface of possible customization. The width of customization possibilities is not surprising when we consider the fact that the interfaces of text editors are the ones in which developers spend most time and

103 *Interview with Torvalds about Git on the occasion of 10 years of Git*. Interview published by Linux.com. Published: 2015-04-06. Accessed: 2015-04-08. Available at: http://www.linux.com/news/featured-blogs/185-jennifer-cloer/821541-10-years-of-git-an-interview-with-git-creator-linus-torvalds.

that software developers – their users – also often carry the knowledge and skills to modify them.

However, customizations are in most cases focused on the editor interfaces, not on the form of the resulting text files where standards are enforced, although customization can help following the standards. For example, when I submitted my first work, the maintainer doing the review told me that my files contained trailing whitespaces. This meant that I forgot space characters at the end of some lines. These characters are generally considered redundant and may cause problems in certain situations and so it is considered a good practice to remove them. After I got the review I had to set up my text editor so that it displayed trailing whitespaces (which were invisible until then) to avoid the issue.

Another instance emphasizing the existence of a standardized output was marked in the discussion among two of the maintainers I witnessed on the Pitivi IRC channel. The subject of the discussion was the “80 column rule” which states that any line in a text file should not be longer than 80 characters.[104] This rule is so common, that many text editors assume it by default. But apparently, the rule makes more sense with regard to some programming languages (like C for example) than others (like Python, which is used in Pitivi). Also, wide computer screens are common today and enforcing the rule leads to leaving much of the screen space unused. In the end, it is up to the project maintainers to make an agreement. There were two justifications for enforcing the rule for the Pitivi source code: (1) “I want to be able to dual screen on modest sized computers.” (2) “Some hackers might have their editor setup with 80 chars assumption.”

The first reason is practical – wide screens are common these days, but they are usually tied to a place because of their size. No one carries a 22 inch monitor around on a daily basis. When traveling, for example, more modest screen sizes are still standard. Furthermore, the developer work-flow usually involves displaying two text files side by side on a screen (“dual screen”) and this requires making the text limited in width. In this context, the old rule that has its roots in the 80 column IBM punch cards still has some relevance.

The second reason is concerned with standardization explicitly. Even though the maintainers are aware that the number of 80 characters is arbitrary (stating that “socrates

104 Unlike the mainstream text processing programs contained in various office suites, text editors used by developers do not employ the metaphor of a page which restricts line width according to the specified paper format. The only limiting factor for editors designed for software development is the size of a computer screen which is nowadays widely variable. Thus, limiting the width of the line gets to be more a matter of agreement than a matter of external constraints.

could troll like this: it could be 74”[105]) they still use it as a reference point as it seems it is the most commonly used option. By applying the 80 column rule, the project maintainers intend to comply with the expectations most experienced developers could have, therefore lowering the barrier (of setting up or adjusting the development environment) to entry for them.

All in all, setting up a development environment is far from a trivial matter. Initially, it took me days to set up my environment in order to contribute to Pitivi. I had to install a new operating system (Fedora) on my computer, get the development version of Pitivi and make it run, install and configure Git (to be able to see and create revisions in source code), make a Bugzilla account (to be able to file and comment on issues), pick a text editor and configure it (actually, I tried a number of them – from Gedit and Geany to Vim and Emacs and back to Nano). All of this before I even started to work with the Pitivi user documentation I wanted to contribute to. But the very first thing I had to do to make this process started was to contact the maintainers in the Pitivi IRC channel.

4.1. Knowledge Channeling

The Internet relay chat[106] is designed to facilitate synchronous communication, that is, communication where both parties are present at the same time. Internet relay chat simulates the space for communication in a way that not being present means missing the communication – no history is recorded for those who join in to read. Pitivi maintainers circumvent this limitation by using “bouncers”, programs that run constantly on a server and through which they connect to their chat accounts. This allows them to be connected to IRC channels even when they are asleep and their computers are shut down. When they connect again, they can read the backlog (what happened when they were away) and see if someone tried talk to them. By this, they are using IRC also as an asynchronous means of communication.

The basic organizational unit of IRC is a channel, which essentially denotes a chat room where those who are present see the communication taking place. Channels are differentiated by topics of interest: there is a Pitivi channel, a Gstreamer channel, channel of the GNOME documentation team etc. People are known to be present in certain channels so when I wanted to talk to them, I had to go where they were. Sometimes I went to a channel because of its overall specialization and sometimes I aimed for a particular person. In this way, the IRC

105 To “troll” means to make controversial statements with the intention to spark heated discussion and conflict.
106 The IRC is a protocol of the application layer of the Internet protocol suite. It is located in the same layer as HTTP, but whereas HTTP is used to transfer static text, IRC is used to transfer text messages.

channels represent reservoirs of knowledge. This is what I noted when I was beginning my work on the user manual:

> I needed to make sure that the new page I created gets to the .pot file for translators to work on. No one at Pitivi knew for sure what was needed so I joined the channel #i18n [internationalization] and asked. I explained my problem and provided info on which module it is related to. Within an hour I got my answer and a place where to check that everything went ok. Went back to #pitivi and told others the result.

Such trips to various channels in order to retrieve specific information are a common practice in this environment. Since one can be connected to multiple channels simultaneously, it is not even necessary to leave the current channel. All that is needed is to know the name of the channel and of the network it belongs to.[107]

What takes place within a channel is similar to a screenplay text. The communication is sequentially structured, actors enter and leave, say something and do something[108] while they are present. All these acts leave traces in the backlog. However, it is not necessary to communicate actively the whole time one is connected. It is common to be present in the channel just to monitor it for relevant discussions (a practice called “lurking”) or to be available for others. For example, at the time of writing, there are around 30 people (not including bots) connected to the Pitivi channel even when no discussion is taking place. The lurking practice was jokingly referred to in one conversation:

> Chris: sounds like it’s probably doable, but i’ll have to do some more poking and research
> Chris: it’s not an 8am poke around before work task ;)
> Steve: Indeed :)
> Steve: Anyway, hope you’re having fun with all this, always interesting to discover new techs :)
> Steve: (and fix stuff on the way)
> Chris: oh yeah i’m having fun diving back into gtk+ stuff
> Chris: did a little work on gimp aaaaages ago (in powerpc mac days)
> Steve: And jack is always very helpful, even when not giving out his code ahaha
> Chris: in the meantime, adding more irc channels to my defaults :D
> * chris idle ALL the channels!
> Steve: Yeah I think I remember you telling me this :)
> Steve: Ahaha that’s the spirit
> Chris: :D

107 IRC channels are grouped into overarching networks which provide servers for their operation. During my fieldwork, I was present in two networks: Freenode and GimpNet. The former hosts channels of FOSS projects in general while the latter hosts channels that belong specifically to the GNOME project.

108 There is a way to write a message so that it refers to the author from the position of a third party. This is often used to let others know what one is doing or why one is not responding. Such messages begin with an asterisk and look like this: “* [nickname] is still reading the backlog”.

Steve: 17 channels open here :)
Steve: All these conversations I will never read :)

In this conversation, Chris was returning after a long period of time to work on code that uses the GTK+ toolkit and investigated a task he wanted to accomplish. We can see that the return and investigation naturally involve adding related IRC channels into a set of channels with which connection is automatically established (the "defaults") when his IRC client is opened. But this does not necessarily mean that he will read all the conversations that take place within the channels. He just wants to have those channels ready when he will need to make an inquiry. This is another example of using IRC channels as reservoirs of knowledge.

Communication inside a channel often oscillates between bursts of activity and pauses in conversation. This is obviously a result of developers being from different timezones and having differently structured work time. However, there are also other mechanisms at work. First, an ongoing discussion draws attention to monitoring the channel. Second, when someone starts a discussion, it tends to sustain itself by provoking new inputs. Third, when participating on a discussion, others can see the actor's availability and often start a new conversation.

Thus, communication on IRC is usually multi-threaded. There can be two or more simultaneous conversations taking place at the same time. Multi-threadedness means not only that there are several dyads communicating at the same time, but also that one can participate in more than one conversation. Such situations are demanding in response speed. However, no one gets offended when one does not respond promptly, others can see how many conversation one is having. But responding slowly represents a risk of losing attention of one's counterpart and having the conversation interrupted. Everyone is multitasking and so one is quickly driven away from the channel if nothing new happens in a while.

There is one mechanism though, that allows for making others pay attention. Writing someone's nickname in a message generates a notification in that user's desktop client. The notification system attempts to draw attention to the discussion. This can get annoying when over- or misused and so I have witnessed the maintainers having to calm down eager newcomers several times. On the other hand, some messages are not aimed directly at anyone, they just function as a way of letting others know what is on one's mind or what one is currently doing. Similarly, solutions to problems are posted even if no one requested them directly, to provide resources for future use and collect feedback on the solution. However, the texts flowing through IRC are not very persistent. The individual maintainers occasionally copy and paste parts of conversations into their private notes, but there is no systematic archiving taking place.[109]

There is a different kind of infrastructure in place exactly for this purpose – the project's wiki and individual blogs. This infrastructure embodies what could be called the phenomenon of self-documentation which is an integral part of free and open source software. This means that problems and their solutions get recorded and elaborated upon, even when no one requested this directly. There is a conviction that the most effective way to deal with problems is to archive them and their solutions publicly. This allows others (and sometimes the writer himself after a longer period of time) to quickly find information or solutions to problems they are facing. Blogs and wiki pages are better suited for this purpose as they offer more accessible (through the standard HTTP protocol) archiving where (contrary to IRC for example), texts are persistent by default.

Furthermore, blogs and wiki differ in that the first represents essentially simple web pages written in the standard HyperText Markup Language (HTML), while the second not only uses its own markup language (which is eventually translated to HTML), but it adds functionality for collaborative writing and version tracking. Thus, blogs are usually used as a simple means of archiving and expression of individuals, while wiki serves as a more "official" source of information about a project. Thus the project's wiki contains pages that serve as information resources dealing with common problems such as how to set up the development environment, how to use source code management or debugging tools effectively, what is the general architectural model of the developed software, or who are the main contributors. An aggregation of such pages creates a pool of knowledge that can be readily referenced to for example in chat discussions (each wiki page has its own HTML address). As a result, the developers don't have to repeatedly explain how to solve the most common problems.

However, the wiki does not work only as a convenient resource for referencing. This function would probably be served just as well by blogs, which are easier to establish and run than the whole MediaWiki platform.[110] There is more to its functionality that balances the

109 This is, however, project specific. Some projects, like the Ubuntu Linux distribution, archive their IRC channels and keep the archives accessible from their website.

110 MediaWiki is a software package originally designed by Magnus Manske to run Wikipedia. However, today it is also widely used in other projects, not necessarily affiliated with Wikipedia. As it is licensed under the GNU GPL license, it is widely used in free and open source software development, where it is common for bigger projects to have their own wiki. Also, the MediaWiki is by far not the only software package with which a wiki can be established. There are numerous other packages among which MediaWiki is probably the most well known.

effort. The differentiating feature is that Wiki pages allow multiple contributors to share work material while all modifications to it are systematically recorded and form a reversible history. This is a result of some of the design principles with which the platform has been built. These design principles have been proposed by Ward Cunningham, the originator of the initial Wiki Wiki Web which served as a model for later wiki packages including MediaWiki. Within the design principles, there is a set of claims that interlock in what could be called "the value of openness":

> Open – Should a page be found to be incomplete or poorly organized, any reader can edit it as they see fit.
> Organic – The structure and text content of the site are open to editing and evolution.
> Observable – Activity within the site can be watched and reviewed by any other visitor to the site.[111]

Translated to the design of a software package, the first two principles mean that there must be a system for free creating and managing user accounts. But by using the word "reader" with regard to editing, the implications seem to go further, to allow even those without a user account to edit pages. Indeed, this is the case with the biggest project using the MediaWiki package – Wikipedia. Also, the word "evolution" used in the second principle hints to the expectation, that there will be a history behind every page. Combined with the third principle, this leads to the design of an archiving function which records every set of edits, assigns them to a time and an author, and make them comparable with every other version of a page (see Appendixes 5 and 6 for examples of page differences and history logs). But to be observable to a full extent, not only editing must be recorded, but also rationalizations behind it. Thus, for every page, there is a place for discussion which is recorded and represents another dimension of the page's history.

However, it is interesting to see that Cunningham explicitly states that some principles, although followed by later designers of wiki packages, were not his primary concerns. These principles are explicitly formulated as follows:

> Trust – This is the most important thing in a wiki. Trust the people, trust the process, enable trust-building. Everyone controls and checks the content. Wiki relies on the assumption that most readers have good intentions. But see: [link to a page called Assume Good Faith Limitations]
> Sharing – of information, knowledge, experience, ideas, views…[112]

111 *Wiki Design Principles*. Accessed: 2014-12-04. Available at: http://c2.com/cgi/wiki?WikiDesignPrinciples.
112 *Wiki Design Principles*. Accessed: 2014-12-04. Available at: http://c2.com/cgi/wiki?WikiDesignPrinciples.

Considering that Wikipedia is seen by many as the hallmark of the culture of sharing, it is ironic to see the originator of the design principles upon which it was built to state that sharing was not his primary concern. But the principle of trust seems to be the problematic point here. The underlying assumption that most readers have good intentions explains the design decision to make wiki pages open for editing by anyone and place the review process after the editing has been done and published. The fact that a link to a page discussing limitations of the assumption is enclosed right after explanation of the principle is indicative of Cunningham's reservations about it. Nevertheless, this principle was followed by other developers and so the current wiki packages inherited this design decision.

However, the Pitivi wiki has different rules for contribution than the what is standard in Wikipedia, where anonymous users (in the sense that they do not have user accounts, but are identifiable by an IP address) may edit pages. Their modifications are only afterwards subject to review and revisions. This creates conditions, in which vandalism and spamming are possible and take hold in the time period between editing and review. Accordingly, vandalism and spamming are cited in the lockdown policy of the Pitivi wiki as main reasons for restricting the rights to edit pages:

> Fighting spam and vandalism has always been a problem in our wiki, and it has been particularly tedious in 2010-2011 where a lot of spam consisted of "throwaway" user accounts made to create lonely pages.
>
> Those pages would typically not be seen by most visitors because they were not linked from any other pages (except the RecentChanges and LonelyPages special pages), and thus would fly under the radar.
>
> === Some statistics === Before the new lockdown policy was enforced, in 2011 there were: * 831 registered users… but only ~10 were real/legitimate users! * 1156 pages… but only 108 were real content pages![113]

As a result, Pitivi developers reached an entirely opposite conclusion – that most users don't have good intentions and they express it by saying: "managing accounts is perfectly acceptable and vastly more efficient than managing spam". This essentially means that while keeping spam under control is not possible, it is necessary to keep under control (manage) user accounts, which would otherwise be entirely up to users. Thus, only users that got in touch with the Pitivi developers and obtained a user account can edit pages.

But there is a reason for going through this trouble to keep the project wiki running. In

113 *Lockdown Policy*. Page in the Pitivi wiki. Last edit: 2014-02-22. Accessed: 2014-12-01. Available at: http://wiki.pitivi.org/wiki/Lockdown_policy.

most FOSS projects I have dealt with (and Pitivi is no exception) wiki pages are used extensively for documentation, be it for users or for developers. Having the MediaWiki platform in place allows for pooling information from various contributors that, ideally, form a manual. Furthermore, wiki pages can play vital role in the prototyping process. Although I haven't really witnessed wiki to be used for this purpose during my presence in the field, I found several older pages made specifically for the purpose of developing complex design concepts (like a plugin system, proxy editing or rendering profiles) and user interface mockups. These wiki pages contain a lot of rationalization, examples of how the problem is solved in other programs and relevant use cases. Often, they also contain a different medium than digital text as visualizations of how a problem solution would appear in the user interface are common.

Using wiki for prototyping is possible because the implementation of the design principles of openness, organicness and observability. The implementation results in key functionality of collaborative writing and version tracking which are necessary for the prototyping process. It must be clear who made which set of changes, and with what rationalization. These changes must then be available for others to review and make their own sets of changes (including rationalizations) that could be differentiated from the original one. Prototyping then takes place through this iterative process of modification (which in this case includes also publication) and review. In other words, wiki pages contain proposals that other contributors can review and further develop, retaining their individual authorship even though the pages are the result of collective effort.

However, all this functionality is necessary only when there are supposed to be more iterations of modification and review. If the aim is to simply show the design ideas once and collect feedback, developers often opt for the use of their blog. This allows them to reach a wider audience than with a wiki page while presenting the content as their own work. In other words, blogs usually represent the efforts of individual developers.

Various content management systems (such as Wordpress) are used for writing and publishing content of blogs. While these systems support the use from multiple users, they omit the functionality of a systematic tracking of revision and thus are scarcely used for collaborative writing. Blogs simply have a different purpose. During my time in the field the purpose of blogging was explicitly negotiated when a newcomer that wanted to apply for the

Google Summer of Code[114] stipend to work on Pitivi brought that issue up:

> Ben: It is necessary to post on a blog about the progress of the proposal for the GSoC? right?
> Steve: Ben, I don't mind if you don't
> Steve: And I think others agree
> Steve: What we want is progress, not blogging about lack of progress :)
> Ben: ;)
> Ben: that's better
> Steve: (not saying you would not progress, just that I preferred working to blogging for my GSoC and I don't think it ever hurt anyone :)
> Eric: we're programmers, not writers haha
> Steve: this :)
> …
> Ted: Ben, though I would like to strongly encourage you to blog
> …
> Ted: we can't force you though
> …
> Ted: I mean even one paragraph or two per week or two weeks, just to keep the pitivi and gnome communities informed, and to get feedback etc.
> Ted: no need for a whole book
> Steve: yeah Ben the thing is you would get a GSoC through GNOME, and they theoretically require your blogging
> Ted: but then, I say that as the person who is pretty much the only one in the entire pitivi team to blog
> Ted: (excluding the new fundraiser blog posts)
> Steve: blogging is good for two purposes IMO : technical stuff (always happy to find a blog post about specific issues I'm also facing)
> Steve: and marketing
> … Ben: more "marketing" I think
> Steve: what do you mean ?
> Ben: because I don't know if I'm wrong, but I really think that there are many USERS who want tutorials, know about new features… I think blogging is more for marketing in the case of Pitivi.
> Steve: Ben, I don't agree
> Ted: being able to explain a technical implementation or problem is a valuable skill
> Ted: blogging is a way to demonstrate that skill
> Ted: and this helps your career

114 Google Summer of Code is a stipend program that annually supports students to work on open source software. The general idea of the stipend program is to teach students practical skills through mentoring, while the students work to contribute to the public realm. But it has more far reaching consequences than this simple exchange. Many students continue to contribute to the projects even after the stipend is over. In the case of Pitivi, at least two of the maintainers got initially involved as participants in the stipend program. Therefore, by attracting students and developing successful strategies when asking for the stipend, some FOSS projects are able to gain new developers periodically.

Ted: (I’m just saying :P)
Steve: Well it also helps other hackers, which is a key argument too
Ted: yes
Ted: and also, I’m sure there are semi-savy fans/users out there who love to read about progress on projects like pitivi, but don’t have time (or skills) to sit around in our IRC channel and read everything that’s going on
Ted: you gotta admit it’s sometimes pretty crazy technical in here
Steve: Well I do hope so :)
Ben: I think marketing is powerful. That gives you users ;)
Steve: Ben, I said both were

From the first part of the conversation, it is clear what is the priority – it is more important to work than to blog about it. This position is summed up in the expression “we’re programmers, not writers”. However, as Ted (who declares himself to be the only one within the Pitivi team to blog consistently) joins the conversation, it shifts toward identifying the purposes of blogging. First, it is the “technical stuff” which may help others in finding solutions to problems they are facing – “it helps other hackers” (even Steve, who was initially skeptical admits he is “always happy” to find such blog posts). The value of this type of blog posts to others facing the same problem is that it spares them the effort put into investigating the problem and creating their own solution (while also demonstrating the technical skills of the post’s author).

Thanks to this practice one can safely assume that most problems were faced already by someone else and therefore, that someone blogged about it. Solving the problem then amounts to finding the appropriate blog post and implementing the solution described. In this way, blog posts create a reservoir of informative instruction materials which were written by knowledgeable authors and which can be picked up and utilized by others. Obviously, solving a problem and writing a blog post about it requires higher level of technical knowledge than finding the post and applying the prepared solution. In this way, actors can perform actions that are beyond their technical knowledge (for example, by copy-pasting commands from a blog to terminal without knowing precisely what will the commands do). This means that blog posts do not always spread knowledge. Knowledge would be transferred if the recipient learned to understand the problem and its solution in such a way that would make it possible to re-apply it in different circumstances. Blog posts are often not detailed enough to allow this, but even if they are not, they may serve as initial impulses to investigate an issue further.

The second purpose, “marketing”, is associated with blog posts that describe the progress of the project. New versions are announced, new features are demonstrated and work in progress is evaluated. Blogging about work in progress can be seen as one stage of the "release early, release often" imperative. It shows others what are the aims and what can be expected in the future and it is also a form of collecting feedback on the work even when the source code is not yet released. This is aimed primarily at users (or developers in other projects) that do not tune in into the developer's communication channels to experience what is going on firsthand, but wait for what the "blogosphere" brings them.

The blogosphere, in this case, is embodied in a blog aggregator called Planet Pitivi. It is the one place, that displays blog posts from Pitivi developers and informs anyone interested about what is new and what is going on in the project. There is also an aggregator called Planet GNOME which displays blog posts from the wider GNOME community and to reach a broader audience some of the Pitivi developers feed their blogs also to this aggregator. The "aggregation" is an arrangement in which texts are automatically redistributed from one place (the author's blog) to another (the aggregator). This arrangement is based on a mutual agreement between the administrators of a planet and the author of a blog and on the fact that the author meets certain requirements. Pitivi is a small project and acceptance to the aggregator is based on individual agreement. However, GNOME, being a much bigger community, has the requirements spelled out explicitly:

> We want readers of Planet GNOME to read and care about most of your posts.
>
> Some posts should be relevant to the GNOME community, either because they're related to GNOME, some underlying projects (like freedesktop.org projects), some technologies using GNOME, etc. or because it's a topic most people in our community care about, like freedom.[115]

The requirements are primarily related to the relevance of the blog posts to the audience. Blogging about projects related to GNOME (be it upstream or downstream) or about GNOME directly obviously meets the requirements regardless of the posts being more about technical issues or "marketing". What is new here are the topics "most people in our community care about" out of which only one example is given – "freedom". This common denominator has a historical background. It is part of the common knowledge within the community that the GNOME desktop environment was founded at least in part because KDE (at the time established free and open source desktop environment) relied on the Qt toolkit, which, at that time, had a proprietary license. Therefore, GNOME filled the need to have

115 *Planet GNOME Guidelines*. Last edit: 2014-04-20. Accessed: 2014-12-04. Available at: https://wiki.gnome.org/PlanetGnome.

desktop environment as independent of a proprietary source code as possible by relying on the GTK+ toolkit which uses licenses made by the Free Software Foundation from the very beginning. Therefore, it is no surprise to find such emphasis on the value of freedom permeating the documents of the GNOME Foundation and being used for moderation of its blog aggregator. That GNOME operates with a certain vision can be seen, apart from licensing choices, in its Code of Conduct:

> GNOME creates software for a better world. We achieve this by behaving well towards each other.
>
> Therefore this document suggests what we consider ideal behavior, so you know what to expect when getting involved in GNOME. This is who we are and what we want to be. There is no official enforcement of these principles, and this should not be interpreted like a legal document.[116]

However, it is indicative, that the "better world" which GNOME strives for is nowhere defined. There are several "advices" for individual behavior in the Code of Conduct (be respectful and considerate; be patient and generous; assume people mean well; try to be concise) but no image of what the world should look like in any of the documents. The utopia is not explicitly elaborated upon. It is left to the individual contributors to fill the words with meaning.

However, the environment that GNOME constitutes does not allow for just any interpretation of the words "better world". What GNOME does is that it provides infrastructure (legal and technical) for all the smaller projects it overarches. This infrastructure is specifically suited for the free and open source model of software development. Thus, the combination of this type of infrastructure with the words "software for a better world" hints at the world view according to which the better world is not only achieved, but also constituted by free and open source software. This meaning is implicitly present because of the infrastructure and there is no need to explicitly specify it.

Apart from insisting on one development model (which gains a moral valence in this context), there is a remarkably wide maneuvering space for various political positions. This "political agnosticism" is rooted in the classical free software license – the GNU General Public License (GPL). Of the three basic rights – use, modification and redistribution – this license focuses solely on the third one, while leaving the first two completely unrestricted.

116 *GNOME Code of Conduct*. Last edit: 2013-12-04. Accessed: 2014-12-04. Available at: https://wiki.gnome.org/Foundation/CodeOfConduct.

Furthermore, redistribution is allowed by just meeting the requirements of applying the original license to the derivative work and marking clearly any modifications that have been made to the original work. Few more supplements can be made concerning the author's liability or identity, but all additional terms are considered to be "further restrictions" which are explicitly disregarded:

> If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. … You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.[117]

Licensing fees and patent claims are stated as the most immediate examples of further restrictions, but this aspect of the license also covers for example the use for commercial purposes. If the author decides to license her work under the GPL, she may not further restrict the conditions for its use or redistribution. The software may be used by big corporations just as well as the unemployed or it may be used by activists just as well as the undemocratic regime they are fighting against. In this way, the license is "agnostic" – it explicitly denies the possibility for introducing any further restrictions than those which are stated in it. As a Linux kernel maintainer, Linus Torvalds elaborated upon this while he was discussing the differences between version 2 and version 3 of the GPL:

> For example, the GPLv2 in no way limits your use of the software. If you're a mad scientist, you can use GPLv2′d software for your evil plans to take over the world ("Sharks with lasers on their heads!!"), and the GPLv2 just says that you have to give source code back. And that's OK by me. I like sharks with lasers. I just want the mad scientists of the world to pay me back in kind. I made source code available to them, they have to make their changes to it available to me. After that, they can fry me with their shark-mounted lasers all they want.[118]

Now it becomes clear why the expression "better world" from GNOME's Code of Conduct is nowhere specified. As the GNU licenses (GPL and LGPL) are used consistently across

117 *GNU General Public License*. Text of the third version of the license published on the website of the GNU project. Published 2007-06-29. Accessed: 2014-11-16. Available at: http://www.gnu.org/copyleft/gpl.html.

118 *Linux Licensing*. Interview with Linus Torvalds published by Forbes.com. Published: 2006-05-03. Accessed: 2015-04-09. Available at: http://webcache.googleusercontent.com/search?q=cache:RaeC8J8_0isJ:www.forbes.com/2006/03/09/torvalds-linux-licensing-cz_dl_0309torvalds1.html+&cd=1&hl=cs&ct=clnk&gl=cz&client=ubuntu.

projects that GNOME associates, it is not surprising that it would apply similar sort of agnosticism in its documents. Given the infrastructure GNOME provides, contributors are free to interpret what better world is and translate this meaning into their willingness to spend hours of volunteer work, or invest in the development as a company. Thus, we can observe a mix of specificity (to a point of implementation) with regard to infrastructure aimed at particular software development model and ambiguity concerning further values and motivations. As we will see further, the same principle holds for most parts of the infrastructure considered individually. In the subsequent sections, I will elaborate on elements typically used in two of the most commonplace activities in software development: debugging and revision tracking.

4.2. Debugging

Bugzilla is a database of issues (bugs or feature requests) that were reported for a given program. Reports can be made by anyone who is willing to make an account in Bugzilla. This mostly involves dedicated users, testers of the given program or its developers. Because reports often come from actors that are unfamiliar with the project development or they focus on other areas inside the project, the database has to be regularly cultivated and organized. The relevant practice is called bug triaging and it involves setting bug severity, checking if the bug is really related to the given program, checking for bug duplicates, or checking if the provided information is correct and sufficient. Once organized and prioritized by bug triagers, the database essentially serves as a large scale to-do list for all contributors.

For newcomers, filing a bug is a rather lengthy process that requires submission of many information that end users are normally not even aware of, creating a barrier to feedback. However, without background information on the version of the program that exhibits the bug, on the environment the program runs in, or without a good description of the bug itself, the bug report is essentially useless. Filling in the background information on program version and environment gets rather straightforward after reporting one or two bugs. The ability to generate a good description of a bug takes a longer time. This is so because when dealing with non-trivial bugs, it is often not clear what triggers them and which component is its source. Therefore, prior to filing a bug report, a good deal of effort needs to go into a practice called debugging.

Debugging is an investigative activity that gets harder with increasing code complexity and with growing number of libraries the program is dependent on because an error can be hiding in one of the dependencies, not the program itself. At the time of writing, Pitivi itself has around 20 000 lines of code (not counting blank lines and comments), but GStreamer, its main dependency, has almost 1.5 million lines of code. Therefore, identifying the source of an error is no straightforward procedure. In a discussion, one of the project members expressed it in the following way: “the trick is always to find a way to simplify the cause of the bug and steps to reproduce to the maximum, it’s somewhat of an art ;)”.

By using the word “art” the speaker points to the fact that there is no precise set of rules that, if followed, would guarantee successful debugging. Rather, it is a process that relies heavily on the experience and knowledge of the person doing the debugging. However, the knowledge and experience do not have to be individual. The IRC channel is often used to share the results of debugging efforts and to discuss what the possible culprit may be. This is possible due to debugging tools that are able to translate an error into a stream (often very long) of digital text expressing what is going on in the internals of the debugged program. In this way, the user description of errors is substituted by a text with common formal properties.

Furthermore, the simplification aims at identifying only those steps that are necessary to trigger the bug. These steps may involve performing specific operations with the program or handling a specific file. In the latter case, it is important that the file is attached to the bug report or shared in some other way with the maintainers. Once the simplification is made, one can often guess which component is responsible for the error. But to get more information on what is wrong inside the component a special program is needed.

The standard tool for debugging is called GNU Debugger (GDB) which is used to pinpoint the part of code responsible for an error. First, the debugger has to be pointed to the program or library that presumably causes the error. The ability to make an informed guess in this area assumes knowledge about the program’s architecture and its internal workings. When pointed to a running process, GDB functions like an observer trying to record everything that is going on with it:

> Ben: #~|@¿~~! Segmentation fault
> Steve: gdb is your friend :)
> Ben: Steve: how do I use gdb when I compile the source?
> Steve: Ben, not sure I understand your question
> Steve: How do you trigger the segfault?
> Ben: I don’t know how to debug something big as Gstreamer… I’ve
> only used gdb for some single files
> Steve: Ben, what do you do to create the segfault?

Ben: I run the command
Ben: ges-launch-1.0 “multifile:///home/nick4/Pictures/Trash/numbers/%d.png?start=100&end=230&framerate=1/1” 0 5
Steve: OK then run:
Steve: gdb –args sh ges-launch-1.0 “multifile:///home/nick4/Pictures/Trash/numbers/%d.png?start=100&end=230&framerate=1/1” 0 5
Steve: when the segfault kicks in, you will type bt
Steve: press Enter
Steve: and see the backtrace of the thread that segfaulted
Steve: Ben, ^
Ben: Steve: thanks
Steve: Do you have the backtrace?
Ben: yes
Steve: Cool
Steve: segfaults are usually pretty straightforward to fix, be happy it’s not a race condition / deadlock ;)

Using GDB in this way generates a text file that is called backtrace or stack trace. At minimum, the stack trace identifies all functions (and libraries they are located in) that were called up to a point when the bug occurs. At best, the stack trace lists exact line numbers inside concrete files, identifying precisely parts of source code that were running before the bug occurred. To render a detailed stack trace, a special debugging version (these versions are created by a different compilation process, thus pointing to the relationship between compilation/black-boxing, and debugging – its reversal) of the tested program usually needs to be installed, one that allows for inspection of the running code. In this way the black-boxing done by compilation can be temporarily reversed and the internals of a running thing exposed. Once an error is debugged in this way, a valuable bug report can be filled. But to fully grasp the role debuggers play, consider the forms a program has before and after compilation:

```
def shutdown(self):
    if Pitivi.shutdown(self):
        self.gui.destroy()
        self.mainloop.quit()
        return True
    return False
```

The above is a part of Pitivi source code written in the Python programming language; the snippet represents a definition of the shutdown procedure. What follows is what one sees when the compiled Pitivi package is opened with a text editor:

```
^?
ELF^B^A^A^@^@^@^@^@^@^@^@^@^B^@>^@^A^@^@^@ø(@^@^@^@^@^@@^@^@^@^@^@^@^@^Hw^@
^@^@^@^@^@^@^@^@^@@^@8^@^H^@@^@^]^@^\^@^F^@^@^@^E^@^@^@@^@^@^@^@^@^@^@@^@@^
@^@^@^@^@@^@@^@^@^@^@^@À^A^@^@^@^@^@^@À^A^@^@^@^@^@^@^H^@^@^@^@^@^@^@^C^@^@
^@^D^@^@^@^@^B^@^@^@^@^@^@^@^B@^@^@^@^@^@^@^B@^@^@^@^@^@^\^@^@^@^@^@^@^@^\^
@^@^@^@^@^@^@^A^@^@^@^@^@^@^@^A^@^@^@^E^@^@^@^@^@^@^@^@^@^@^@^@^@@^@^@^@^@^
@^@^@@^@^@^@^@^@ôl^@^@^@^@^@^@ôl^@^@^@^@^@^@^@^@
^@^@^@^@^@^A^@^@^@^F^@^@^@^@p^@^@^@^@^@^@^@p
```

This is, however, not what a computer operates with. This is a result of a text editor taking a binary code and translating it into signs. These seem to be random because it is no longer binary code of a text, but of a program. To see what the computer operates with, one must access the contents of its memory:

```
009c000 0066 0138 0000 0000 3801 0800 0000 0000
009c010 0008 0b72 1003 001e 0200 0000 0001 0100
009c020 0001 5850 0124 41ed 0000 0000 ed41 0001
009c030 0000 0000 0100 0000 0000 0000 0000 03e8
009c040 0000 0000 e803 4654 011a 720e 030b 1810
009c050 0033 0b72 1003 1518 7200 030b 1a10 0004
009c060 4d4e 0105 0002 0066 011e 0000 0000 1e01
009c070 0800 0000 0000 0008 0b72 1003 001e 0200
009c080 0000 0001 0100 0101 5850 0124 41ed 0000
009c090 0000 ed41 0001 0000 0000 0100 0000 0000
009c0a0 0000 0000 03e8 0000 0000 e803 4654 011a
009c0b0 720e 030b 1810 0015 0b72 1003 0c18 7200
009c0c0 030b 1a10 0004 4d4e 0105 0004 0088 0d47
009c0d0 0000 0000 470d 0b64 0000 0000 640b 0b72
009c0e0 1003 001e 0000 0000 0001 0100 4713 4f4e
```

The memory contents take the form of a structured set of hexadecimal numbers.[119] The first column denotes a memory address, while the rest of the numbers in a row are representations of binary information. Each pair of hexadecimal numbers represents a binary byte.[120] This is the closest we get to see ones and zeros, the mythical building blocks of the digital. We can see that at this point (when inspecting the contents of a compiled program in a binary form), the logic according to which signs are organized is closer to the performance of voltage differences that hardware operates with than a language intelligible for humans. Hence the role of debuggers which make it possible to inspect the contents of running binaries in a more intelligible form.

The central part of every bug report is a description of the issue. In the description, three things should be articulated: expected behavior of the program, its actual behavior and steps

119 While the traditional decimal numeral system operates with a basic set of symbols `0 1 2 3 4 5 6 7 8 9`, the basic set of hexadecimal symbols is `0 1 2 3 4 5 6 7 8 9 a b c d e f` so instead of orders of ten, it operates with orders of sixteen.

120 This can be so because the number of states two combined hexadecimal digits can acquire (16^2=256) is equivalent to the number of states eight combined binary digits (one byte) can acquire (2^8=256). For example, 4d in hexadecimal (which is 77 in decimal) means 01001101 in binary.

to reproduce the bug. It is essential for others to be able to reproduce the bug for two reasons. First, newly added bug reports are automatically considered unconfirmed. A bug report has to be reproduced by at least one more contributor in order to be confirmed. Only then it is considered for further investigation and fixing. Second, in order to fix the bug, other contributors usually need to reproduce the bug in order to gain additional information and insight into the issue:

> Eric: Why do I get this? [link to an error message]
> Roy: Eric, That looks pretty wrong, how did that happen?
> Eric: I click a clip with two Box Filter effects
> Roy: Eric, Can you share the project so we can debug it?
> Roy: (and possibly open a bug report)
> Eric: which MTS did I send you last time?
> Eric: found the video, but sorry, cannot reproduce with a new project
> Roy: Eric, So you can't reproduce at all?
> Eric: nope, I re-added the filters and it works fine now
> Roy: Erg, that sounds like a bug in the effect priority management but I would need a way to reproduce to fix it

Bug reports provide public space for discussion of the problem, evaluation of alternative solutions, or assignment of severity and responsibility. Discussions often focus on identifying the problematic component, evaluation often takes into consideration how are similar problems solved in other programs. Responsibility is divided among the maintainers depending on their specialization within the project. Severity represents a continuum with blocker bugs on one side and enhancements on the other. Blocker bugs represent highest severity issues that need to be fixed before the next version is released. They are mostly regressions since previous versions or bugs that prevent testing of other issues. However, the decision on classifying a bug as blocker is never final. The bug can be reclassified to non-blocker or the version that it blocks can be heightened so that it does not stand in the way of releasing the next version.

This pattern can be illustrated by bug 570118 which was filed in February 2009 and was classified as blocker after a small discussion. Before the release of Pitivi version 0.13.1 in May 2009, its severity was demoted to normal so that the release would not be blocked by something that "would be a nice addition". After the May release, the bug severity was promoted to blocker and again demoted to normal before the release of version 0.13.3 in September 2009. Eventually, the bug was labeled as enhancement and after more than three years of no activity (except for minor adjustments made by a bug triager) in the bug report, it was resolved as “won’t fix” with the justification that the solution would “needlessly complicate things, and nobody else actually requested this feature”.

This fate is shared by many low severity bugs that are largely ignored by the core developers. They expect either the reporter (“scratching his/her own itch”) or some other occasional contributor to submit a patch. As the lack of manpower seems to be a constant condition, core developers rarely find the time to pursue low severity bugs. However, they have the power to demote the bugs that they see as low priority and that would stand in the way of a next release. In this way, bug severity can be subject to tug of war among core developers and bug reporters or other interested contributors. This was the case with bug 570118 which was promoted twice by its bug reporter (who also happened to be a bug triager) and repeatedly demoted by a core developer. Eventually, the reporter agreed and labeled the bug report as an enhancement request and after some time closed it.

In this context, submitting a bug report for a feature that is considered low severity from the start is considered futile effort. As one of the maintainers put it in a discussion: “the problem is we don’t want to add more surface for bugs with new features unless we have a very good reason”.

Filing a bug report represents an impulse for maintainers to react and it simultaneously creates public space (a “surface”) for anyone else to weigh in. In contrast to IRC, bug reports represent asynchronous form of communication that persists. This means that bug reports can mobilize broader audience than local and temporal chat discussions:

> Brian: I’m working on sth that is not a bug, and it’s not in Bugzilla as well. Should I create a new bug?
> Ted: Brian, it would be nice yeah
> Ted: it gives a public way to develop the idea
> Ted: and something to refer to
> Brian: Ok, I’ll do it. Thanks again, Ted!

By providing a “public way to develop the idea”, bug reports constitute a space for prototyping. This is more apparent in reports that are essentially feature requests. Within them, comments often involve descriptions of how a given feature is handled in various other programs and argumentation on which option would be best to pursue. The interesting thing is that general agreement is seldom reached and official decision seldom made. After some discussion, the prototyping process just moves to a new stage in which a self-assigned contributor attempts to implement the feature by creating a new branch in the source code history. The ultimate design decisions then lie in the hands of the contributor. It is the contributor who spends time and effort on the problem and this is compensated by the power to decide. However, this power is balanced by the existence of the review process. The contributor either has to align his work with the theory which the maintainers hold, or make an argument convincing enough to get them to go out of their way. In either case, the contributor is bound in his design decisions by negotiations with the project's maintainers.

Bug reports can also serve a purpose even when they do not result in a patch. They represent persistent traces that can be easily referenced (every bug report and every comment has its unique HTML address) and that inform anyone who was linked to them that the issue is known, that it is (or it is not) worked on and shows the progress that has been made on the issue. By this, repeated inquiries about the problem that keep the maintainers from productive work and makes them explain the problem multiple times are avoided. Also, by showing if anyone is working on a particular issue, bug reports help in avoiding parallel effort that may result in sensitive situations where there are two fixes for one issue and the maintainers have to pick one over the other, preferably without offending either.

4.3. Revision Tracking

Git is a source code management system used to track revisions of source code. This means that it is able to track changes in a given text file and create diffs – detailed representations of changes comparing two versions of a text file. As such, Git and other source code management systems are able to track only plain text files. It is the standard form in which text is stored in software development.

Git differs from older version control systems, in that it is distributed. This means that there is no central repository that one would need to access in order to make changes to the source code. Anyone can clone the source code (make a self-sufficient copy of everything that constitutes the developed program) and make their own repository containing their changes. This facilitates prototyping of modifications to the program. However, there is one main repository which contains the master branch that consists of reviewed commits and represents the official state of the source code. Write access to the main repository is restricted to the project maintainers and any contribution must be reviewed by one of them before pushing it to the main repository. Therefore, write access represents[121] the main hierarchical distinction

121 There are also other indicators of the maintainer position like administrative access to the project's web page, having an account on the project's Wiki or having maintainer rights in Bugzilla. All have in common that they provide administrative access to a part of the project's infrastructure. However, commit access to

between maintainers and the rest of the developers.

However, the effects of this hierarchical break are mitigated by the distributed nature of source code management that Git provides. Anyone with the abilities to do so can clone the main repository into their own personal repository and start modifying it without asking maintainers for permission. One can keep piling up commits in a personal repository as long as desired, and once satisfied with the outcome, the contributor can demonstrate that the modified version performs better in a certain respect. This provides an incentive for maintainers to review and appropriate the commits into the main repository. Thus, the distributed code management system is labeled as “truly open” and “meritocratic” in the Pitivi wiki. The wiki page also links to a video of Linus Torvalds (the originator of Git) describing the advantages of distributed source code management:

> Because you have a central repository means that everybody that works on the project needs to write to the central repository. Which means that since you don’t want everybody to write into the central repository, because most people are morons, you create this class of people who are ostensibly not morons. (…) So this whole commit access issue (…) is a huge psychological barrier and causes endless hours of politics in most open source projects. If you have a distributed model, it goes away. Everybody has commit access; you can do whatever you want to your project. You just get your own branch, you do great work, you do stupid work, nobody cares. It’s your copy, it’s your branch. And later on, if it turns out you did a great job, you can tell people: hey, here’s my branch and by the way, it performs ten times faster than anybody else’s branch so, how about pulling from me. And people do, and that’s actually how it works and we never have any politics. That’s not quite true, but we have other politics and we don’t have to worry about the commit access thing. And I think this is a huge issue and that alone should mean that every single open source system should never use anything but a distributed model, you get rid of a lot of issues.[122]

In his talk, Torvalds makes a hyperbole: everybody has commit access. This is true with regard to the cloned personal repositories, but not for the official main repository of a project that is still managed only by maintainers. By using Git the hierarchy is maintained but contributors gain a better position to demonstrate, argue and persuade maintainers to include their commits. This is so because the individual repositories represent a means of publishing[123]

---

the main repository is the key distinctive point as it gives access in an area central to software development: source code management.

122 *Tech Talk: Linus Torvalds on Git*. Video of a talk published by Google on Youtube.com. Published: 2007-05-14. Accessed: 2014-10-06. Available at: http://www.youtube.com/watch?v=4XpnKHJAok8#t=18m05s.

123 But it does not mean that anything the contributors do is necessarily public. All work is initially taking place on a private local machine and only afterwards is pushed into a server-hosted public repository. This allows contributors to control what stays private while making publication as easy as typing one command into a terminal.

work on an individual behalf. The modified source code is publicly available through a personal repository. Thus, when arguing about its quality, the sides are able to point directly to particular expressions on particular lines in particular files. This makes possible discussing the modifications with precise references which could be described as “talking the code” instead of “talking about the code”. Moreover, the modified cloned source code could be compiled to an independent version of the program in order to be tested. In these conditions, there is less space for what Torvalds calls “politics” that is, an unproductive challenging of power relations generating large communication load.

During development, personal repositories serve as prototyping spaces. Suppose we have a contributor whose name is Paul, he writes some new code, pushes it to his individual repository, gets feedback during review and amends his code accordingly. Then, the code is appropriated to a personal repository of the reviewer, where further revisions can be applied. At this stage, every revision the reviewer makes is discussed with Paul. Finally, after reaching a satisfactory state, the code is pushed to the main repository. After that, Paul resets his development branch and synchronizes it with the master branch of the main repository. This creates a shared reference point from which further contributions can be made. Thus, development using a distributed source code management system is an iterative process balancing divergence (branching out with new modifications) and integration (review and appropriation into the main repository).

Using Git involves first and foremost dividing work into units called “commits” (for illustration, see Appendix 7). These units denote logical wholes so that when it is necessary to revert a certain modification, a corresponding commit can be easily identified and edited.[124] In other words, commits should be conceived in such a way that the Git history is “atomic”.[125] The effort put into structuring Git history pays off in that the history is fully reversible. Furthermore, it provides information on authorship, time stamps and a detailed comparison of files before and after every modification which is essential for review. Still further, commits

124 An example to explain what the expression “logical wholes” means in this context: suppose I use Git to keep a history of changes when writing this text and I make a commit that consists of adding two paragraphs to the theoretical section. However, to form a logical whole, the commit should not consist only of those two paragraphs, but also of adding any new references (to the appropriate section) they introduce. This way, if I later decide to remove the commit, no other editing is necessary.

125 This requires planning and discipline in work defining clearly what the current task is and what will constitute a commit because Git monitors every modification that is made to the project files. It is possible to separate changes into different commits when they are made to different files quite easily. But separating changes made to one file into different commits requires manual intervention. To deal with these constraints, it is common to use to-do lists which enable planning tasks in advance. One other way is to use the *stash* function which puts aside all current modifications and reverts the files into the state of the last commit.

can be clustered into branches that provide isolated space for safe experimentation.[126] Thus, Git can be a very useful tool when utilized properly. But as my field notes indicate, proper use may not be a trivial matter for a newcomer:

> Looked at my repository today and I realized that the Git history, if not performed properly, is useless. I think I will have to delete most of the commits and edit some through interactive rebase to get some sense out of it.
>
> Now that I know rebase and amend I feel more empowered because I can fix whatever mistake I make in the future. Until now, everything was stored in the history and it was beginning to look messy and unuseful. The history is not necessarily what exactly happened, it is revisable and it is revised to serve a purpose.

However, for a seasoned developer, the impression may be very different:

> Therefore, I'll say that Git is great because it provides version control in a very non-intrusive way, and because it provides version control very easily for individual projects, too. … You don't have to be connected to the Internet, you don't have to setup a server, you don't even need a separate directory. You don't need to tell the world in advance what you're doing.
>
> "git init" or "git checkout -b" are enough to start a project or a feature, and enjoy version control from the very beginning. I think that this leads to code that is better and more maintainable.[127]

For this developer, the tool is "non-intrusive" by lowering the requirements for establishing version control for individual projects. All that is needed is to have Git installed and execute one command. Such lowering of requirements may lead to abandoning the practice of starting version control only when a project is sufficiently large or is being published. Instead, Git encourages the application of version control from the very beginning of the development process. As a result the planning and disciplination associated with committing modifications is present also from the beginning, avoiding the typically very large initial commit which aggregates (and thus obfuscates) all the changes made before version control was applied. Therefore, applying version control from the start results in more maintainable code in the sense that the commit discipline is enforced at all stages of development and can be observed retrospectively.

126 I discovered the full power of branching once I realized that switching from one branch to another means that the working files change literally under my own hands as different commits get applied. This feature allows for having available several versions of a file in a repository without the need to have several distinct copies, all the while detailed line-to-line comparisons of the branched versions can be summoned at any time.

127 *Git Success Stories and Tips from KVM Maintainer Paolo Bonzini*. Interview published by Linux.com. Published: 2015-04-07. Accessed: 2015-04-08. Available at: http://www.linux.com/news/featured-blogs/200-libby-clark/821899-git-success-stories-and-tips-from-kvm-maintainer-paolo-bonzini.

As I point out in my field notes, Git functions more like a tool for work coordination than like an archive that records what exactly happened. However, some of its features can be considered to perform archiving functions. For example, if someone gets interested in a certain part of code, Git can (through its *blame* command) provide information on who was the last one to edit that part. Furthermore, commits can be browsed as they were made to a branch, they can also be filtered by author or searched for a specific expression contained in the log messages attached to them. Every commit also has a unique identifier (called SHA) for referencing.

Now to the features that serve the purpose of work coordination. If two developers, working in parallel, edit the same line of code, Git will generate a conflict and guide one of them in its resolution. This means that after announcing a conflict, Git will open a text editor showing the modifications. The lines that were changed by both developers appear in three versions – the original one, the one modified by developer A and the one modified by developer B. It is then up to the human operator to pick appropriate parts from the three versions and merge them into the correct result, a fourth version that is saved as the conflict resolution.

If a developer needs to edit a commit which has been in the meantime covered by several other commits, Git can temporarily revert all the piled up commits to get to the desired one. Through this operation, Git essentially moves back in history to achieve a state when some commits are not yet applied. Then the developer is free to amend the current commit at will. When this is done, Git re-implements all reverted commits on top of the edited one as if it had been like that all along. However, the identifiers of commits that were either edited or reverted and re-implemented are changed because now they contain different code.

Such revisions of source code history are usually made only in the personal repositories of contributors, because a revision in the main repository would effectively and immediately change the reference point against which everyone makes their modifications, leading to many conflicts, and making subsequent integration of contributions problematic. Because of that, changes in the main repository are made only by submitting more commits. The personal repositories, on the other hand, constitute a safe space for experimentation, as anything can be reverted or modified. This is probably the most important implication of the distributed nature of Git: with individual repositories, developers get their own self-sufficient space to develop and refine their modifications, that is, to branch out of the official version in the main repository and still enjoy the benefits of version control. As one project maintainer remarks:

> Instead of having a single repository that everyone feeds from and into, everyone now has their own repository, their own branches. The meaning of branch changed. It's so cheap now.[128]

The ability to version control a patch created outside of the main repository was not something that other version control tools could provide at the time Git was created. This meant that large patches were difficult to review because they could not be dissected to smaller parts and the divergence between the patch and the development in the main repository that took place while the patch was being written was not systematically tracked, possibly leading to conflicts between the patch and other parallel modifications. By now, the distributed approach became standard as it brings significant refinement in creating patches. Patches are more refined because they consist of a number of smaller and well defined commits that simplify review and conflict resolution. Another project maintainer expressed his fondness of this approach:

> As maintainer I love that I can review changes as series of small commits instead of one big patch. I'm constantly asking developers to split their changes even more…[129]

It is clear that, through its command line interface, Git offers large functionality for source code management.[130] However, in order to operate correctly, Git needs to be supplemented with other programs such as diff tools (programs that generates detailed comparisons between files). It would also not be possible to resolve conflicts or revise history without pairing Git with a text editor. Furthermore, there are programs that serve as graphical front end for the command line tool that Git is, providing convenient user interface that is especially handy for a quick overview of a repository and its branches. By not attempting to include all the

128 *Git Success Stories and Tips from Ceph Creator Sage Weil* Interview published by Linux.com. Published: 2015-04-13. Accessed 2015-04-16. Available at: https://www.linux.com/news/featured-blogs/200-libby-clark/823164-git-success-stories-and-tips-from-ceph-creator-sage-weil.

129 *Git Success Stories and Tips from Wine Maintainer Alexandre Julliard*. Interview published by Linux.com. Published: 2015-04-10. Accessed: 2015-04-16. Available at: https://www.linux.com/news/featured-blogs/200-libby-clark/822789-git-success-stories-and-tips-from-wine-maintainer-alexandre-julliard.

130 To take advantage of it, one must undergo a demanding learning process. Luckily, Git is a widely used tool and so there are vast information resources available online. I quickly learned that in most cases, it is safe to assume that someone already faced similar problem before me and left an online trace containing the solution. Thus, apart from reading a coherent manual (/), my primary sources of information became blog posts of other software developers and question and answer sites like Stack Overflow (http://stackoverflow.com). The interface of such sites is adjusted for this particular purpose as it consists of conversation threads augmented with voting capabilities that make the most voted for answers most visible, providing fast reference for solving common problems. Checking Stack Overflow for problem solutions is such a common practice that for example a plugin was introduced to Atom, a text editor developed by Github, that displays the inquiries from Stack Overflow directly inside the editor.

functionality Git is in practice dependent on into Git itself, the developers of Git show their adherence to what is commonly referred to as “the Unix philosophy”. This approach to software design has been summed by Doug McIlroy (McIlroy in Raymond, 2003) in a following way:

> This is the Unix philosophy: Write programs that do one thing and do it well. Write programs to work together. Write programs to handle text streams, because that is a universal interface.

The approach encourages specialization to narrowly defined tasks that by themselves may seem trivial, but are general enough to be used in a wide variety of use cases. The practical utility is then based on the ability to combine the programs in a way that the output from one constitutes an input for another.[131] The universal medium which flows to and from programs is digital text. Searching for an expression in a log message, generating comparisons of files (diffs), resolving conflicts, and revising history, all those tasks are based on an interchange of digital text between Git and other programs. These are all built around the idea of automated manipulation of digital text.

Such text interchange is also taking place when publishing new commits in a personal repository. This is necessary in order for the commits to be reviewed and included into the official version of the developed program. The review process is (especially for a newcomer) an opportunity when knowledge is passed and norms are negotiated among maintainers and contributors. During the review of my first commits I learned many things, including how to make proper commits in the first place. But more importantly, I learned what the expected style of documentation writing is, pointing me in the direction I should proceed to in order to get my commits included. After a few iterations of writing new commits, receiving feedback and modifying them, I learned enough to make commits that got accepted without needing to be modified during review. It was the sandboxed space constituted by my personal repository that allowed me to publish, receive reviews and revise my work in order to develop the knowledge to contribute fluently to the project.

131 It seems that the tools generally abide to the ethical imperative of cybernetics, which was formulated by Heinz von Foerster in a following way: “act always so as to increase the number of choices” (Von Foerster, 2003, p. 227). In software development, this means that severing user options should be either backed by explicit reasoning (i.e. blocking unwanted actions) or due to missing feature that has not been implemented yet. As a result, providing a set of tools with general functionality and capable of almost arbitrary interconnection provides users with a vast array of possible courses of action. All these possibilities are accessed through the command line interface which has been a traditional interface for programs developed with the Unix philosophy in mind. However, in the present we see a rise of direct manipulation interfaces which define good design by leaving the smallest number of technical decisions to the user.

When the first batch of my commits was included into the main repository, I came to understand why contributions appear in clusters. When a contributor makes a pull request, that is, asks for his commits to be included into the main repository, one of maintainers reviews the commits, appropriates them and pushes them to the main repository. This can happen immediately or take several weeks, depending on how extensive the commits are and how busy the maintainers are. Sometimes, the order in which commits from various contributors will be reviewed and pulled has to be negotiated. This was the case when Eric made a pull request for his branch A:

> Eric: Steve, what’s the status with the branch B?
> Steve: Eric, I’ll need Keith to tell me what the “remaining bug” is
> Eric: I have the feeling the intention is to merge his branch first then mine
> Steve: No intentions here
> Steve: But I have the feeling reviewing his branch will be faster than reviewing yours :)
> Steve: I understand your concern, we should settle on a merge order
> Steve: If the remaining bug in Keith’s branch is benign / can be fixed easily, I think we’ll go the branch B first way
> Steve: if not then branch A
> Eric: I’m fine with everything, just want to make progress
> Steve: so you should wait and see what Keith is saying before rebasing
> Steve: Eric, I understand, but I’m pretty much the only one to review your branches so bear with me please :)
> Eric: mine is invasive, I don’t have any expectation
> Eric: I keep amending the main commit.. :)
> Steve: OK
> Steve: The beginning of the release cycle is the good time to do such things
> Steve: so I’ll make sure to have a look at both Keith’s and your branches soon

By the time Eric made his request, another contributor, Keith, finalized work on his branch B. Both branches had a common reference point in the main repository, but Eric’s branch modified the source code in many places and the changes it made were pervasive. Pulling Eric’s branch first would drastically change the source code in the main repository and that would result in generating many conflicts when pulling Keith’s branch. From the maintainer’s standpoint, it made sense to wait for solving the last issue on branch B and then pull it first. But there is one more angle to this situation: knowing that branch A will dramatically change the source code, Eric is essentially blocked from working on anything else while he is waiting for review. Working on something new would introduce a new branch B that would generate many conflicts if not merged prior to the invasive one. The only option he is left with is to pile his work onto branch A ("I keep amending the main commit"). By this, he risks only conflicts that would result from changes introduced by the review process.

Another strategy for avoiding conflicts when two developers work on the same part of code is to pass a modified branch from one of them to the other to let him implement his modifications on top of it before reviewing and merging the whole work into the main branch:

> Ted: isn't that stuff going to conflict with whatever changes Steve was doing to the behavior of the timeline last month?
> Roy: Ted, I will let him reimplement the right behaviour on that branch instead, right Steve ? xD

To this, Ted replies with a link to an image of a frightened cat that has a label: "You make kitty scared", indicating humorously the audacity of the procedure. But the procedure is illustrative of the degree to which Git makes it possible to make sets of modifications independent of their author and pass them around to others.

These examples of pull request coordination highlight the restrictions resulting from the use of source code management tools. In theory, these tools allow for almost any thinkable operation (from a purely technical standpoint, it does not matter which branch is pulled first, the final result will be the same). In practice, developers navigate by applying conventions to accomplishing the common tasks (in this case, opting for creating the smallest possible number of conflicts that require human assistance in their resolution). But these rules are always negotiated – if following the convention meant that a contributor would be blocked from working for a long time, it would not be followed. What is stable and cannot be negotiated, however, are the implications of using tools such as Git (if you have a shared reference point and a branch of pervasive modifications, you either merge everything else first, or you end up with many conflicts). In this sense, the infrastructure described above mediates the process of software development.

4.4. Rules of Mediation

Now is the time to explicitly formulate how the components described until now perform as mediators. The contents of this section (and its subsections) may seem repetitive at times, but this is only because I need to reiterate or elaborate upon some of the observations in order to relate them to the infra-language elaborated more thoroughly in chapter 2. In my analysis, I rely on the concept of technical mediation and its four meanings – composition, translation, black-boxing, and delegation – which are used to structure the this section.

4.4.1. Composition

The basic composition is expressed by the components already described in the previous sections. But to go one step beyond the most immediate tools, I will attempt to grasp the general implications of using the software that is usually presupposed in free and open source software development. Suppose I want to become a developer and buy a brand new computer with no software installed (except for firmware, of course).[132] What should I do?

The first step is to install an appropriate operating system. This represents a mobilization of a vast network of relations. Out of the three well known types of operating systems (Windows, Mac OS, Linux), Linux distributions are used in this area as FOSS developers usually prefer using free and open source software. Specifically, the Pitivi maintainers use a distribution called Fedora. Linux distributions generally consist of a Linux kernel, a set of packages from the GNU project, a package management system, some libraries, a display server, a desktop environment, and a selection of pre-installed applications.

The Linux kernel, although originally developed by Linus Torvalds as a hobby project, now consists of millions lines of code and its development is sponsored by multiple companies. The kernel is a basic building block of the operating system. Among other things, it is responsible for controlling hardware and so the correct functionality of hardware components often depends on which version of kernel is installed.

The GNU project, founded by Richard Stallman, is overarching a number of software development projects. It is sponsored by the Free Software Foundation and represents the cusp of the free software development efforts. The packages from the GNU project are typically present in Linux distributions to provide basic functionality. The set includes the following widely used packages:

Bash provides functionality needed for the command line interface. Without this functionality, the following programs (and many others) would lack an interface to communicate with their human operators. Bash defines a set of basic commands that are accepted and can be combined. Furthermore, it is used to run programs and control them. As such it is an essential piece of software for any kind of software development.

GCC is the GNU Compiler Collection, a program used to compile (human readable) source code into (machine readable) executable binary files. It is the tool that turns an editable

132 Naturally, the readily available computer presupposes a huge network of it's own (going as far as to mining raw minerals as for example Jussi Parikka (2014) shows), but since this work is focused on software, I will cut the network relations at the edge of the hardware/software distinction.

text into an interface that functions according to its own logic. It also allows for compiling versions that are specifically aimed at debugging.

GDB is the GNU Debugger, a program used to temporarily and partially revert the compilation process by explicating the parts of source code that run prior to an error. It allows the developers to see the internals of a running program, which is essential for finding errors (bugs) in the source code.

Text editors. There are multiple editors maintained within the GNU project. *Nano* is a simple text editor aimed at beginners or those that perform a more casual text editing. *Emacs* is a complex editor with so many features that it is often jokingly described as an operating system on its own. *Sed* is an editor that accepts commands and thus makes possible for text editing to be automated. All three editors are capable of processing regular expressions, which is essential for performing automated or more complex editing. Text editors are the tools that developers spend most time with as they are used to view and edit the source code.

Diffutils is a set of utilities aimed at making comparisons between text files. These tools are commonly used to produce and see differences (between an original file and a later version, or between two files edited in parallel by two developers). As such they are at the core of any source code management or version control system.

To be sure, the GNU project maintains many more packages[133] and there are also alternative packages outside of the GNU project that provide the described functionality. But from my experience in the field, these are the standard packages available for any Linux distribution.

Contrary to the standard components described above, the package management systems differ for various Linux distributions. This means that once a stable version of a program is released, it needs to be packaged independently for the different package managers and distributions. The packaging process gets complicated as the versions of the dependencies required by the program must be met by what is currently packaged for the given distribution. This creates an interdependence between software packages which in some cases can be hard to satisfy. Thus, when working on a program, the developers must draw on their knowledge of the ecosystem which boils down to striking a balance in the following area – will we use the functionality provided by the new versions of libraries even though we risk a slower adoption of our software because the new versions of libraries may not be available in the targeted

133 See the full list here: https://www.gnu.org/software/.

distributions when we release a stable version? The knowledge that allows for acting with a satisfying answer for this question comes from a long involvement in the broader ecosystem of a variety of Linux distributions.

The components described up to this point could form a development environment using only the command line interface. To provide the graphical user interface (if only for testing the developed program), a display server and a desktop environment must be present. Desktop environments typically represent broader forms of organization associating many smaller projects with the aim of providing all the functionality an end-user would expect. Thus, desktop environments span from libraries providing the basic blocks for building software to applications that are built on top of these libraries and that provide users with the interfaces and functionalities they use. More specifically, Pitivi is developed within the framework of the GNOME desktop environment and thus uses its libraries (Glib, GTK+, or Clutter). To take advantage of the functionality provided by those libraries, the developers need to know the (changing) programming interfaces that they offer. This requires further monitoring and involvement with the broader software ecosystem.

But knowing how to utilize an existing functionality is not enough to become a developer. One needs to learn the language in order to be able to write the source code in the first place! Programming languages are made of syntax conventions that translate into instructions performed by the computer. In theory, anyone can combine a set of conventions to create a new language. What counts, however, is whether the language is supported by a compiler and thus has the possibility to be turned into an executable program. Without a compiler, a programming language is just a set of conventions without a utility. Furthermore, without a text editor supporting the language's syntax, it is very hard to work with as syntax highlighting greatly increases the source code readability. Because the introduction of new languages has these restraints, it is no surprise that there is a limited number of standard, widely known languages. Two of which are used in the Pitivi stack: C and Python. While Pitivi itself (mainly its interface) is written in Python – a language that is easier to pick up, but which is ultimately not very efficient for high performance tasks – the libraries which "do the heavy lifting" of video processing are written in C – a more difficult language to learn, yet much more efficient. In such a case, knowledge of one of the languages essentially determines the areas in which a developer is able to contribute to the project.

Now if we add to these components the tools and platforms elaborated upon in the previous sections of this text (Git, Bugzilla, Wiki, blogs, IRC channels, licenses, and the development versions of the software itself), we are starting to grasp the extent of the composition that goes into a single project of free and open source software development. But we get a more complete picture when we realize that each of these components is a development project with a composition of its own, thus widening the number of relations in an exponential degree. When we put this together with the fact that development decisions require knowledge of a broader development ecosystem, we see what is called the barrier to contribution. Thus, for a newcomer, the world of *open* source software development is anything but *transparent*. This is not because the key information are secret, they are indeed publicly available. But there is just so much of it, that it requires significant effort to process them. This point has been expressed in a public video stream with developers from another project:

> Nobody is allowed to push code directly to the trunk just because we believe in code reviews so there's not really a barrier like hey I need to approved to such and such team before I can push code. That's not true.
>
> Once someone's been around for a little bit, then they start to find out where we keep certain information and things. I mean it's not like we intentionally keep people out, it's just like once you get into the flow of how things work then you figure it out and then one day you're like "oh, I'm an [project's name] developer" just because you know where the stuff's at.
>
> The only difference between us and a lot of other people is that we've been around longer and all that kind of stuff.

Here, the developers say that there is no higher authority that would select who can contribute. The selection happens on an entirely different front: the effort put into finding out where information are kept and learning how to use them. Obviously, the presupposition here is that the newcomers already have the skills to contribute. If this is true, then the difference between a newcomer and a recognized developer is knowing "where the stuff's at". It is a problem of orientation, not accessibility. In other words, the word "open" designating this model of software development certainly means "accessible" but it does not mean that modification and redistribution are effortless. Quite the contrary.

As I have shown, there are lots requirements new contributors have to stand up to. They need to know their tools, the standard platforms and also the project specifics. This goes directly against the advertised claims that “anyone”[134] can contribute. Certainly, there are roles in FOSS projects that are easier to pick up (like a translator or documentation writer), but these are rather supportive of the main activity and more often than not don’t allow for employing the main incentive for development: scratching one’s own itch. Therefore, there is a barrier to contribution that selects contributors according to their skills, motivation and free time. The other side of the coin is that overcoming the barrier means learning, which is empowering either on its own (the actors are able to modify software to fit their needs) or in other institutional contexts (the actors can demonstrate their skills in educational institutions or at the labor market).

But how can the developers presuppose that the newcomers already have the skills necessary to contribute when every development project is relying on mastering so many other entities? The first part of the answer could be expressed by a single word: specialization. Developers choose the components they want to work on and over time, they become specialists in those areas. This is nothing uncommon. The second part of the answer rests on the fact that the relations are not attached to ever more new entities. The meaning of standards is that they are present through the field. For example, Bugzilla uses Git and wiki in its own development while MediaWiki uses Git for managing its source code. And almost all of the projects have an IRC channel. To be sure, all the projects have different rules for using the components in development. For example, the commit messages will contain different information – in the MediaWiki project, the commit messages must contain the name of the reviewer, while in Pitivi they contain only a description of what the commit does. But the repetitive occurrence of these development components gives the landscape a sense of arrangement. Therefore, when a core developer of the Drupal project is asked what makes Git a great tool, she answers:

> For me, it’s Git’s ubiquity. Particularly in the last couple of years, Git has become the clear winner in the version control wars, and having one common language to speak with and collaborate with other developers has solved SO many problems.[135]

134 Equating public access to source code with the claim that anyone can modify it and get the modifications through the review process is common. It is also the implication of what the first page of the Pitivi website states in the biggest font: “We believe in allowing everyone on the planet to express themselves through filmmaking, with tools that they can own and improve.” There is a direct relationship between the words “everyone” and “improve” in that statement.

135 *Git Success Stories and Tips from Drupal Core Committer Angie Byron*. Interview published by Linux.com. Published: 2015-04-08. Accessed: 2015-04-10. Available at: http://www.linux.com/news/featured-blogs/200-libby-clark/822227-git-success-stories-and-tips-from-drupal-core-committer-angie-byron.

Given the same question, a maintainer of the Qt project remarks:

> And since it's now so popular, it's not a barrier of entry for new contributors.[136]

The advantage to using a standard tool is the lower entry barrier for newcomers. This is based on the assumption that most newcomers will already be familiar with the tool and do not have to learn it in the course of getting involved in the project. Furthermore, once the tool is used widely enough, one can assume that others know its commands. Therefore, it ceases to be necessary to describe at length various courses of action, it suffices to only name the commands (which, in this sense, constitute the "common language").

All these findings point to one claim – that the knowledge of standard tools reaches very far. This claim draws on the conceptualization of knowledge as a relationship between an actor and information (for example information that constitutes an interface). And in a situation where a certain piece of information is standard, i.e. has established presence in multiple locations, knowledge follows the actor wherever he goes (within the field). This relationship is symmetrical in the sense that both actor and information are needed to produce knowledge, but it is also asymmetrical in the sense that actors cannot be copied (and thus reach multiple locations simultaneously), only information can. Thus we are getting back to reproducibility and automated manipulation of digital text.

4.4.2. Translation and Delegation

This digital medium is crucial also for translation and delegation within free and open source software development. In this environment, translation occurs in a very literal sense when ordinary text is translated into one of the languages that are spoken by the major actants. For example, if a text is to be displayed on the project's wiki, it must contain the markup signs consistent with the wiki markup language. This means that the formatting of such text needs to be marked by additional characters. When saved in the wiki, these additional characters are translated into the desired formatting for the original text. As a result, there are two texts: first, the raw text with visible markup signs which is edited by the author; second, the result translated into a formatted text without the markup signs that is accessible to the reader. Unless the reader decides to inspect the page by looking into the "backstage" of the text (in MediaWiki, this is achieved by clicking the "View source" button which is present on every

136 *Git Success Stories and Tips from Qt Maintainer Thiago Macieira*. Interview published by Linux.com. Published: 2015-04-07. Accessed: 2015-04-10. Available at: http://www.linux.com/news/featured-blogs/200-libby-clark/821948-git-success-stories-and-tips-from-qt-maintainer-thiago-macieira.

page), she will not see the markup signs. They serve as instructions for the translating agency (in this case, a component inside the WikiMedia platform), but are invisible for the reader.

> `''text in italics''` gets translated as *text in italics*
> `'''bold text'''` gets translated as **bold text**

The same applies for example to the Hyper Text Markup Language (HTML), which is used for publishing on the project's web pages or developer blogs. The syntax of the HTML markup signs is more complex than that of the wiki markup because the language is aimed at a more general use. But it is still a markup language[137] – it is predominantly concerned with formatting of documents (websites) and their display. Adding to the complexity, there are number of translating agencies for HTML – commonly known as web browsers, with minor differences in how the markup is interpreted in each one of them. However, the full complexity is uncovered by the fact that more than one language may be involved in the translation. For example, because wiki pages are accessed by web browsers, it is necessary to translate the content into HTML. Therefore, the wiki markup is translated first into HTML markup and only then into the reader-ready formatted text.

> Wiki: `''text in italics''`
> HTML: `<i>text in italics</i>`
> formatted: *text in italics*

> Wiki: `'''bold text'''`
> HTML: `<b>bold text</b>`
> formatted: **bold text**

Therefore, we have a chain of translations at the beginning of which stands the author, who decides which markup language to use and writes the first version of the text in it – thus, augmenting plain natural language with markup. All subsequent translations are done by parsers which employ automated manipulation of text to translate it into different languages or reader-ready results. Each of these parsers consists of set of rules for substitution of markup signs of one language to functionally equivalent signs of another one (as can be seen above). This is only possible because of the existence of a set of conventions aimed at searching and replacing text patterns. These conventions may be part of a particular programming language, but in their raw form, they are formulated in what is called "regular expressions". Whatever their form, their general function is the same – instead of the limited

137 Or traditionally has been. The latest version called HTML5 represents a departure from this category, but this fact is not relevant for the purpose of this text.

options of searching and replacing literal strings of text, they introduce general concepts like a word, a number, a letter, beginning or end of line; thus allowing for formulation of patterns that match entire classes of literal strings. I will use few examples of regular expressions to demonstrate their functionality:

> `.*@example\.com` matches any email address at the example.com provider
> `[0-9][0-9]?\.[0-9][0-9]?\.[0-9][0-9]` matches any date in the dd–mm–yy, mm–dd–yy, d–m–yy, or m–d–yy formats
> `={2,6}.*={2,6}` matches any second to sixth level heading in the wiki markup language
> `<b>\w</b>` matches a word formatted to bold in the HTML markup language

The importance of regular expressions is also shown by the fact that they are regulated by the POSIX standard – a standard that regulates key components of operating systems to assure compatibility. However, despite standardization efforts, there are several versions, or “flavors” of regular expressions and so their exact formulation always depends on the parser that is used to process them. Now we arrive at the point that language translation is a nested problem. This is because parsers of markup languages require regular expressions (or programming languages) that require their own parsers (in the case of programming languages a compiler), which may comply to different standards. Therefore, as we can see, even (literal) translation has its own composition.

The results of the described chain of translations are usually used for publishing. The contents of both blogs and wiki pages can be served to multiple readers at the same time as long as they are present on a server, thus creating a (more or less) stable point for referencing. Essentially, the function of these pages is to delegate information to a number of places (monitors of the connected readers) for a given period of time. In this way the delegation overcomes distance (the text was written somewhere else) and time (the text was written at another time) and is able to transport the same information into entirely different contexts.

In programming languages (as opposed to markup languages), the translation gets a whole new dimension. It is no longer the case that only markup signs are added to natural language, the whole statement needs to be reformulated according to the logical structure of the programming language and of the interface that is provided by libraries. This is so because unlike markup languages that are focused on organizing text into the desired shape, programming languages are aimed at performing actions if certain conditions are met. The markup languages are full of markup signs while programming languages are full of conditionals. As an example, consider this part of a Bash script (which, technically, is not a programming language, but its use of conditionals is analogical) with every line commented in natural language:

```
while read line             # read every line of a given file
  do                        # at each line, do the following
    if [[ ${line} =~ $n ]]  # search for pattern specified in variable "n"
    then                    # if the pattern is found, do the following
       buff1="$buff1 $line" # add the line variable "buff1"
    else                    # if the pattern is not found, do the following
       buff1="$buff1\n\n"   # add two new lines to the variable "buff1"
    fi                      # no other options will follow
  done < $file              # do all of the above to a file specified in
                              the variable "file"
```

In the IRC channel, explanations of what parts of source code “do” are common. It allows for fast overview of the code and saves hours of browsing through it to discover its function. Provided there is someone knowledgeable enough to explain it.

> Ben: You have a media stream, imagine a video… you want to seek to the time 30s… how Gstreamer knows it has to stream the data at (the time) 30s and not 40s or 100s or whatever
> Ben: Steve: ^
> Steve: you’re asking me to explain you the seeking mechanism in gstreamer?
> Steve: Basically a seek event travels upstream until an element (such as yours, or a demuxer) answers “yep I’ll handle that”
> Steve: It then seeks itself in ways that are relevant to its job (for example your element finds the image that has to be output first, a demuxer might look up an index table to find out the byte offset in a file at which he’ll be able to resume streaming, preferably a keyframe)
> Steve: It then propagates a segment downstream, saying “the segment I’m going to play starts at the nth second in media time and ends at nth second in media time”
> Steve: And starts to output buffers once again
> Steve: In the case of a demuxer and accurate seeking, the demuxer might output data prior to the requested start, necessary for decoding of the first actual frame to be rendered, the decoder will clip these “decode-only” frames but that’s irrelevant for your element
> Steve: As it can be accurate at no cost even when the requested seek is not accurate
> Steve: That means a demuxer usually needs to convert a time seek to a byte range when operating in pull mode
> Steve: But imagesequence doesn’t need that
> Steve: Ben, does that answer your question?

From this part of conversation, we can see that the seeking mechanism is no simple matter. Luckily, we don’t have to go through all of its complexities as for the present purpose the excerpt serves to demonstrate how source code is translated back into the natural language.

There are events and elements that travel, answer, need to do things, say things and so on. The result of this complicated interconnection is that a video is played exactly from the point selected by a user – also an action. Here we can see that the source code, (after compilation of course) literally does things. It is designed to do so by its originators – its developers. They devise the actions to be taken and design the interlocking of elements and events. By writing, they populate the internals of a program with entities that delegate action and wait for the specified conditions to trigger it. The software entities that are closest to a literal expression of this principle are daemons.[138] These are programs on their own, designed to run in background, monitor other tasks, and trigger an action if certain conditions occur. Thus, by formulating precise rules for triggering conditions and actions to be carried out, these entities allow for a very sophisticated delegation of action.

The other side of the coin is that this sophistication leads to a high rate of errors, that is, delegated actions that are not intended by the developers. The presence of errors is seen as inevitable and is considered to be unavoidable when writing source code. One of the maintainers expressed this point aptly when he substituted writing code for creating new bugs by saying “We all love creating new bugs :)”. There is a distinction between the creative and fun activity of writing source code that gives a program new features and the often hard and frustrating maintenance that involves finding and fixing errors. Because the first activity usually means introducing new bugs to the existing code, it is common for development projects to have a “feature freeze” period before releasing a stable version. This period is dedicated to maintenance only and the design and implementation of new features is put on hold. During my fieldwork in the Pitivi project, the development efforts were stretched towards releasing the 1.0 stable version. As a result, there has been a feature freeze period for more than a year. This was indicated by a seldom use of collaborative drafting tools like wiki pages on the one hand, and common instances when new bugs and debugging information filled the IRC channel.

I have described the basic principles of debugging in one of the previous sections, but let us return to this activity and identify the translation and delegation that are performed in the course of it. Debugging starts with an error, an unexpected course of action performed by the program, which does not correspond to (or directly hinder) its functionality. Errors are

138 The term daemon is probably used with reference to the Maxwell’s daemon which represents a well known image of an entity quietly and tirelessly working in the background and performing predictable actions depending on input. This image is not limited to science and technology, for example Pierre Bourdieu uses the metaphor to describe the function of educational system (Bourdieu, 1998, p. 20).

usually discovered by using the program (regardless of whether use by a developer aimed specifically at testing or general use by an end-user). This means that their natural form is contained in the user interface of the program. The whole practice of debugging is then focused on identifying the component and, if possible, the part of source code within it that causes the error. The manifestation within an interface (which is in the beginning usually expressed in natural language) is translated into an identified part of source code. In this process, the debugging tool works like an interpreter that bridges the barrier made by compilation and translates the error from one form to another.

Usually, when an error is discovered, a bug report is created to account for it and to track its development. The report ideally contains both an expression of the error in natural language (that deals with the compiled interface) and the information produced by a debugging tool (that identifies parts of source code). These information are submitted to Bugzilla, which stores the data in a database and creates a dedicated web page for each bug. Thus the submitted information are embedded in HTML to be viewable in a web browser. The translation into the markup language goes hand in hand with publishing the information. During the process, the information are delegated from their origin at someone’s local computer into a database run on a server and then into different local computers that belong to the developers who access the report through web browsers.

If the bug is confirmed and no other information is necessary to identify its cause, the report serves as a space where public discussion takes place about the options of how to approach it, how to design a patch that would fix it. In this phase, the error is transformed into a task that can be picked up and solved by a developer. Now, it is halfway between problem and solution. Information about bugs are not posted only to the IRC channel (even though the discussion is faster and more convenient there), because fixing a bug usually takes a longer period of time (during which the debugging information must be handy) which can be overcome only with the asynchronous communication that Bugzilla provides. And by also keeping track of all errors that were fixed, Bugzilla delegates the information even further to the future when the fixed bug can manifest itself as a regression. The small memory footprint of digital text and the efficiency of automated search functions allow for this luxury of archiving.

When a developer picks a task from Bugzilla, or decides to write a brand new feature, he needs to translate the feature into a complete design artifact first. A common vehicle in this process are artifacts called mockups. In software development, mockups are essentially screenshots of how the future versions of a program will look. They take the form of pictures expressing the intended shape of the user interface and can be used to visualize adding buttons, resizing panes, moving menus, or simplifying the interface by removing any of the elements. Thus, the idea of a new design is translated from natural language to a non-text medium directly showing the result. After that, mockups are usually published on a blog or in the project wiki. Both of these publishing options serve to delegate the design idea to other developers (and possibly users), while collecting feedback. Thus, this is a type of delegation that is aimed at provoking action, not propagating it.

The next step in the design process is to create a functioning prototype by modifying the existing program's source code appropriately. However, to ensure that the modifications remain distinguishable and reversible, it must be tracked by a source code management system. Hence, the translation at this point is double: first, from a mockup and natural language to a programming language; second, from a continuous stream of developer's work to a set of self-sufficient units. The first translation, as I already showed, leads to the delegation of intended actions from a private computer the source code is modified on, to all the computers on which the code is running in the form of a compiled program. In the second type of translation, the work is planned in order to be dissected into a set of atomic commits. In Pitivi, the commits are labeled in an imperative form, expressing what is achieved by applying them. Here is a log with labels of some of my commits that were included into the main repository:

```
commit 5b06c4686ffbe0eb76d08fd1b6f3b618384f5057
Author: Tomas Karger <tomkarger@gmail.com>
Date:   Sun May 25 14:03:52 2014 +0200

help: replace menu bar and main toolbar with header bar and app menu
everywhere

commit b103ce8947026d03c1f507200c7479fea38d6d4c
Author: Tomas Karger <tomkarger@gmail.com>
Date:   Tue May 13 16:09:33 2014 +0200

help: update sysreq.page

commit 90ac17a661a9a29686cf4a4f6f926e6f75f417d9
Author: Tomas Karger <tomkarger@gmail.com>
Date:   Tue May 13 14:35:50 2014 +0200

help: remove unnecessary note from mainwindow.page
```

```
commit ee93d664894b5253941d58f834f00d3bdd8f87f7
Author: Tomas Karger <tomkarger@gmail.com>
Date:   Tue May 13 14:29:02 2014 +0200

help: add tip on detaching the previewer to mainwindow.page

commit 6497108acf97c8c772e5dfea3d2824c68d5fb4b8
Author: Tomas Karger <tomkarger@gmail.com>
Date:   Tue May 6 17:24:33 2014 +0200

help: adjust see also links
```

The labels are indicative of the delegation performed by commits. They too delegate actions, only not directly to user's computers, but to the source code (or, in the above case, to the user manual). In this sense, the delegation of source code management systems (such as Git) is more enclosed in the development process (the commits usually don't reach the users, they remain a development aid) while the delegation performed by the programming languages (or by the source code which is the concrete manifestation of them) goes practically from end to end (from developers to users).

By creating a database of labeled, time stamped, uniquely identifiable and precisely recorded commits, source code management tools translate the work of an individual, so that it is self-sufficient and can be shared with other programs and programmers. It uses the "universal interface" of digital text (formed into predefined data structures) in order to pass information for processing to other entities. Commits can be pushed to a repository and pulled from it by someone else, conflicting modifications are highlighted and resolved, different versions of a file are sent to difftools to visualize differences, commit history can be sent to a search tool to find work of a particular contributor. These are just common examples.

By translating work into standardized commits, Git performs an important function of delegation. Publishing commits in a repository makes possible independent cloning of the code by anyone interested. The source code is delegated to a public space so that it can be reviewed, modified and appropriated, it can become a subject of discussions, and it can trigger learning. This delegation is key for free and open source software development, because it allows for the existence of one its prime organizational features – work self-assignment.[139] This seems to be consistent with the fact that FOSS projects are often driven by volunteer effort, and so there is a lack of any leverage to enforce work assignment. This is often contrasted with the fact that FOSS projects sustain themselves for long periods of time and are

139 A study by Crowston et al. (2007, p. 6) indicates that self-assignment is the most frequent form of work assignment in free and open source software development.

able to produce stable and widely used software.

Seen from this perspective, a FOSS project represents a puzzling combination of stability and fluidity. However, such configuration can be explained when we take into consideration the usual process of how new contributors get involved in a project. In this process, the minority of maintainers serve as bearers of knowledge that is necessary for meaningful contribution. The heavy involvement of maintainers does not correspond to the mythical image[140] of aggregates of unrestrained contributors that swarm around projects to contribute when the conditions are right. To be sure, there are occasional contributors who get involved from time to time, but cannot be relied upon for consistent input. In an interview for the Linux Voice magazine, Lennart Poettering, a well-known developer of the controversial Systemd init system talks about such occasional contributions as “drive-by patches”:

> LP: So anyway, long story short, we came to the conclusion that Upstart is conceptually wrong, and it moved at glacial speeds. It also had the problem that Canonical tried very hard to stay in control of it. They made sure, with copyright assignment, that they made it really hard to contribute, but that’s what Linux actually lives off. You get these drive-by patches, as I would call them, where people see that something is broken, or something could be improved. They do a Git checkout, do one change, send you it and forget about it.
>
> LV: And you never see them again!
>
> LP: Yeah, and this is great – these are the people you want to have, because the vast majority of patches are actually of that kind. It gives you this polishing that you want. The people invested in the project all the time do the big things, and don’t care so much about the polishing. So these kind of patches are what you want. But if you do these copyright assignment things, you will never get those people because they would have to sign a contract before they can send you something.[141]

In the interview, Poettering compares Systemd with Upstart, a different init system developed by Canonical. He points out, that the additional negotiation (Yochai Benkler would see this as an increase in transaction costs) involved in signing a contract before contributing discourages potential contributors (which, in turn, may be one of the reasons why, according to him, its development moved at “glacial speeds”). A group that is most affected by the requirement of contract are the occasional contributors, whose work is highlighted by Poettering. He claims

140 For example, the Fedora contributor conference is called Flock, drawing on the images of behavior of bird collectives which is chaotic, but still organised and also on the popular proverb “birds of feather flock together”.

141 *Interview: Lennart Poettering*. An interview published by the LinuxVoice magazine. Published: 2015-01-16. Accessed: 2015-01-16. Available at: http://www.linuxvoice.com/interview-lennart-poettering/.

that Linux literally “lives off” the drive-by patches, that the “vast majority” of patches are of this kind and that the occasional contributors are “the people you want to have”. On the other hand, the maintainers (“the people invested in the project”) are mentioned only in one sentence and their importance is diminished even though Poettering acknowledges that they are doing “the big things”. This tendency is described more precisely by Paolo Bonzini, a maintainer of a different project, who, in an interview for Linux.com, talks about the “long tail” distribution:

> Each release of QEMU has contributions from roughly 170 people. The distribution has a very long tail: about 40 percent of those 170 people contribute only one patch, and about 60 percent contribute less than five.
>
> KVM is smaller, with about 25 people contributing to each release. The same “long tail” effect is visible there, about half of the people only contribute one or two patches.
> The long tail is very important. A lot of those “drive-by” patches are bug fixes.[142]

These remarks indicate that while the work of occasional contributors is emphasized, the work of maintainers is taken for granted and is, in a sense, invisible (as elaborated by Bonnie Nardi and Yrjö Engeström (1999)). The work on the “big things” carried out by few dedicated individuals[143] goes largely unnoticed. However, the core developers, by being available for communication most of the time (and keeping their involvement for extended periods of time), holding necessary knowledge (with the willingness to share it) and performing the (not so fun) maintenance, form the backbone of a project without which the drive-by patches would not be reviewed and merged into the main branch.

This directly relates to Peter Naur’s claim about the importance of knowledge transmission in software development. He states that if a software project is abandoned by its original development team, it can be proclaimed dead as no new team of developers will be able to revive it. This is so because the new team would lack the knowledge developed by the original one. Thus, a discontinuity in knowledge transmission may be fatal for a software development project. However, Naur’s claim describes a corner case and therefore, I do not aim to judge its validity. With it, I merely aim to show an important feature of software development. That is, because the knowledge established to pursue development is more extensive than what is recorded as information, all those non-humans that serve as delegators

142 *Git Success Stories and Tips from KVM Maintainer Paolo Bonzini*. Interview published by Linux.com. Published: 2015-04-07. Accessed: 2015-04-08. Available at: http://www.linux.com/news/featured-blogs/200-libby-clark/821899-git-success-stories-and-tips-from-kvm-maintainer-paolo-bonzini.

143 This is supported by studies which note the participation inequality issue (Holtgrewe, 2004; Krishnamurthy, 2002; Kuk, 2006; McInerney, 2009).

cannot be relied upon to transmit all that is needed (even though with all the archiving options afforded by the digital text based information infrastructures, the situation is clearly different from the one Naur wrote about). Therefore, the continuity of a project relies on a continuity of involvement of at least a narrow group of human operators.

But with this claim, I don't want to simply shift the measure of importance in favor of humans. They still use delegators to transmit knowledge all the time, even though simply chatting in the IRC channel. In fact, since this kind of software development is taking place online, they have no other choice. Some tool must be used to delegate the information through the Internet infrastructure. Furthermore, the whole organization of development projects relies on delegation of formalized chunks of information, be it commits within source code management systems, or bug reports within issue databases. They allow for cloning, branching, prototyping, review, merging, testing, debugging, confirmation, and fixing – all the key practices that there are in free and open source software development.

In this way, commits and bug reports resemble what Bruno Latour calls inscriptions (Latour, 1986, p. 20; Latour & Woolgar, 2013, p. 236). They are mobile, immutable, flat (they are just text!), can be reproduced at little cost, can be recombined or superimposed. The mobility of digital text on the Internet is self-evident. Immutability is enhanced by the ability of Git to clone whole repositories. Thus, when a developer makes modifications to the source code, he can demonstrate their effect with compiling and running the whole self-sufficient version of the program. Similarly, the reports in Bugzilla are public. Should there be a report about a critical bug that no one is paying attention to, it is always possible to link directly to the report in public discussions, be it in the project's IRC channel, in a blog post that is displayed by the project's blog aggregator, or anywhere outside the project. The pressure to address these initiatives comes from the fact that speed of development and fixing of critical bugs is a major indicator of the project's health for anyone outside, including other development projects that consider using the software in question:

> Steve: No,
> Steve: the errors are in the tests
> Steve: not in our software
> Steve: and maintaining tests is exactly what you don't want to do ;)
> Tim: i know! i was saying that, the point in writing tests is only if they pass!
> Steve: Look: [link to a bug report]
> Steve: That's my problem
> * Tim looks

Steve: And the lack of activity on the report makes me doubt the reactiveness of the team
Steve: I mean it’s a fatal bug
Steve: [link to a git repository log]
Steve: I also look at that
Steve: 30 commits in 2013 is not really intensive dev
Tim: oh! I see, just 30 in complete yr!!
Tim: that’s worse than my local repos lol ;)
Steve: We had like 600 in pitivi alone ahaha
Tim: quite obvious!
Steve: A stable project might of course have less commits
Steve: But I don’t believe [project’s name] is one
Steve: so …
Steve: I would give a good look at our other options ;)

In this conversation, a maintainer and a newcomer discussed the project’s options with regard to automated testing. They were discussing what software to include into their infrastructure for testing. But there was a problem with the program they tried, it showed errors that the team found out to be caused not by the software being tested, but by the testing tool itself. As a result, the members of the Pitivi project would have to contribute to the testing software first in order to fix it before using it. And this is not a very pleasing outlook. Furthermore, the contribution would probably mean a longer involvement to keep the tests functional because the project does not seem to be “healthy”.

As an evidence, Steve provides a link to a report about a critical bug in which the response time from an assigned maintainer is rather long – weeks to reply to a comment and 10 months to fix the issue. As a further evidence, Steve provides a link to the project’s stable repository which shows that only 30 commits were made during a period of one year. This is a very low rate for a development project that has not yet reached maturity and stability. Even Tim, as a newcomer, recognizes that this is a problem and states that he has more commits in repositories that only he contributes to. The final point is made by comparing the development speed to that of Pitivi which has twenty times more commits for the same time period (and this does not include the commits made to the Pitivi backend – the GStreamer and Gnonlin/GNL libraries – to which the Pitivi developers regularly contribute to). As a result, Steve encourages Tim to avoid using this testing software and look for alternatives.

This decision, based on indicators described above, will have unwanted consequences for the project which was evaluated. First, it will not advance in gaining new users, which will further it from becoming a widely supported standard in its area of utility. Standard software typically has the advantage of other tools being designed to work well with it and so it often requires less effort to set it up in various environments. Its users will also be able to draw from a more extensive knowledge base as there will be more users that could provide information. Thus, becoming a standard is a process that could be characterized by bootstrapping – the more standard a software is, the easier it is for it to gain new user, which will result in it becoming even more standard.

Second, the evaluated project will lose potential future contributors which are often recruited from satisfied users that only need to scratch their own itch by fixing a minor issue. To see that in order to use a program for contributing to one project (in this case Pitivi), a newcomer would have to become a maintainer of a different project (the testing software in this case) is hardly appealing. But to see that this second project is performing fairly well and just needs a limited contribution from time to time is acceptable. In that situation, both sides would benefit – the first project for being able to draw on work that has already been done on the tool and by being able to influence the direction of its development, and the second project for gaining users and potential contributors to further enhance the speed of development. Here too, the bootstrapping process is at work once a project reaches the state of performing acceptably for others. And this state is indicated to others by immutable inscriptions described above.

The inscriptions are also flat. They usually consist of digital text organized by markup or programming languages. The use of pictures or geometry is not very common except for mockups or other design schemes. The result is a set of files that are rather easy to modify (in principle, it is much harder to make purposeful modifications as a part of the development process) and adapt to one’s needs. However, the application of automated manipulation, which permeates the use of digital text and, to a lesser extent, digital images, allows for reproduction only with the use of computational resources. This makes possible for those files to be translated and delegated multiple times without representing a significant resource burden.

Furthermore, these inscriptions can be recombined and superimposed. The availability of self-sufficient commits in public repositories makes possible for their taking over and inclusion into other repositories, be it the one owned by a reviewer or the repository containing the reviewed stable version of the software. In these repositories, the original commits are combined either with modifications made by the reviewer, or with subsequent commits pushed to the stable repository. Superimposition, on the other hand, manifests itself mainly in the review process. Here, the proposed commits are evaluated not only by the criterion working/not working, but also the overall design of the modification is assessed with regard to the theory that the core developers maintain. For example, the Pitivi maintainers would not accept a patch that introduces video editing functionality to Pitivi itself as they believe that the right place for this functionality are the libraries that form Pitivi's backend. However, in the past, this functionality was part of Pitivi. It was only in 2009, that the design decision was made to separate the interface from the functionality into different segments. Thus, by superimposition of concrete modifications to the source code, the reviewers are able to assess, whether they adhere to the basic development principles agreed upon within the project.

4.4.3. Black-boxing

The primary purpose of Bugzilla, Git and other tools is to organize work and make it connectable. They are not intended, nor used to work as archiving tools that records what exactly takes place during software development. Only reviewed and agreed upon commits are included into the main repository, and the prototyping space of personal repositories is periodically cleared when repositories are synchronized to share a reference point. All the failed attempts, experimental branches, or history revisions stay hidden when browsing the main repository. Furthermore, chat discussions, blog posts, issue database entries, or conceptual prototyping on the project's wiki pages are not even part of the information that Git handles. Having available detailed records of all the successful work that has gone into a piece of software is seductive but misleading. The full history of a project lies scattered over numerous wiki pages, blog posts, issue reports, chat discussions or mailing lists and need to be read against each other in order to reconstruct the theories that guide the development.

The complicated composition of a FOSS project, involving few platforms, several contributor roles, number of tools, hundreds of bug reports and thousands of lines of code results in the project appearing as a thick black-box to a newcomer. But not only to the newcomer. Given that contributors have certain roles and specialize in certain areas, they too face smaller black-boxes that represent areas that are out of their scope. This can mean two things. Either they never spent enough effort to familiarize themselves with what constitutes the composition of that area, or their familiarity is not recent enough.

Software (i.e. the tools and platforms used) is constantly developed and new versions are released. This means that it is a moving target for anyone who wants to stay current. New versions always mean changes in interface or, more importantly, in behavior. These changes are documented in release notes, which are usually very extensive. In order to deal with the extent and complexity, developers often use the strategy of emphasizing several important innovations and for the rest provide links to all the bug reports that were closed or commits that were made during development of the released version.

However, even this amount of raw information does not cover every change of behavior of the program. There will likely be unforeseen relations, unintended consequences or regressions to previously fixed errors. Every new version will have it's own unexpected ways of behavior. When we take this insight and multiply it with the number of software components used in one project, we may start to wonder how is any software development possible at all? The answer to this question lies in the constant effort to stabilize the components (debugging), to make them behave predictably. If a program is developed long enough, it might achieve what is called "maturity", which means that all major features are implemented and stabilized and that it now needs rather small modifications aimed at maintenance. And small modifications mean lower chances of introduction of unexpected behavior. At this point, most of the unpredictability is eliminated and the unreliable mediators turn into mostly unproblematic intermediaries.

Of course, for any software development project, it is advantageous to mobilize programs that are generally considered mature and thus, stable. It reduces the amount of friction, frustration and workarounds needed to run the project. However, old and mature programs will not provide all the cutting edge functionality which may be appealing for automating tasks that until now had to be done manually. As a result, development projects strive to balance the degrees of stability and innovation in the choice of their tools. Therefore, any project represents a blend of the agreed upon tools that introduce certain degrees of uncertainty together with their functionality.

The amount of information present in a project, together with an intricate network of interrelations and mutual dependencies creates a barrier that is not easily overcome. There is certainly considerable knowledge load assumed. I can illustrate this with my entering into the field and assuming the role that allowed me participant observation. I have been using Linux and open source software for several years prior to my research and it was still a challenging process for me. In my memos, I made a brief list of things I had to do before I could make the first contribution:

- discover a blog post on planet Gnome saying: documentation specialist wanted
- set up an IRC client
- connect to the IRC channel #pitivi on Freenode
- negotiate my role in the project
- install a Linux distribution as a second operating system on my computer
- install the dependency packages
- install development version of Pitivi by running an automated script
- get familiar with the application (download sample video file and play with it)
- get familiar with the user documentation
- go through a documentation to-do list of one of the developers
- go through documentation bugs filed against the project
- set up a suitable text editor
- get familiar with the Mallard markup language
- get familiar with the Git version control system
- create a Github account
- clone the Pitivi Git repository
- create my own branch and set up a Git repository on Github
- share link to the Github repository to others
- learn the “commit etiquette”

Note that the points are very different in extent. Some of them (set up an IRC client, for example) took only minutes while the others (like getting familiar with Git) required days of iterative effort. For most of the tasks, there were resources online that provided information on how to achieve them. There are manuals, troubleshooting posts on forums and blog posts on how to set up an IRC client and connect it to a channel. There are wiki pages dedicated to installing dependencies and the development version of Pitivi. There is a manual describing the Mallard language syntax. And there is a whole book online on how to use Git. One can follow the instructions, step by step to get the result with little knowledge about what is going on.

However, it is only when the bare information are related to one another during an intentional course of action, that knowledge arises. It was only when I got familiar with the current state of the application, its user documentation and a list of issues, that I was able to start contributing. The triangulation between the information on how the application behaves, how is this behavior represented in documentation and what are the known issues allowed me to possess knowledge that oriented my actions. All other points from the list above could be labeled as supportive. This is not to say that they are not important. I wouldn’t be able to contribute one bit without a text editor for example. But a text editor does not specify the content of contribution just as an IRC client does not specify the contents of discussions. And neither does Git or the other tools. It was the triangulation, which was aimed towards comparison of the current state of documentation and the state that could be reached that provided guidance for content.

Applied to programming, this takes us back to Peter Naur’s definition of the practice as matching a significant aspect of a real world activity to formal symbol manipulation (Naur, 1985), something that we could call modeling. The significance of modeling for software development is highlighted in a blog post of one of the main developers of Light Table, a text editor that was mentioned on Pitivi IRC channel as a new and possibly useful tool. In the post, its author is trying to argue against the new trend to consider programming as a new literacy:

> Reading and writing gave us external and distributable storage. Coding gives us external and distributable computation. It allows us to offload the thinking we have to do in order to execute some process. To achieve this, it seems like all we need is to show people how to give the computer instructions, but that’s teaching people how to put words on the page. We need the equivalent of composition, the skill that allows us to think about how things are computed. This time, we’re not recording our thoughts, but instead the models of the world that allow us to have thoughts in the first place.
>
> We build mental models of everything - from how to tie our shoes to the way macro-economic systems work. With these, we make decisions, predictions, and understand our experiences. If we want computers to be able to compute for us, then we have to accurately extract these models from our heads and record them. Writing Python isn’t the fundamental skill we need to teach people. Modeling systems is.

The author of the post argues that writing source code is not the key activity in software development. The most significant activity is to create models according to which source code is written. This is so because there is a large contingency in which activities can be modeled or mapped. As we have seen earlier, Keil-Slawik pointed at this with his claim that programs do not have to follow the sequential constraints of the mapped activities (Keil-Slawik, 1992, p. 182). The author of the blog post expresses it in this way:

> While properties of physical modeling are useful to us as guiding principles, the digital world offers us an opportunity to step out of their limitations.

When facing such contingency it is commonplace that rules are formulated and taught and that experience is associated with increased performance. This is why the author claims that modeling is the “fundamental skill we need to teach people”. However, modeling is not that independent from writing source code as the blog post seems to imply. The characteristics and possibilities afforded by programming languages and libraries interfere heavily into how the models are implemented in the end. Therefore, creating functional models requires the whole process.

Such process is usually iterative and depends on entities around for providing instructions and feedback. Although most interfaces in this environment provide the rudimentary feedback by not working, giving an error message, or performing an unintended operation if not handled properly, there certainly are sources of more elaborate instruction and feedback. As already noted in prior research (Hemetsberger & Reinhardt, 2006; Lee & Cole, 2003), browsing developer documentation or, more importantly, re-experiencing communication from chat or mailing list archives can be very instructive or even provide feedback (by responding to a particular search query). To be sure, hybrid cases of maintainers providing a link to documentation or archives in response to a newcomer query are common. But adaptive iterative feedback is the area where humans fit in. From guidance during setting up the development environment to reviewing the finalized commits, humans, and maintainers in particular, are able to provide sophisticated feedback to a newcomer and guide his actions from the start. In this way, they are compensating the knowledge needed to take purposeful action in this complicated environment until the newcomer develops his own.

Needless to say that this is a time consuming process for both sides and so one would think that there should be reciprocity in that the possible contribution the newcomer could make is worth the time spent on guiding him. But most of the time (the students of GSoC being an important exception), there is no contract that would obligate either party to anything. Thus, the guiding relationship with a newcomer is based on a tacit sense of what seems to be worth it. And with the topic of worth, we are approaching the question of how resources are used in free and open source software development.

4.5. Resources Propelling Development

Most FOSS projects start as volunteer projects to which developers contribute in their spare time. Pitivi is no exception in this regard. It was started as a student project and although some of its developers eventually got full-time jobs working on some of the underlying technologies, Pitivi itself is still a volunteer project. Thus, the continuous existence of the project depends on the ability to attract new volunteers and keep them engaged. Someone has to spend time by writing the source code, reviewing it, making design proposals, writing developer and user documentation, translating the interface and documentation. Software development is not simple, nor cheap. But still, there are many volunteer projects that manage to keep the inflow of resource high enough to survive. How is this possible?

To achieve this, the project needs to advertise itself as producing a well designed tool to provide a useful functionality and doing this the right way. Or, in other words, it needs to present itself as worthy of contribution. There is a specific page on the Pitivi website which targets potential contributors. This page serves as an index of communication channels and development platforms, but also strives to provide rationale for contribution. The quality of the overall design is demonstrated with the modular architecture which allows for reusing of functionality worked into the underlying libraries. Being a well designed software, Pitivi has many users, a lot of developers and a long history proving the project’s resilience. This implies several advantages for a newcomer: his contributions will be distributed widely and so he will be able to affect the experience of many users; the number of developers and history length means that he will be joining a well established project with maintainers that will be able to mentor him and review his contributions appropriately. Strategically, it also means that the newcomer will not be at the mercy of decisions made by a single person as this might be the case in one-man projects. At the end of the rationale, the emphasis is placed on the single most important point: “when you contribute to Pitivi, your time is not lost”.

Time is a valuable resource and even more so when spent by highly qualified workforce. There is no one else who could contribute to a project because, as we have seen, high qualification is necessary to overcome the barrier to involvement. Therefore, it is no surprise that the demographic from which I have seen most of the contributors to originate was the one that abounds with time and qualification - students. Considering that Pitivi was a student project in the first place, that its current maintainers got involved when they were students, and that currently newcomers are also students in most cases, this seems to be the most common background among contributors.

This is so for several reasons. The first being that there is an institutional support for student involvement in open source projects. Once a year, Google launches its stipend program called Google Summer of Code. It allows organizations managing open source projects to select students who will be paid to contribute over the period of few months. However, there are several limitations that reveal the interconnectedness with areas outside software development. First, the organizations involved in the program must develop software under a license approved by the Open Source Initiative (OSI). Founded in 1998, this California based standards body serves as a maintainer of *the Open Source Definition*[144] and a reviewer of popular licenses. In this configuration, the rules that for some (traditionally in the Free Software movement) have ethical or moral significance, translate into eligibility to gain resources.

Second, the students must be able to provide a certificate of their enrollment into an accredited institution. Furthermore, they must be eligible to work in the country in which they will reside during the program. Finally, students residing in countries with whom the U.S. law prohibits engaging in commerce are not eligible to participate. Here, we can see how the program is connected to the countries of residence of the respective parties. In the present context, the approval of a license by the OSI and a decision to use the license by an organization developing software, together with an approval from an educational institution, while all taking place in the right geopolitical area creates an intersection at which the quality of applicants only begins to be assessed and through which the funds are potentially available.

The incentive of earning several thousand dollars in the period of few months is the first at hand when we consider the motivation of new contributors (provided they are involved in the stipend program). But there are also other incentives. Starting with the most pragmatic ones, further learning and earning experience in the field of study is obvious. In this regard, active contribution to a FOSS project is similar to an internship. And considering that developers in some projects are paid by private companies for their work, it might also be an “internship” very close to a potential future employer. Furthermore, there is one important advantage for students contributing to FOSS projects. The combination of the project’s transparency (in this specific sense) and emphasis on recording authorship allows students to demonstrate their skills by simply pointing to the work that they have already done and that is

144 This is a rather short document defining the open source development method in ten points. The full text is available at http://opensource.org/docs/osd.

marked in publicly available records. Just as the personal repositories create a leverage for reviewing and including commits from a contributor, the commits included to the main repository create a leverage to accept an applicant.

Apart from the pragmatic ones, there are also other incentives to be added to the list. One of them is based on the value of the developed software and it was expressed most strongly at the start of a fund-raising campaign:

> Free and Open Source video editing is something that can help make the world a better place, as it gives people all around the world one more tool to express themselves creatively, fight oppression, create happiness and spread love.

The expression does not go as far as changing the world, it is not revolutionary. It just aims to add one thing to the list of what is good in the world. The assets that the software offers permeate the personal domains (happiness, love), possibly reach to professional relations (creative self-expression), but might also serve a political struggle (fighting oppression). This hints at the modes of existence[145] that the developers envision for their creation when it leaves the haven of the main repository.

Furthermore, one of the Pitivi maintainers shared his motivations in a presentation at the GNOME User and Developer Conference. According to him, there is a variety of subtle motives like scratching a personal itch, making friends with other contributors and thriving from the enthusiasm and trust that runs in the group. But one of the major motivations is a value which, according to his own words, resonates with most developers working on Pitivi. It is expressed on the main page of the Pitivi website as a text in the largest print:

> We believe in allowing everyone on the planet to express themselves through filmmaking, with tools that they can own and improve.

The expression has two important dimensions. First, the words “everyone on the planet” means (as explained by one of the maintainers in a presentation given at the GUADEC conference) that the software is developed explicitly with no market segmentation in mind. The software should be suitable for anyone from kid in school or an activist to independent professional filmmaker. In this, the developers see one of the main differences between their program and software that is developed for commercial purposes and thus has to be tailored to a specific group of customers. In a way, the Pitivi maintainers claim that providing the software free of charge liberates them from the restrictions originating from classical business

145 In the sense which Latour attaches to this expression (Latour, 2007a, p. 24).

strategies.

For the second dimension of the expressed value, the words “own” and “improve” are key. Users truly own the program only when there is the possibility to see its internals, discover how it works, modify it or reuse some of the work that has been done. With proprietary software, this is not possible by definition. Its source code is not distributed publicly and any attempts at reverse engineering or modifications are forbidden by licenses. From this point of view, the users of proprietary software only get the right to use the software with all other rights denied. Therefore, users are dependent on the decisions the provider of proprietary software makes and are left with no options to steer the direction of development or maintenance. While in the open source model, a qualified user would be able to trade her right to participate on the decision-making process for spending time volunteering for the project. In this context, ownership is constituted by access to information (most importantly the source code).

This point is based on one of the basic ideas the movement formed as a criticism of proprietary software. One can find it in popular interpretations of the GNU project such as the one provided by the British actor Stephen Fry:

> If you have, I don’t know, plumbing in your house, it may be that you don’t understand it, but you may have a friend who does and they may suggest you move a pipe here or stack [something] there or valve somewhere else. And you’re not breaking the law by doing that are you? Cause it’s your house, you own the plumbing. You can’t do that with your computing, you can’t actually really fiddle with your operating system and you certainly can’t share any ideas you have about your operating system with other people because Apple, Microsoft who run the two of most popular operating systems are very firm about the fact that they own that and no one else can have anything to do with it. Now this may seem natural to you, why shouldn’t they? But actually, why can’t you do with it what you like, why can’t the community at large alter and improve and share, that’s how science works after all. All knowledge is free and all knowledge is shared in good science. If it isn’t, it’s bad science and it’s a kind of tyranny.[146]

Fry uses the classic opposition between big companies that only provide a restrictive license for using software they own and free software that is unrestricted in this sense and therefore can be owned by anyone, while the latter alternative is further legitimized by the positive associations in the images of “community” and “good science”. However, ownership, in this sense, relies on the potentiality of engaging with the internals of a program (or an operating

146 *Stephen Fry talks about free software (GNU 25th Birthday)*. Published: 2009-04-28. Accesses: 2015-03-25. Available at: https://www.youtube.com/watch?v=YGbMbF0mdPU.

system) and therefore, it is substitutable (if someone does not understand it, someone’s friend might). But my point is that the relationship between information access and ownership goes one step further, because one can have the source code available and still does not own the software anymore than if it was proprietary. One needs to engage with the information available, appropriate it, in order to really own the thing. As I will show at the end of this section, such relationship between knowledge and ownership serves as a strong incentive for involvement of private companies in FOSS projects.

However, the trade-off between spending time and gaining rights is probably not the most important part of volunteer motivation. If only for the fact that contributing consumes large amounts of time and leaves the desired result uncertain (bugs may prove to be harder to fix than expected, commits might not pass review, other design choices may prevail). But there is added value to this trade-off that takes it to a new level of attractiveness – that the results of the volunteer work are made available to everyone who uses the software. The volunteers know they are giving a gift to the wider community and they are prepared to accept praise for it. What they give away in hours of skilled work for free, they gain in their status. Apart from providing positive gratification, the status can also be translated into resources. Being employed for one’s individual merits is one way of doing so. Public fund-raising may be another one.

When Pitivi version 0.93 was released, one maintainer emphasized the volunteer nature of the project on his blog: “0.93 is the result of continued efforts in our spare time - occasional hacking during vacations, nights and week-ends”. The emphasis was not random, it was setting stage for a fund-raising campaign that aimed to intensify the development by providing funds to the developers so that they could spend more than spare time on the project. “Just imagine what could be achieved if Gary and Randy could be funded to work full-time towards bringing us to 1.0!” reads the next sentence.

The fundraising campaign was no easy undertaking. It required preparations months in advance – an agreement with a legal entity representing the project was made, a video was shot, a separate website was established, a press release was published and payment and voting mechanisms were put into place. What the fundraiser meant for the developers (and why they invested so much effort into setting it up) was that they felt they spent a lot of time on it for free on their own and now they needed a push to finally leave the testing stage and reach the 1.0 stable version. It was a moment charged with emotional valence, even though the developers felt justified in asking for the funds. One of them wrote this into his blog post:

> I’m writing this the day before launching the campaign, and I have the website in the background, taunting me with its “0 € raised, 0 backers” message. Fortunately I also have the spinning social widgets to cheer me up a bit, but it’s not exactly enough to get me rid of my anxiousness. I know that what we do is right, and that requesting money for stabilization first is the correct and honest thing to do.

At first the fund-raising campaign went well and received coverage by news sites that delivered it to audiences interested in free and open source software. The coverage got quickly translated into funds as the news was getting to the potential donors. However, after the initial burst of enthusiasm, the campaign stagnated. It soon became clear that the fundraising campaign will not achieve the target amount. This sparked a discussion about a fall-back plan:

> Eric: Also, maybe you should set a date when you make a plan, considering there is a chance you won’t get to 35.000?
> Steve: “make a plan” ?
> Steve: Eric, ?
> Steve: fall-back plan ?
> Steve: Gonna go sleep, if that was your question Eric the plan remains unchanged, we just do it with less money and more good will :)
> Steve: Not that good will was lacking, but it’s an apt replacement for money, just means it will take more time for us to do the stabilization as we’ll go on taking contracts on the side

Even though Steve claims that money and will are interchangeable, they seem to have different characteristics. The will to work on the software is there, no matter how much money is present. Money just provide more time for the will to materialize. More money implies smaller number of other contracts and more focus on Pitivi development. Thus, money seem to play a little role in deciding whether to work on the software or not. It plays a much bigger role with regard to how much time the volunteers are able to spend, which translates to the speed of development.

This appears to be the opposite of how private businesses operate. There, the investment return is the biggest criterion in deciding whether to develop a piece of software or not. Furthermore, as Brooks (1995) claims, within private businesses, the allocation of more resources to a project sometimes leads to a later delivery. By this, I don’t mean to imply that the open source method of development, being completely opposite to the proprietary, is incompatible with any form of business. As we will see in the following paragraphs, it is quite the contrary.

The economy of free and open source software development does not stop with the motivation for contribution. Besides humans, all other entities present in a project must have a reason to be there. The whole infrastructure described up to now is dependent on a continual inflow of resources. In other words, someone has to provide the server time for running Git repositories, Bugzilla, the IRC channel, or the project's wiki. Without these elements, free and open source software development would not be possible in the way it is now commonly performed. The presence of these elements is possible due to existence of non-profit organizations and specific business models of some private companies.

First, there are non-profit organizations like the GNOME Foundation. Operating from a donation or sponsorship based funds, the foundation provides several key services for the individual projects while also maintaining libraries that form a low-level programming infrastructure. The foundation serves as a legal body that represents the projects in formal relationships with other organizations. This allows the projects to have fundraising campaigns or to participate on the GSoC stipend, even though the informal mode of organization of the projects themselves would disqualify them in such circumstances. Thus, the foundation shields the individual projects from the necessity of establishing formal organization. In a way, it provides them the opportunity to reap some of the benefits formal organizations enjoy while allowing them to stay informal.

Furthermore, the foundation also controls some server infrastructure which is provided for the individual projects. Therefore, the Pitivi main repository is hosted by GNOME. Also, Pitivi uses the Bugzilla instance provided by GNOME as its issue tracker. When I entered the project, I was made aware that it was not always like this. In the past, the main Pitivi repository was hosted elsewhere and there was a duplicate hosted by GNOME that was used to collect translation and documentation contributions. This points to another type of services that the foundation provides. It is able to attract contributors that translate and document the software developed by the individual projects. It usually takes orders of magnitude less time to translate or document a program then to develop it, and therefore it makes sense to associate those activities under an overarching body so that the contributors may move from one project to another and still stay on the same infrastructure.

Second, there are services that are provided by some companies for free. A typical example of such a case are personal repositories of the individual developers. These are hosted by GitHub, a company which specializes in managing Git repositories. The company makes revenue by offering paid plans for individuals and organizations that require private repositories. However, there are free plans for both types of customers which offer unlimited number of public repositories. This configuration is tailored to be used as an intermediary in volunteer projects. A newcomer aiming to contribute to a FOSS project just needs to create an account and configure a repository to use the service. For the company, it obviously serves a marketing function and creates a positive public image. But, more importantly, by drawing a large number of developers to use its services for free, the service is becoming the de-facto standard in the market segment of providing source code management services.

The case of the main Pitivi repository is different. It is located in a space provided by the GNOME Foundation which in turn has its servers hosted by Red Hat. Red Hat is also a company with a business model revolving around the open source development method, but one which is quite different from that of GitHub. This constitutes the third option. Red Hat is sponsoring a number of free and open source projects ranging from the Linux kernel to the Fedora Linux distribution or the GNOME desktop environment. The technologies derived from these projects form a portfolio of what the company is able to deploy and maintain for a customer. However, the software itself is not sold, it is the services around it (deployment and maintenance) that create the revenue.

In this context, the communities around the projects Red Hat sponsors serve as sources of innovative and tested technologies, while also providing a pool of skilled workforce (which is already familiar with the products) to recruit from. On the other hand, the company keeps the inflow of resources to the project by providing server time, sponsoring events, and acting as motivational force drawing in those who seek careers related to open source technologies. At the same time, the developed software is still publicly available as it employs licenses approved either by the Free Software Foundation or the Open Source Initiative. The close relationship that a business might establish with a FOSS project can be illustrated precisely by the case of Red Hat:

> But of course, all of that value that Red Hat is able to offer its customers is built on the contributions of the much larger open source community, both as a whole, and the specific communities that feed directly into Red Hat products.
> …
> Our most notable involvement is with The Fedora Project, the results of which feed directly into Red Hat Enterprise Linux.
> …

> Fedora releases come out every six months, showing the edge of innovation and new features. Red Hat engineers participate in that process from the beginning. (However, 65–70% of Fedora's code is maintained by volunteers.) Then, Red Hat dedicates its quality assurance resources to testing, hardening, and certifying those features to ensure that they meet the requirements for enterprise-level interoperability and performance. Code that started in the upstream community becomes the code that Red Hat customers … rely on to solve their daily business problems.[147]

However, this is only half of the picture. After being used and modified by Red Hat, the source code is made publicly available again which makes it possible to be reused in a different community run Linux distribution called CentOS. From the CentOS perspective, the process is described in the following way:

> The upstream vendor is using open source (mostly GPL) software in their business model. They take software that other people write (Gnome.org, X.org, KDE.org, OpenOffice.org to name a few). They repackage the source files into RPM format for redistribution. Because they chose an open source model to obtain the software they distribute, they must provide their source code to others. That is how the GPL works. The upstream vendor provides much added value by creating the Source RPMS and distributing them. They also fix problems in software and provide feedback to the software developers … this is what makes open source software work.
>
> The CentOS Project takes the publicly available source packages (SRPMS) provided by the upstream vendor and creates binary (installable) packages for use by anyone who wishes to use them.[148]

A similar type of relationship (with the difference that after modification, the source code is integrated back into its original community source instead of being reused by another project) can be found between the GStreamer project and Collabora, a company sponsoring its development. Collabora employs several GStreamer developers (some of which contribute to Pitivi in their free time) and for Pitivi, it provides server for building and testing daily versions of the program. The emphasis is put on developing GStreamer and other backend technologies as these provide the functionality that Collabora can in turn offer to its customers. But in the past, there were also developers assigned by the company to work on Pitivi itself as it represented the storefront demonstrating what the underlying technologies are capable of.

However, the relationships between sponsoring companies and communities often

147 *Q&A. What Is the Secret of Red Hat's Success?*. An article linked from official Red Hat website. Published: January 2012. Accessed: 2015-04-09. Available at: http://timreview.ca/article/513.

148 *Frequently Asked Questions about CentOS in general*. Last edited: 2015-03-20. Accessed: 2015-04-09. Available at: http://wiki.centos.org/FAQ/General#head-4b2dd1ea6dcc1243d6e3886dc3e5d1ebb252c194.

exhibit tensions.[149] This is so because the development decisions preferred by companies and those preferred by communities (or their parts) can diverge. In such a tug-of-war, companies have a more advantageous position as they are able to mobilize developers that they employ to spend more hours of work than volunteers. Combined with the rule that decisions (rather the smaller ones, bigger decisions are left for governing bodies like councils in which, however, the companies also have their representation) are made by the ones carrying out the work (and review), companies can gain an upper hand just by employing the key actors. On the other hand, a community around a sponsored project is very valuable for any company and so their steering power is counter-checked by the possibility of the community abandoning or forking the project if a controversy reaches sufficient intensity. Volunteers are not assessing the project only before they start contributing, this is an ongoing process. Therefore, a perceived lack of good design decisions combined with the feeling that one does not get to be heard can lead to lowered willingness to contribute. As a result, the relationship between companies and communities involves careful balancing. Companies may lose a wealth of volunteers and risk competition if the project gets forked. Communities may lose resources from sponsorship and risk criticism for duplication of efforts if forking takes place.

But to return to the resources issue, one can ask: how can there exist a business model around software, that is (together with it's documentation) publicly available? The answer lies in the strategy I already mentioned with regard to Red Hat. It is not the software itself that is sold, the services around it are. The publicly available information are complex enough to require significant amount of effort to be processed in order to be put to use. This can be demonstrated on a case when a newcomer (Ben), after finishing his first bigger task, thought about taking on a much more demanding one:

> Steve: As for the task you're thinking about, to be done correctly, it would certainly require changes in blender, and intimate knowledge of its code base, plus willingness from their upstream to expose an API

149 A well known example of such tensions was the relationship between the community around the Ubuntu Linux distribution and Canonical, its sponsoring company. Canonical was often criticized by members of the community for taking decisions behind closed doors and introducing them as fait accompli. The tensions culminated in 2013, when the website www.fixubuntu.com was established to instruct Ubuntu users on how to deactivate offending features introduced to Ubuntu by Canonical. At first, the company attempted to enforce taking down the website by using its ownership of the Ubuntu trademark, which resulted in an outburst of controversy over the step. Eventually, Canonical founder Mark Shuttleworth apologized for this step and the website is accessible to this day. Such tensions are largely absent from my descriptions because I did not run into them during my fieldwork. This may be so because Pitivi is largely sponsored indirectly (for example through infrastructure provided by the GNOME Foundation) or the involvement of direct sponsors is limited and does not currently provide developer time (as in the case of Collabora which provides developer time for GStreamer, but for Pitivi per se, it provides "only" a server).

> Steve: I am thus strongly hinting at you that it is *hard*, and will require changes in blender to do correctly
> Steve: I don't know if you already had a look at blender's code, but it's *huge*
> Ben: I suppose.
> Ben: I will need to read Blender code.
> Ted: I can concur on that, maybe I was not clear enough (but I thought I was) to Ben earlier today when I was hinting that you are *vastly* underestimating the complexity of something like integrating blender with pitivi
> Ted: I mean even if it was one of the gstreamer core devs doing it I would imagine a year of work
> Ted: Ben you realize that Pitivi is 18 thousand lines of code and Blender is *2 million* lines of code? you can't just go "read its source code" :)
> Ted: I mean you can… but we'll see you again in 10 years
> Jim: reputedly the VSE (video sequence editor) in blender is a nightmare
> Jim: terrible C code from the early 90s that nobody loves

In this conversation, the Pitivi maintainers (Steve and Ted) discourage Ben from taking the task because of the difficulties it presents. To get an impression, it suffices to go through the emphasized points (marked with asterisks): the task is *hard*, the source code is *huge* (which is reiterated by stating that it has *2 millions* lines of code), and Ben is *vastly* underestimating the complexity of the task. The maintainers point out that this is not a suitable task for someone who has been around for just several months, because, even for someone as knowledgeable as a GStreamer core developer, the estimate time for completing the task would be a year. The conversation culminates in the statement that "you can't just go read its source code", pointing out that approaching such a task head on would require an amount of resources (illustrated by the expression "see you again in 10 years") that are out of scope for any individual there and that the codebase could perhaps be better appropriated by interacting with it (doing smaller tasks). To this, Jim adds a remark about the state of the codebase ("terrible C code from the early 90s that nobody loves", it is a "nightmare"), which, together with its length, is also a significant indicator of the difficulty of dealing with it.

Such discouragement from experienced developers is indicative of how large barrier complexity is even when approaching documented and publicly available source code. For a more elaborate description of the problem, I can reach for the one provided by Brooks in his classical essay on programming:

> Large-system programming has over the past decade been such a tar pit, and many great and powerful beasts have thrashed violently in it. Most have emerged with running systems – few have met goals, schedules, and budgets. Large and small, massive or wiry, team after team has become entangled in the tar. No one thing seems to cause the

> difficulty – any particular paw can be pulled away. But the accumulation of simultaneous and interacting factors brings slower and slower motion. Everyone seems to have been surprised by the stickiness of the problem, and it is hard to discern the nature of it. But we must try to understand it if we are to solve it. (Brooks, 1995, p. 4)

Although this characterization of the difficulties associated with software development is dated, it still retains its point. Even though it is normal today to make documented source code publicly available, it does not mean that the complexity barrier will disappear.

Returning to the question of business models, the answer seems to lie in the fact that employing developers who are already familiar with a codebase represents a considerable advantage for any company, as it is able to put the information to use without the need to spend large amounts of resources on overcoming the barrier. In other words, the company's competitive advantage is possessing knowledge (through employing developers). And it seems that possessing knowledge provides the advantage to such an extent, that giving away the information is not threatening the business strategy.[150] This highlights the value (and the difference) of possessing knowledge compared to just holding information.

Indeed, when we look at what the companies involved advertise as their competitive advantage, we can see that it is the employment of experienced contributors who carry significant expertise. Collabora is particularly explicit about this on their website:

> Whether you are getting ready for a new product development or upgrading a current one, adopting Open Source can seem challenging. Collabora will save you time and money by helping you leverage existing Open Source software so that you can focus on the truly differentiated value of your product.[151]

> Collabora employees are not just professional Open Source developers. They are also longtime contributors and form an integral part of the Open Source community. And the years they have spent exploring projects and distributions and forming relationships with members of the Open Source community have resulted in expertise they can pass along to you.[152]

Browsing through Collabora's portfolio, it offers the following services around GStreamer: consulting assistance, training, custom development, architecture (design review and creation). Note that three of the four offered services do not involve writing new (or

150 Moreover, as Josh Lerner and Jean Tirole note, making source code publicly available requires it to be adjusted to make orientation and contribution easier (Lerner & Tirole, 2002, p. 226). Therefore, even though the barrier is being actively lowered, it is still so high that knowledge of the codebase is very valuable.

151 *Services: Planning*. Collabora marketing materials. Accessed: 2015-04-09. Available at: https://www.collabora.com/services/planning.html.

152 *Services: Guiding*. Collabora marketing materials. Accessed: 2015-04-09. Available at: https://www.collabora.com/services/guiding.html.

modifying current) software and selling it as a product. The one service that does (custom development) could potentially involve writing source code that is not made public. This is also explicitly indicated on the website:

> We believe that developing the vast majority of software publicly in a collaborative fashion must become the standard. Of course there will always be room for differentiated value; we don't suggest that every line of code must be made public (although that would be nice). Assisting customers maximize their use and contributions to Open Source is our raison d'être.[153]

This excerpt suggests that extensions of the publicly available source code that are held private are one of the ways to add a differentiated value to the customers' product. However, the text hints at the preference to make even the extensions public ("that would be nice") by integrating it back into the publicly available codebase.[154] This process is elaborated elsewhere on the website:

> Collabora has helped many customers to upstream their software contributions to existing Open Source projects. Compliance with the terms of Open Source licenses governing the software our customers use is of paramount importance to Collabora. Whether the code is originally developed by Collabora or our customers, we help our customers lower their maintenance burdens by ensuring that all relevant code is merged upstream. Collabora is committed to maintaining the code as part of our involvement in the Open Source community.[155]

This excerpt shows that integrating extensions back into the open source codebase is not only a potentiality, but an ongoing practice. The incentive for Collabora customers to do this is expressed here as the possibility to "lower their maintenance burdens". The reason for this is that creating modifications of software that are held privately constitutes a parallel (privately forked) version of the software that has to be maintained separately. Such maintenance consists of monitoring the development that takes place in the public codebase and manually

153 *Paving the Way*. Collabora marketing materials. Accessed: 2015-04-08. Available at: https://www.collabora.com/open-first/open-source.html.

154 To protect the extensions from the necessity of being published, which would be normally required by the GNU GPL, some companies resort to strategies like dual licensing or requiring contributors to sign contribution license agreements (CLA), which state that the code they write may be relicensed in the future. We can see that the requirement made by the GPL to distribute derivative works under the same conditions as the original work is problematic for some parties and requires further procedures to be dealt with. This may be one of the reasons why the so-called permissive licenses (such as the MIT or Apache licenses) that lack this requirement have been gaining momentum over the past several years. (*What are the Most Popular Open Source Licenses Today?*. A report based on data from Black Duck, a company specializing in monitoring FOSS projects. Published: 2014-11-14. Accessed: 2015-05-13. Available at: http://redmonk.com/sogrady/2014/11/14/open-source-licenses/)

155 *Services: Integrating*. Collabora marketing materials. Accessed: 2015-04-09. Available at: https://www.collabora.com/services/integrating.html.

including all modifications that result from the public development (which may also mean resolving conflicts that arise between public and private modifications). It follows that such approach requires significantly more resources than integrating the private modifications into the public codebase in which case every subsequent modification is built upon them and consistent with them, thus doing away with the costs of separate maintenance. As we can see, resharing of modified source code is not only a condition introduced by the “viral feature” of the GPL licenses, but it is also backed up by incentives based on cost and effectivity.

We can see how practiced knowledge empowers volunteers and businesses alike in the sense that it allows them to steer the direction of development in FOSS projects. For volunteers, this represents an opportunity of raising status which can be eventually translated into resources through donations, fundraisers, or employment. For businesses this represents a competitive advantage in overcoming the barrier of technical complexity. This advantage is often monetized in specific types of business models aimed at providing services around publicly available technologies.

In sum, we are now able to see the rough outline of the network that allows many FOSS projects to sustain themselves. The project must be able to draw volunteers and motivate them to stay. It may use some of the services provided by non-profit organizations or by companies whose business model involves free services. Finally, if strategically placed, the project may enjoy the benefits of direct sponsorship from a private company. Most of these sources are affected by the design decisions and quality of implementation within the project as companies and volunteers alike will evaluate the worth of the project before contributing. Therefore, software is often initially developed privately and it is made public only after reaching a certain level of completeness. It is because at this point, the developers are able to demonstrate their skills and motivation. From the point of view of potential contributors, this serves as a guarantee that the initial developers are able to deliver and are willing to do so as they already invested a significant amount of effort before reaching out to public.

5. Assembling a FOSS Project

We are now in a position to see the overall shape of a FOSS development project. The projects are architected to facilitate and encourage autonomous retrieval of information and there are several structures that are particularly significant in this regard.

On a most basic level, there is the upstream first principle, which implies that the maintenance of functionality inherited from other programs should be done in the original programs, not in the one using the functionality. This is the difference between “being a good citizen” and “doing your own thing in your corner”.[156] This approach helps to keep generalized functionality allocated in reusable libraries. Therefore, the functionality does not have to be recreated for each new program, but also (and perhaps more importantly) it allows for concentration of expertise around a technology that is considered standard.

Furthermore, licensing is key in establishing conditions for unconstrained information flows. But, as we have seen in the case of the GStreamer library, using just about any license permitting free reuse of information does not lead to frictionless sharing. Using non-standard licenses (with a clause, for example) introduces the necessity to negotiate the terms of reuse with some parties and therefore, as Yochai Benkler would point out, raises transaction costs (Benkler, 2006, p. 109). As a result, the use of standard licenses, recommended by significant parties (such as GStreamer developers), or vetted by definition-maintaining organizations (such as the Free Software Foundation or the Open Source Initiative) is imperative for autonomous retrieval of information.

Apart from being part of the sharing infrastructure, licenses can carry a significant moral load, while at the same time be otherwise politically agnostic. This is the case with the GNU licenses created by the Free Software Foundation, which considers hoarding software under different licenses a moral fallacy. On the other hand, the GNU licenses explicitly preclude restrictions on reuse based on any further terms. This ambiguity is also reflected in the images of a “better world” put forward by organizations such as the GNOME Foundation. How exactly will this better world look like is nowhere specified, but it certainly involves the use of FOSS development model.

But then again, this is no ordinary counterculture. The primary activity is not waging

156 It also indicates that free and open source software development assumes a collective form of authorship, similarly to, for example, users of the Creative Commons licenses who see the accessibility of their works as an acknowledgment of the intellectual debt they have toward their influences. Minjeong Kim contrasts this position with what he calls a “private property vision”, in which authorship is seen as an exclusively individual achievement (Kim, 2007, p. 195).

political fight and trying to achieve a state of utopia. The primary concern here is branching out and establishing self-sufficient alternatives. Everything here revolves around translating ideas expressed in natural language into formal languages (either markup, or, more importantly, programming languages). Once translated and compiled, the actions devised by software developers are fully automated and delegated to the computers of their users. This translation is achieved by writing, (or making modifications to) the source code. The modifications are tracked by version control tools, translated into standardized form and delegated to a public repository in order to be appropriated by others. Personal repositories then represent sandboxes for experimentation, prototyping and learning. Once a contributor is confident in her work, she makes a pull request to indicate to others an intention to merge the work into the main development branch.

What follows is peer review, which is the center of gravity for power relations. Compared to the collaborative practices implied by Mediawiki, the review process utilizing Git has specific spatial and temporal characteristics. Here, peer review takes place in a distinct place (personal repositories) before the work is incorporated into the main repository. On the Mediawiki platform, there are no personal repositories, there is one central repository which is by default open to modifications. Peer review takes place only afterwards as other users browse the content. While this model seems to produce acceptable results for the Wikipedia community, maintainers of the Pitivi project decided to make further restriction in order to avoid spam and vandalism. Considering that the Pitivi project is orders of magnitude smaller than Wikipedia and that maintaining wiki pages is not the main concern here, it seems reasonable to suggest that the wiki collaboration model works[157] when a certain threshold of number of active users is exceeded, while the model utilizing Git works even when the numbers of contributors are low.

Proceeding exactly in the opposite direction, debugging represents temporal and local reversing of the blackboxing introduced by source code compilation. Debugging is a procedure which translates defects initially formulated in natural language into formalized descriptions known as stack traces. These are part of bug reports which delegate the defect and its description into publicly available Bugzilla database and provide space for negotiation, initial prototyping, and also a reference point. The contents of either Bugzilla database or Git repositories can be seen as inscriptions, establishing non-human allies to be mobilized in

157 The term “works” here means that the model leads to performance which is close enough to the intentions of its users so that they do not decide to alter it.

demonstrating a claim and persuading others inside or outside a project. Inside a development project, this may be used to alter design decisions or allocation of time by individual contributors. Outside of a project, these information are used for assessing the project, its health and future prospects. This, in turn, affect the rate of adoption, which is also the rate of reaching possible contributors, in a bootstrapping process of becoming (or not) a standard.

There are conventions which structure the procedures such as writing the source code (the 80 column rule), making commits (the commit etiquette) or merging reviewed work (opting for the lowest number of conflicts in revision tracking). But as it turns out in some cases, the conventions are negotiable. If we look for something more stable, or pressing, we would have to focus on the behavior enforced by the tools used. These are not negotiable, and are hard to alter. Granted that tools developed in accordance with the FOSS model are preferred, it is in theory possible to modify their behavior. But doing so requires effort going beyond the modification itself. Either one can opt for modification in cooperation with the developers of the tool. In this case, the developers will have to be persuaded that the modification is necessary and it will also have to go through the standard review process. Or one can opt for making the modification without cooperating the developers of the tool but this will establish an alternative version of the software which will require maintenance (including updates from the official version) in the future. As a result, such situations are often dealt with by searching for a different tool that fits the requirements.

A similar dilemma can be found when we consider dependencies. However, from what I have experienced, there is a stronger tendency to contribute to dependencies than to tools. Some of Pitivi’s core developers contributed more to GStreamer (its main dependency), than to Pitivi itself. This preference seems logical given that the dependencies are presupposed in many parts of the source code and without them the developed program would simply not run. Therefore, switching a dependency always means modification to the source code (and, perhaps, functionality of the developed software), while switching a tool means that the change is contained within the project’s infrastructure.

But even though the individual tools may vary, we can always find certain types of tools put to use in a project. Together with programming languages, tools like text editors, compilers, debuggers, or version tracking systems establish the necessary minimum for a FOSS project. These tools make it possible to perform not only the pragmatic action (text editors allow for writing, compilers to compile, version tracking systems to push and pull to code around), but also the epistemic action – programming languages allow developers to *think* in a way that is designed to be automated; text editors allow developers to *see* the source code, to perform searches, or to display unwanted patterns (like trailing spaces in my case); debuggers allow for *seeing* the internals of a program while running in order to identify which part is responsible for an error; version tracking systems allow to *see* differences in the source code so that the work of an individual can be *known*. In this sense, the tools serve as wideware of software development.

Thus, the epistemic action performed with those tools leads to establishing knowledge necessary for contribution. While this kind of knowledge is specific for every project, the knowledge needed to operate the tools is not (or only to a limited extent). In the area of tool use, the knowledge problem can be circumvented by resorting to standards, which will not only decrease the barrier to entry for most possible contributors, but also make the output of their use standard. On the other hand, the problem of decontextualization is most pressing in the area of design artifacts. These intermediary results of work need to be examined thoroughly each time they appear. As I have already pointed out, their examination requires the use of tools to be possible at all, but the added value is that a consistent use of tools yields consistent output. Thus, the deployment of standard wideware facilitates the emergence of knowledge in different places, it allows knowledge to "travel".

This process is very valuable because, as we have seen, knowledge is closely related to practicing rights traditionally associated with ownership in this context. Users do not own proprietary software, they are only licensed to use it. This is a common claim in the FOSS world, which, as Coleman (2013, p. 6) shows, leads to the prevailing opinion that source code should not be subject to property rights. Indeed, by using FOSS licensing, the authors of source code voluntarily abandon most rights associated with ownership. Formally, this prevents ownership from being exercisable (unless, of course, the licensing conditions are violated). FOSS licensing disposes of ownership by ascribing the most fundamental rights associated with it to anyone. But practically, if we consider ownership to be defined exactly by those activities the licenses are explicitly permissive about (that is, source code access, modification and redistribution) we can see that they are not practiced by just about anyone. The most significant prerequisite for doing so is holding specific kinds of knowledge. Hence my claim about the close relationship between ownership (at the level of practice) and knowledge. Although this claim has one important caveat – it uses ownership in a sense, in which it is no longer an exclusive right. In this context, ownership is redefined to a form in which it can be exercised by multiple parties simultaneously.[158]

The distribution of source code and provision of free software rights introduces ownership only potentially, we could say (with reference to Rob Shields (2003)) virtually. It is only by exercising the rights to study, modify and redistribute the source code that the ownership is actually performed. These knowledge intensive activities (performed through pragmatic and epistemic actions enabled by tools) allow for knowledge accumulation which, in turn supports further activities. In its course, this bootstrapping process renders ownership actual. This claim is further supported by the observations of business models which do not rely on (formal) ownership of information. Instead, these business models are based on employing actors knowledgeable of publicly available information and through them, providing paid support for their customers. Therefore, even though a product (software) is publicly available, it is owned (to the extent that profit can be made) only by those holding knowledge of it.

Reintroducing the concept of ownership (although highly modified) to FOSS development can help explain the interrelatedness of significant FOSS projects with private companies, even though prominent authors in this area consider the movement as incompatible with capitalism (Coleman, 2013; Himanen et al., 2001; Söderberg, 2008). Indeed, the FOSS movement is at odds with classical capitalist values like the duty based work ethic or the legal form of private property rights. However, this does not seem to matter all that much as long as there is some source of differentiating value that can be utilized to build business models around. The barrier of information complexity and the necessary investments to appropriate the information represent such source. Its existence allows for establishment of contexts where decoupling of differentiating values from dutiful work or

158 The extent of the redefinition becomes visible, when we compare this case with a classical model of ownership such as, for example, the one that Bruce Carruthers and Laura Ariovich use as a starting point for their overview work on the sociology of property rights (Carruthers & Ariovich, 2004, p. 24). In the model, ownership is defined by a simultaneous validity of three points: (1) A has the right to use P; (2) A may exclude others from using P; (3) A may transfer rights defined by rules 1 and 2 to others by consent. The mismatch between the model and the empirical case at hand would be worth examining in detail and, as such, represents a venue for following up on this research. But within the scope of this work, I must limit myself only to pointing out that the sources of this problematical relationship seem to lie in an ambiguity of the word “use” (in this case stemming from the difference between use by users and a more knowledge intensive use by software developers), the assumed exclusivity in use of property (which is problematic for the whole domain where digital data are concerned, as I can illustrate with the work of Majid Yar on piracy, particularly the part where he stresses the difference between tangibles and intangibles (Yar, 2008, p. 612–613)), and in the unforeseen possibility, that rights (1) and (2) could be systematically suspended through a consistent use of a specific type of licensing.

private property rights is possible. It is not very important whether we view this as the result of an adaptive capacity of capitalism, or of a transformative capacity of the FOSS movement. What matters is the existence of a symbiotic (although fragile at times) balance between two entities that were initially considered inconsistent.

At its face value, the claim about the central role of knowledge in this environment seems to support the assumption of utopian virtualism, that knowledge is currently the most important production force. However, as we have seen, knowledge does not stand on its own even in the digital realm and so there are too many caveats to the claim. In FOSS projects, knowledge production is made possible by free access to information granted by specific type of licensing. Furthermore, knowledge production is taking place through involvement in an intricate network of tools and platforms, with investments of significant amounts of time, allowing for epistemic action to be conducted continuously in order for a newcomer to become part of a software development project. The tools and platforms are conditioned, in turn, by hardware such as servers, personal computers and their connection to the Internet.[159] The amounts of available time are dependent on the life situations of the respective actors. Combined with appropriate motivational impulse (such as intrinsic interest, status, moral or ideological positions), only this configuration results in accumulation of significant amounts of knowledge. It should become clear now, that for all its significance, knowledge is not self-sufficient, knowledge does not immaterially operate upon knowledge.

Furthermore, in utopian virtualism the implicit concept of immaterial action is an assumption enabling the images of free and flexible association.[160] It is based on the images of self-organizing masses, flocking, swarming around problems to solve them, to push the advance further. However, the idea, held by utopian virtualism, that taking advantage of the functionality that digital technologies bring is only a matter of having these technologies available omits significant contingencies. As we can see on how FOSS projects, the avant-garde of digital culture, operate, the interlocking of various conditions can not be assumed unproblematic. The dropout rate of newcomers demonstrates this very well. As my findings indicate, even in the supposedly frictionless digital environment, the center – periphery structure emerges, signifying the central role of few heavily involved individuals. This is also

159 The tables or basements filled with digital equipment that were recorded on video in the series on Linux kernel developer workspaces by Linux.com are cases in point. See, for example: https://www.youtube.com/watch?v=HSgUPqygAww or https://www.youtube.com/watch?v=NomqUIC_Uzs.

160 Which could also be related to Castells' claim about mind being a direct productive force – nothing is more flexible than a mind with unrestricted access to the reality.

supported by other research uncovering participation inequality (Holtgrewe, 2004; Krishnamurthy, 2002; Kuk, 2006; McInerney, 2009). This phenomenon may be explained by taking into consideration a combination of specific interests an individual has and a heavy knowledge investment one has to make in order to be able to contribute to a project. The core maintainers are the bearers of the deepest knowledge about a project, their association with it is anything but loose. Switching to another project would deprive them of their status and place them under the pressure of learning how to deal with a new codebase.[161] Thus, paradoxically enough, knowledge can be a limiting condition with regard to the images of utopian virtualism just as it seems to be enabling.

As represented by the claim that “for the first time in history, the human mind is a direct productive force, not just a decisive element of the production system” (Castells, 2010a, p. 31), the tendency to invoke the images of frictionless association seem to be implied by Castells’ work. But how can the mind be a direct productive force? All the “knowledge workers” who are entrenched with computing technology through their working hours always interact with a software interface. But when it gets to its users, an interface is no longer a fluid thing that can be meddled with by modifying text (source code) as was the case while it was developed. In interaction with users, an interface is part of a compiled program, having a binary form and operating closer to the logic of voltage differences transmitted by hardware, than to the logic of (programming) language and text (source code). Therefore, when software reaches its users, we can see that the culturally contingent construct has materialized (literally) into a solid thing that (together with hardware input/output devices) forms the interface for the mind.

It is said that a good interface should be invisible (in the sense that it does not get in the way of user’s actions), but that does not mean it is not there. As we have seen, interfaces are there to display data (because digital information is not directly accessible to the senses, by definition, it needs an intermediary to be accessed) and to offer to the user a set of possible actions. Anything that is not part of an interface, is impossible to perform for a user.[162] This dependency on interface is not consistent with the claim of mind being a *direct* productive

161 However, it is not uncommon for such switching to take place. This seems to indicate that involvement in FOSS projects, in a sense, constitutes the kind of education that Castells envisions to produce “self-programmable labor” (Castells, 2010c, p. 377).

162 In this regard, Lawrence Lessig famously claims that source code is the law of cyberspace (Lessig, 2006, p. 5) while Richard Spinello builds upon this analogy to argue that software developers should aspire to a similar moral competence to that of lawmakers in a democratic establishment (Spinello, 2001, p. 149).

force. Castells ascribes epistemic credit only to human mind[163], but as we can see, there is still a production system revolving around a mind, mediating its input and output.

To avoid this criticism, one could argue that the Castells' statement should not be considered with relation to an individual worker, but with regard to the minds of all concerned workers. That is to say, that the mind of one is operating with something that is a product of another's mind, but it is still a product of a mind. Therefore, with regard to the *type* of production (production by mind) the statement could be theoretically correct. But when we consider the argument formulated in this way, that in production, one mind builds upon what other minds created, it loses its claim to discontinuity. Was production, with its use of tools designed by a narrow group of people and utilized by a larger one, not organized in this way before the 1970s? It certainly was.

I do not aim to disprove, or argue against Castells' work as a whole. A proper analysis of his comprehensive work is beyond the scope of this text. My argument is centered around just one of his claims that I see as being symptomatic of utopian virtualism. In the light of my findings, Castells' premise about the mind being a direct productive force seems to be untenable. This is so because, in the sense elaborated above, he omits the problematic, mediating role of at least some of the technologies he writes about and because he ascribes cognitive accomplishments solely to a universalistic model of a human mind. What my findings, connecting to previous developments in Actor-Network Theory and the theory of distributed cognition, demonstrate, is that by itself, the human mind (if we can at all talk about such singularity) can achieve very little. How could knowledge exist without all the elements that together form a (digital) interface? This is not to argue that the impulses of the mind have to be embodied in a material form. I point to the thesis that material objects (including the digital ones) are constitutive of cognitive processes. Therefore, the mind is surely a part of the production process, but it is not the sole, nor direct productive force.

163 According to Edwin Hutchins, the image of human mind as the sole origin of cognitive accomplishment is the result of a reified analogy between the human mind and the computer: “The computer was not made in the image of the person. The computer was made in the image of the formal manipulations of abstract symbols. And the last 30 years of cognitive science can be seen as attempts to remake the person in the image of the computer.” (Hutchins, 1995, p. 363).

References

Aigrain, P. (1997). Attention, media, value and economics. *First Monday*, *2*(9).

Ashby, W. R. (1962). Principles of the self-organizing system. In H. V. Foerster & G. W. Zopf (Eds.), *Principles of Self-organization* (pp. 255–278). Oxford: Pergamon Press.

Atkinson, P., & Coffey, A. (2004). Analysing documentary realities. In D. Silverman (Ed.), *Qualitative research: Theory, method and practice* (pp. 56–75). London: Sage.

Barad, K. (1998). Getting real: Technoscientific practices and the materialization of reality. *Differences, A Journal of Feminist Cultural Studies*, *10*(2), 87–126.

Barad, K. (2007). *Meeting the Universe Halfway: Quantum Physics and the Entanglement of Matter and Meaning*. Durham: Duke university Press.

Baszanger, I., & Dodier, N. (2004). Ethnography: relating the part to the whole. In D. Silverman (Ed.), *Qualitative research: Theory, method and practice* (pp. 9–34). London: Sage.

Bateson, G. (1972). *Steps to an ecology of mind: Collected essays in anthropology, psychiatry, evolution, and epistemology*. Chicago: University of Chicago Press.

Benkler, Y. (2002). Coase’s Penguin, or, Linux and “The Nature of the Firm”. *Yale Law Journal*, *112*(3), 369–446.

Benkler, Y. (2004). Sharing nicely: On shareable goods and the emergence of sharing as a modality of economic production. *Yale Law Journal*, *114*(2), 273–358.

Benkler, Y. (2006). *The wealth of networks: How social production transforms markets and freedom*. New Haven: Yale University Press.

Bergquist, M., & Ljungberg, J. (2001). The power of gifts: organizing social relationships in open source communities. *Information Systems Journal*, *11*(4), 305–320.

Blumer, H. (1986). *Symbolic interactionism: Perspective and method*. Englewood Cliffs: Prentice Hall, Inc.

Bourdieu, P. (1998). *Practical reason: On the theory of action*. Stanford, CA: Stanford University Press.

Brewer, J. (2000). *Ethnography*. Buckingham: Open University Press.

Brooks, F. (1995). *The mythical man-month: Essays on software engineering*. Boston: Addison Wesley.

Buckley, W. (1967). *Sociology and modern systems theory.* Englewood Cliffs: Prentice-Hall.

Cantoni, L., & Tardini, S. (2006). *Internet*. New York: Routledge.

Carruthers, B. G., & Ariovich, L. (2004). The sociology of property rights. *Annual Review of Sociology*, *30*(1), 23–46.

Castells, M. (2010a). *The rise of the network society. The information age: Economy, society and culture* (Vol. 1). Chichester: Wiley–Blackwell.

Castells, M. (2010b). *The power of identity. The information age: Economy, society, and culture* (Vol. 2). Chichester: Wiley–Blackwell.

Castells, M. (2010c). *End of Millennium. The Information Age: Economy, Society, and Culture* (Vol. 3). Chichester: Wiley–Blackwell.

Castells, M., & Portes, A. (1989). World underneath: The origins, dynamics, and effects of the informal economy. In A. Portes, M. Castells, & L. A. Benton (Eds.), *The informal economy: Studies in advanced and less developed countries*. Baltimore: The Johns Hopkins University Press.

Cetina, K. K. (1999). *Epistemic cultures: How the sciences make knowledge*. Cambridge, MA: Harvard University Press.

Clark, A. (1998). Where brain, body, and world collide. *Daedalus*, *127*(2), 257–280.

Clark, A. (2006). Material symbols. *Philosophical Psychology*, *19*(3), 291–307.

Clark, A., & Chalmers, D. (1998). The extended mind. *Analysis*, *58*(1), 7–19.

Coase, R. H. (1937). The nature of the firm. *Economica*, *4*(16), 386–405.

Coase, R. H. (1960). The Problem of Social Cost. *Journal of Law and Economics*, *3*(1).

Coleman, G. (2009). Code is speech: Legal tinkering, expertise, and protest among free and open source software developers. *Cultural Anthropology*, *24*(3), 420–454.

Coleman, G. (2010). The hacker conference: A ritual condensation and celebration of a lifeworld. *Anthropological Quarterly*, *83*(1), 47–72.

Coleman, G. (2013). *Coding freedom: The ethics and aesthetics of hacking*. Princeton: Princeton University Press.

Cook, J., Laidlaw, J., & Mair, J. (2009). What if there is no elephant? Towards a conception of an un-sited field. In M. A. Falzon (Ed.), *Multi-sited ethnography: Theory, praxis and locality in contemporary research* (pp. 47–72). Farnham: Ashgate Publishing, Ltd.

Coole, D., Frost, S., Bennett, J., Cheah, P., Orlie, M. A., & Grosz, E. (2010). *New materialisms: Ontology, agency, and politics*. Durham: Duke University Press.

Corbet, J., Kroah-Hartman, G., & McPherson, A. (2015). Linux kernel development. Annual report, The Linux Foundation.

Crowston, K., Li, Q., Wei, K., Eseryel, U. Y., & Howison, J. (2007). Self-organization of teams for free/libre open source software development. *Information and Software Technology*, *49*(6), 564–575.

Dafermos, G., & Söderberg, J. (2009). The hacker movement as a continuation of labour struggle. *Capital & Class*, *33*(1), 53–73.

Dahlander, L., & Magnusson, M. (2008). How do firms make use of open source communities? *Long Range Planning*, *41*(6), 629–649.

Darking, M., & Whitley, E. A. (2007). Towards an understanding of FLOSS: Infrastructures, materiality and the digital business ecosystem. *Science Studies*, *20*(2), 13–33.

Davis, M. (2013). Doing Research “on and Through” New Media Narrative. In M. Andrews, M. Tamboukou, & C. Squire (Eds.), *Doing Narrative Research* (pp. 159–175). London: Sage.

Demazière, D., Horn, F., & Zune, M. (2007). The Functioning of a Free Software Community. *Science Studies*, *20*(2), 34–54.

DiMaggio, P. (1997). Culture and cognition. *Annual Review of Sociology*, *23*(1), 263–287.

Dittrich, Y. (2002). Reaching out for Commitments: Systems Development as Networking. In Y. Dittrich, C. Floyd, & R. Klischewski (Eds.), *Social Thinking – Software Practice* (pp. 243–262). Cambridge, MA: MIT Press.

Ducheneaut, N. (2005). Socialization in an open source software community: A socio-technical analysis. *Computer Supported Cooperative Work (CSCW)*, *14*(4), 323–368.

Ehn, P. (1988). *Work-oriented design of computer artifacts* (PhD thesis). Umeå University, Stockholm.

Fairclough, N. (1992). *Discourse and social change*. Cambridge: Polity Press.

Falzon, M. A. (2012). *Multi-sited ethnography: theory, praxis and locality in contemporary research*. Farnham: Ashgate Publishing, Ltd.

Fekete, L. (2006). The ethics of economic interactions in the network economy. *Information, Community & Society*, *9*(6), 737–760.

Fetterman, D. M. (2010). *Ethnography: Step by step*. Thousand Oaks, CA: Sage.

Floyd, C. (1992). Human questions in computer science. In C. Floyd, H. Züllighoven, R. Budde, & R. Keil-Slawik (Eds.), *Software development and reality construction* (pp. 15–27). Berlin: Springer.

Freeman, S. (2007). The material and social dynamics of motivation. *Science Studies*, *20*(2), 55–77.

Garfinkel, H. (1967). *Studies in ethnomethodology*. Englewood Cliffs: Prentice-Hall.

Garzarelli, G., & Fontanella, R. (2011). Open source software production, spontaneous input, and organizational learning. *American Journal of Economics and Sociology*, *70*(4), 928–950.

Geertz, C. (1973). *The interpretation of cultures: Selected essays*. New York: Basic Books.

Geertz, C. (1996). *After the fact*. Cambridge, MA: Harvard University Press.

Ghosh, R. A., Glott, R., Krieger, B., & Robles, G. (2002). Free/libre and open source software: Survey and study. Maastricht: International institute of Infonomics. Retrieved from http://www.flossproject.org/report/FLOSS_Final4.pdf.

Giere, R. N. (2002). Discussion note: Distributed cognition in epistemic cultures. *Philosophy of Science*, *69*(4), 637–644.

Giere, R. N., & Moffatt, B. (2003). Distributed cognition: Where the cognitive and the social merge. *Social Studies of Science*, *33*(2), 301–310.

Glanville, R. (1982). Inside every white box there are two black boxes trying to get out. *Behavioral Science*, *27*(1), 1–11.

Glanville, R. (2007). A (Cybernetic) Musing: Ashby and the Black Box. *Cybernetics & Human Knowing*, *14*(2–3), 189–196.

Goldhaber, M. (2006). The value of openness in an attention economy. *First Monday*, *11*(6).

Goldhaber, M. H. (1997). The attention economy and the net. *First Monday*, *2*(4).

Greschke, H. (2007). Logging into the Field—Methodological Reflections on Ethnographic Research in a Pluri-Local and Computer-Mediated Field. *Forum Qualitative Sozialforschung / Forum: Qualitative Social Research*, *8*(3). Retrieved from http://www.qualitative-research.net/index.php/fqs/article/view/279.

Hakken, D. (1999). *Cyborgs@cyberspace?: An ethnographer looks to the future*. New York: Routledge.

Hakken, D. (2003). *The knowledge landscapes of cyberspace*. New York: Routledge.

Hammersley, M. (1990). *Dilemma of Qualitative Method*. New York: Routledge.

Haraway, D. (2006). A Cyborg Manifesto: Science, Technology, and Socialist-Feminism in the Late 20th Century. In J. Weiss, J. Nolan, J. Hunsinger, & P. Trifonas (Eds.), *The International Handbook of Virtual Learning Environments* (pp. 117–158). New York: Springer.

Hemetsberger, A., & Reinhardt, C. (2006). Learning and knowledge-building in open-source communities a social-experiential approach. *Management Learning*, *37*(2), 187–214.

Hemetsberger, A., & Reinhardt, C. (2009). Collective development in open-source communities: An activity theoretical perspective on successful online collaboration. *Organization Studies*, *30*(9), 987–1008.

Heylighen, F. (1999). Collective Intelligence and its Implementation on the Web: algorithms to develop a collective mental map. *Computational & Mathematical Organization Theory*, *5*(3), 253–280.

Heylighen, F. (2002). The global brain as a new utopia. In R. Maresch & F. Rötzer (Eds.), *Zukunftsfiguren*. Frankfurt: Suhrkamp.

Heylighen, F., & Bollen, J. (1996). The World-Wide Web as a Super-Brain: from metaphor to model. In R. Trappl (Ed.), *Cybernetics and Systems*. Vienna: Austrian Society for Cybernetics.

Heylighen, F., Heath, M., & Van, F. (2004). The Emergence of Distributed Cognition: a conceptual framework. In *Proceedings of collective intentionality IV*. Siena.

Himanen, P., Castells, M., & Torvalds, L. (2001). *The Hacker Ethic and the Spirit of the Information Age*. New York: Random House.

Hine, C. (2000). *Virtual ethnography*. London: Sage.

Holtgrewe, U. (2004). Articulating the Speed (s) of the Internet The Case of Open Source/Free Software. *Time & Society*, *13*(1), 129–146.

Hutchins, E. (1991). Organizing work by adaptation. *Organization Science, 2*(1), 14–39.

Hutchins, E. (1995). *Cognition in the Wild*. Cambridge, MA: MIT press.

Hutchins, E. (2011). Enculturating the supersized mind. *Philosophical Studies*, *152*(3), 437–446.

Hutchins, E. L., Hollan, J. D., & Norman, D. A. (1985). Direct manipulation interfaces. *Human–Computer Interaction*, *1*(4), 311–338.

Jansen, K., & Vellema, S. (2011). What is technography? *NJAS-Wageningen Journal of Life Sciences*, *57*(3), 169–177.

Karatzogianni, A., & Michaelides, G. (2009). Cyberconflict at the edge of chaos: Cryptohierarchies and self-organisation in the open-source movement. *Capital & Class*, *33*(1), 143–157.

Keil-Slawik, R. (1992). Artifacts in software design. In C. Floyd, H. Züllighoven, R. Budde, & R. Keil-Slawik (Eds.), *Software development and reality construction* (pp. 168–188). Berlin: Springer.

Kelty, C. (2004). Culture's open sources: Software, copyright, and cultural critique. *Anthropological Quarterly*, *77*(3), 499–506.

Kelty, C. (2008). *Two bits: The cultural significance of free software*. Durham: Duke University Press.

Kim, M. (2007). The Creative Commons and copyright protection in the digital era: Uses of Creative Commons licenses. *Journal of Computer-Mediated Communication*, *13*(1), 187–209.

Klischewski, R. (2002). Reaching out for Commitments: Systems Development as Networking. In Y. Dittrich, C. Floyd, & R. Klischewski (Eds.), *Social Thinking – Software Practice* (pp. 309–329). Cambridge, MA: MIT Press.

Krishnamurthy, S. (2002). Cave or community?: An empirical examination of 100 mature open source projects. *First Monday*, *7*(6).

Kuk, G. (2006). Strategic interaction and knowledge sharing in the KDE developer mailing list. *Management Science*, *52*(7), 1031–1042.

Lakhani, K., Wolf, B., Bates, J., & DiBona, C. (2002). The boston consulting group hacker survey. Retrieved from ftp://mirror.linux.org.au/pub/linux.conf.au/2003/papers/Hemos/Hemos.pdf.

Landström, C., Whatmore, S. J., & Lane, S. N. (2011). Virtual Engineering. *Science Studies*, *24*(2), 3–22.

Latour, B. (1986). Visualization and cognition. *Knowledge and Society*, *6*(1), 1–40.

Latour, B. (1991). Technology is society made durable. In J. Law (Ed.), *A Sociology of Monsters: Essays on Power, Technology and Domination* (pp. 103–132). London: Routledge.

Latour, B. (1994). On technical mediation. *Common Knowledge*, *3*(2), 29–64.

Latour, B. (1996). On interobjectivity. *Mind, Culture, and Activity*, *3*(4), 228–245.

Latour, B. (1996). Social theory and the study of computerized work sites. In W. J. Orlinokowski & W. Geoff (Eds.), *Information technology and changes in organizational work* (pp. 295–307). London: Chapman & Hall.

Latour, B. (2000). When things strike back: a possible contribution of “science studies” to the social sciences. *The British Journal of Sociology*, *51*(1), 107–123.

Latour, B. (2003). The Promises of Constructivism. *Chasing Technoscience: Matrix for Materiality*, 27–46.

Latour, B. (2005). *Reassembling the social: An introduction to actor-network-theory*. New York: Oxford University Press.

Latour, B. (2007a). A Textbook Case Revisited. Knowledge as mode of existence. In E. J. Hackett, O. Amsterdamska, M. E. Lynch, & J. Wajcman (Eds.), *The Handbook of Science and Technology Studies-Third Edition* (pp. 83–112). Cambridge, MA: MIT Press.

Latour, B. (2007b). Can we get our materialism back, please? *Isis*, *98*(1), 138–142.

Latour, B. (2009). Spheres and networks: two ways to reinterpret globalization. *Harvard Design Magazine, 30*(Spring/Summer), 138–144.

Latour, B. (2010, February). Networks, societies, spheres: Reflections of an actor-network theorist. Lecture notes.

Latour, B. (2011). "What's the story?" Organizing as a mode of existence. In J.-H. Passoth, B. Peuker, & M. Schillmeier (Eds.), *Agency without Actors? New Approaches to Collective Action* (pp. 164–167). London: Routledge.

Latour, B. (2012). *We have never been modern*. Harvard University Press.

Latour, B., & Woolgar, S. (2013). *Laboratory life: The construction of scientific facts*. Princeton: Princeton University Press.

Lave, J. (1988). *Cognition in practice: Mind, mathematics and culture in everyday life*. Cambridge: Cambridge University Press.

Law, J., & Lien, M. E. (2013). Slippery: Field notes in empirical ontology. *Social Studies of Science*, *43*(3), 363–378.

Lawson, M. P. (1999). The Holy Spirit as conscience collective. *Sociology of Religion*, *60*(4), 341–361.

Lee, G. K., & Cole, R. E. (2003). From a firm-based to a community-based model of knowledge creation: The case of the Linux kernel development. *Organization Science*, *14*(6), 633–649.

Lerner, J., & Tirole, J. (2002). Some simple economics of open source. *The Journal of Industrial Economics*, *50*(2), 197–234.

Lessig, L. (2006). *Code: version 2.0*. New York: Basic Books.

Lévy, P. (2005). Collective intelligence, a civilisation: towards a method of positive interpretation. *International Journal of Politics, Culture, and Society*, *18*(3-4), 189–198.

Lévy, P., & Bonomo, R. (1999). *Collective intelligence: Mankind's emerging world in cyberspace*. Perseus Publishing.

Leydesdorff, L. (2011). "Meaning" as a sociological concept: A review of the modeling, mapping and simulation of the communication of knowledge and meaning. *Social Science Information*, *50*(3-4), 391–413.

Ljungberg, J. (2000). Open source movements as a model for organising. *European Journal of Information Systems*, *9*(4), 208–216.

Long, E. (1993). Textual interpretation as collective action. *The Ethnography of Reading*, 180–211.

Luhmann, N. (1995). *Social systems*. Stanford, CA: Stanford University Press.

Luhmann, N. (2014). *Die Gesellschaft der Gesellschaft*. Frankfurt: Suhrkamp.

Luyt, B. (2011). The nature of historical representation on Wikipedia: Dominant or alterative historiography? *Journal of the American Society for Information Science and Technology*, *62*(6), 1058–1065.

Macek, J. (2011). *Poznámky ke studiím nových médií* (PhD thesis). Masarykova univerzita, Brno.

Magnani, L., & Bardone, E. (2008). Distributed morality: externalizing ethical knowledge in technological artifacts. *Foundations of Science*, *13*(1), 99–108.

Magnus, P. (2007). Distributed cognition and the task of science. *Social Studies of Science*, *37*(2), 297–310.

Malone, T., Laubacher, R., & Dellarocas, C. (2010). The collective intelligence genome. *MIT Sloan Management Review*, *51*(3), 21–31.

Marcus, G. E. (1995). Ethnography in/of the world system: the emergence of multi-sited ethnography. *Annual Review of Anthropology*, *24*(1), 95–117.

Markham, A. N. (2004). Internet communication as a tool for qualitative research. *Qualitative Research: Theory, Method and Practice*, 95–124.

Maturana, H. R. (1980). *Autopoiesis and cognition: The realization of the living*. Dodrecht: D. Reidl Publishing Company.

Maturana, H. R., & Varela, F. J. (1987). *The tree of knowledge: The biological roots of human understanding.* Boston, MA: Shambhala Publications, Inc.

McCarthy, D. (1996). *Knowledge as culture: The new sociology of knowledge*. London: Routledge.

McInerney, P.-B. (2009). Technology movements and the politics of free/open source software. *Science, Technology & Human Values*, *34*(2), 206–233.

Mead, G. H. (1972). *Mind, self, and society: From the standpoint of a social behaviorist*. Chicago: University of Chicago press.

Miles, M., & Huberman, M. (1994). *Qualitative data analysis: An expanded sourcebook*. Thousand Oaks, CA: Sage.

Mol, A., & Law, J. (1994). Regions, networks and fluids: anaemia and social topology. *Social Studies of Science*, *24*(4), 641–671.

Nardi, B. A., & Engeström, Y. (1999). A web on the wind: The structure of invisible work. *Computer Supported Cooperative Work (CSCW)*, *8*(1), 1–8.

Naur, P. (1985). Programming as theory building. *Microprocessing and Microprogramming*, *15*(5), 253–261.

Neyland, D. (2008). *Organizational ethnography*. Thousand Oaks, CA: Sage.

Nimmo, R. (2011). Actor-network theory and methodology: Social research in a more-than-human world. *Methodological Innovations Online*, *6*(3), 108–119.

Nørbjerg, J., & Kraft, P. (2002). Software Practice Is Social Practice. In Y. Dittrich, C. Floyd, & R. Klischewski (Eds.), *Social Thinking – Software Practice* (pp. 205–222). Cambridge, MA: MIT Press.

Norris, P. (2001). *Digital divide: Civic engagement, information poverty, and the Internet worldwide*. Cambridge: Cambridge University Press.

Nørskov, S., & Rask, M. (2011). Observation of online communities: A discussion of online and offline observer roles in studying development, cooperation and coordination in an open source software environment. *Forum Qualitative Sozialforschung/Forum: Qualitative Social Research*, *12*(3).

O'Mahony, S., & Ferraro, F. (2007). The emergence of governance in an open source community. *Academy of Management Journal*, *50*(5), 1079–1106.

Ondrejka, C. (2004). Escaping the gilded cage: User created content and building the metaverse. *New York Law School Law Journal*, *49*(1), 81–101.

Osterloh, M., & Rota, S. (2004). Trust and community in open source software production. *Analyse & Kritik*, *26*(1), 279–301.

Packer, M. (2011). *The science of qualitative research*. Cambridge: Cambridge University Press.

Parikka, J. (2014, September). Digital Culture as the Desire of the Geophysical: A Geology of Media. Conference keynote.

Prior, L. (2004). Doing things with documents. In D. Silverman (Ed.), *Qualitative research: Theory, method and practice* (pp. 76–94). London: Sage.

Qureshi, I., & Fang, Y. (2010). Socialization in open source software projects: A growth mixture modeling approach. *Organizational Research Methods*, *14*(1), 208–238.

Raeithel, A. (1992). Activity theory as a foundation for design. In C. Floyd, H. Züllighoven, R. Budde, & R. Keil-Slawik (Eds.), *Software development and reality construction* (pp. 391–415). Berlin: Springer.

Ratto, M. (2007). A Practice-Based Model of Access for Science. *Science Studies*, *20*(1), 73–105.

Raymond, E. (1999). The cathedral and the bazaar. *Knowledge, Technology & Policy*, *12*(3), 23–49.

Raymond, E. (2003). *The art of Unix programming*. Boston, MA: Addison-Wesley Professional.

Reay, M. (2010). Knowledge Distribution, Embodiment, and Insulation. *Sociological Theory*, *28*(1), 91–107.

Reimer, K. (2005). Fiat Lux: Religion as Distributed Cognition. *Journal of Psychology & Christianity*, *24*(2), 130–139.

Rogers, Y., & Ellis, J. (1994). Distributed cognition: an alternative framework for analysing and explaining collaborative working. *Journal of Information Technology*, *9*(1), 119–128.

Rönkkö, K. (2002). "Yes-What Does That Mean?" Understanding Distributed Requirements Handling. In Y. Dittrich, C. Floyd, & R. Klischewski (Eds.), *Social Thinking – Software Practice* (pp. 223–241). Cambridge, MA: MIT Press.

Rosenzweig, R. (2006). Can History Be Open Source? Wikipedia and the Future of the Past. *The Journal of American History*, *93*(1), 117–146.

Rybas, N., & Gajjala, R. (2007). Developing Cyberethnographic Research Methods for Understanding Digitally Mediated Identities. *Forum Qualitative Sozialforschung / Forum: Qualitative Social Research*, *8*(3). Retrieved from http://www.qualitative-research.net/index.php/fqs/article/view/282.

Saldana, J. (2009). *The coding manual for qualitative researchers*. London: Sage.

Shah, S. K. (2006). Motivation, governance, and the viability of hybrid forms in open source software development. *Management Science*, *52*(7), 1000–1014.

Shields, R. (2003). *The virtual*. London: Routledge.

Shiga, J. (2007). Copy-and-persist: The logic of mash-up culture. *Critical Studies in Media Communication*, *24*(2), 93–114.

Schütz, A. (1953). Common-Sense and Scientific Interpretation of Human Action. *Philosophy and Phenomenological Research*, *14*(1), 1–38.

Simon, H. A. (1971). Designing organizations for an information-rich world. In M. Greenberger (Ed.), *Computers, communications, and the public interest* (pp. 37–72). Baltimore: Johns Hopkins Press.

Söderberg, J. (2008). *Hacking Capitalism*. New York: Routledge.

Spinello, R. A. (2001). Code and moral values in cyberspace. *Ethics and Information Technology*, *3*(2), 137–150.

Star, S. L. (1999). The ethnography of infrastructure. *American Behavioral Scientist*, *43*(3), 377–391.

Star, S. L. (2002). Infrastructure and ethnographic practice: Working on the fringes. *Scandinavian Journal of Information Systems*, *14*(2), 6.

Star, S. L., & Ruhleder, K. (1996). Steps toward an ecology of infrastructure: Design and access for large information spaces. *Information Systems Research*, *7*(1), 111–134.

Stehr, N., & Ufer, U. (2009). On the global distribution and dissemination of knowledge. *International Social Science Journal*, *60*(195), 7–24.

Stewart, D. (2005). Social status in an open-source community. *American Sociological Review*, *70*(5), 823–842.

Stewart, K., & Gosain, S. (2006). The impact of ideology on effectiveness in open source software development teams. *Mis Quarterly*, *30*(2), 291–314.

Strathern, M. (2002). Abstraction and decontextualization: An anthropological comment. In S. Woolgar (Ed.), *Virtual society* (pp. 302–314). Oxford: Oxford University Press.

Suchman, L. (2007). *Human-machine reconfigurations: Plans and situated actions*. Cambridge: Cambridge University Press.

Suchman, L. A. (1987). *Plans and situated actions: the problem of human-machine communication*. Cambridge: Cambridge university press.

Teli, M., Pisanu, F., & Hakken, D. (2007). The Internet as a Library-Of-People: For a Cyberethnography of Online Groups. *Forum Qualitative Sozialforschung / Forum: Qualitative Social Research*, *8*(3). Retrieved from http://www.qualitative-research.net/index.php/fqs/article/view/283.

Thurk, J., & Fine, G. A. (2003). The Problem of Tools: Technology and the Sharing of Knowledge. *Acta Sociologica*, *46*(2), 107–117.

Tovey, M. (2008). *Collective Intelligence: Creating a Prosperous World at Peace*. Oakton: EIN Press.

Uspenski, I. (2013). Mass intelligence and the commoditized reader. In P. Zahrádka & R. Sedláková (Eds.), *New Perspectives on Consumer Culture Theory and Research*. Newcastle upon Tyne: Cambridge scholars publishing.

Vinck, D., & Blanco, E. (2003). *Everyday engineering: An ethnography of design and innovation*. Cambridge, MA: MIT Press.

Von Foerster, H. (2003). Ethics and second-order cybernetics. In H. von Foerster (Ed.), *Understanding understanding* (pp. 287–304). New York: Springer.

Von Foerster, H. (2003). For Niklas Luhmann: How Recursive is Communication? In H. von Foerster (Ed.), *Understanding Understanding* (pp. 305–323). New York: Springer.

Von Foerster, H. (2003). On self-organizing systems and their environments. In H. von Foerster (Ed.), *Understanding Understanding* (pp. 1–19). New York: Springer.

Von Krogh, G., & Von Hippel, E. (2006). The promise of research on open source software. *Management Science*, *52*(7), 975–983.

Von Krogh, G., Spaeth, S., & Lakhani, K. R. (2003). Community, joining, and specialization in open source software innovation: a case study. *Research Policy*, *32*(7), 1217–1241.

Wagner, R. P. (2003). Information wants to be free: Intellectual property and the mythologies of control. *Columbia Law Review*, *103*(1), 995–1034.

Wall, S. (2015). Focused Ethnography: A Methodological Adaption for Social Research in Emerging Contexts. *Forum Qualitative Sozialforschung / Forum: Qualitative Social Research*, *16*(1). Retrieved from http://www.qualitative-research.net/index.php/fqs/article/view/2182.

Wan, P. Y.-z. (2011). *Reframing the social: Emergentist systemism and social theory*. Farnham: Ashgate Publishing, Ltd.

Wechsler, D. (1971). Concept of collective intelligence. *American Psychologist*, *26*(10), 904.

West, J., & O'Mahony, S. (2008). The role of participation architecture in growing sponsored open source communities. *Industry and Innovation*, *15*(2), 145–168.

Westrup, C. (2002). On Retrieving Skilled Practices: The Contribution of Ethnography to Software Development. In Y. Dittrich, C. Floyd, & R. Klischewski (Eds.), *Social Thinking – Software Practice* (pp. 95–110). Cambridge, MA: MIT Press.

Wikipedia. (2015). Wikipedia — Wikipedia, The Free Encyclopedia. Retrieved from http://en.wikipedia.org/w/index.php?title=Wikipedia&oldid=642784283.

Wise, N. M. (2011). Collective intelligence and its corollaries. *History and Technology*, *27*(2), 197–203.

Woolgar, S. (2002). Five rules of virtuality. In S. Woolgar (Ed.), *Virtual society* (pp. 1–22). Oxford: Oxford University Press.

Wright, P. C., Fields, R. E., & Harrison, M. D. (2000). Analyzing human-computer interaction as distributed cognition: the resources model. *Human-Computer Interaction*, *15*(1), 1–41.

Yar, M. (2008). The rhetorics and myths of anti-piracy campaigns: criminalization, moral pedagogy and capitalist property relations in the classroom. *New Media & Society*, *10*(4), 605–623.

Zerubavel, E., & Smith, E. R. (2010). Transcending cognitive individualism. *Social Psychology Quarterly*, *73*(4), 321–325.

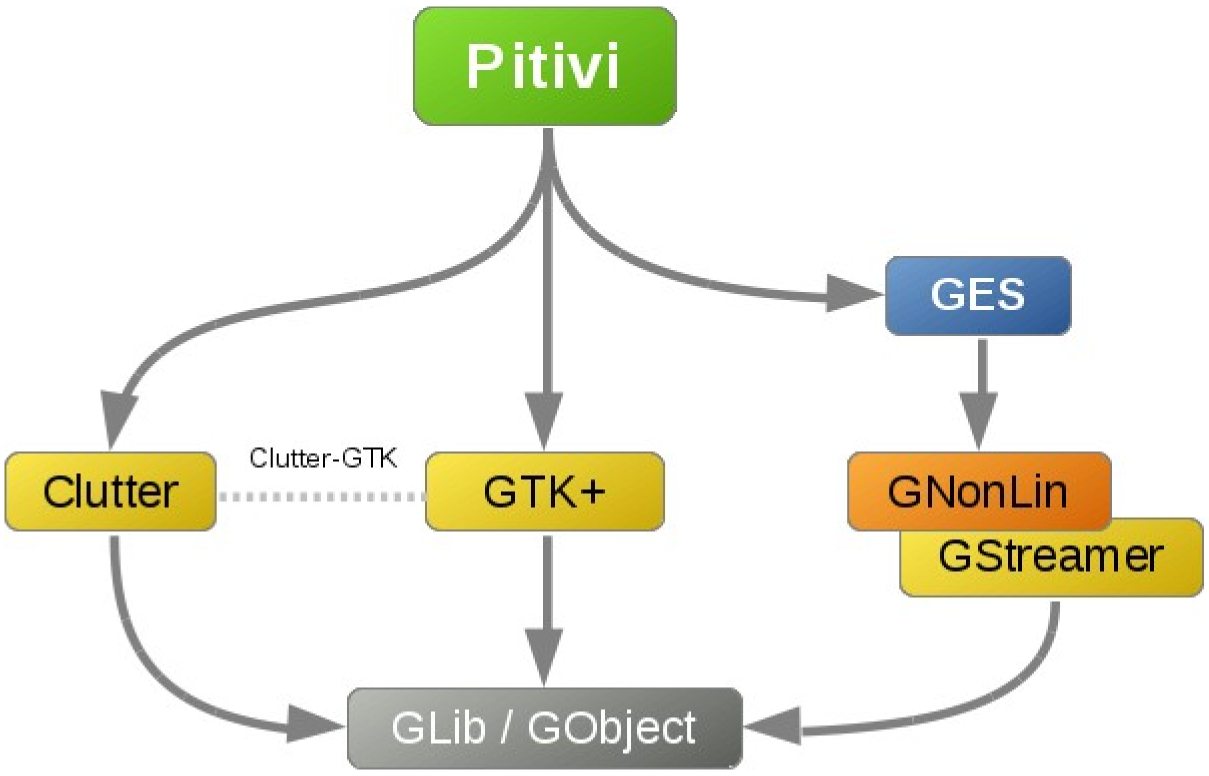
Appendix 1: An Example of Scripting

```
1   #!/bin/bash
2
3   echo -n "Please enter the absolute path and press [ENTER]: "
4   read -e file
5
6   sed -i "s/{\\\'a}/á/g" $file
7   sed -i "s/{\\\'A}/Á/g" $file
8   sed -i "s/{\\\'e}/é/g" $file
9   sed -i "s/{\\\'E}/É/g" $file
10  sed -i "s/{\\\'\\\i}/í/g" $file
11  sed -i "s/{\\\'\\\I}/Í/g" $file
12  sed -i "s/{\\\'o}/ó/g" $file
13  sed -i "s/{\\\'O}/Ó/g" $file
14  sed -i "s/{\\\'u}/ú/g" $file
15  sed -i "s/{\\\'U}/Ú/g" $file
16  sed -i "s/{\\\'y}/ý/g" $file
17  sed -i "s/{\\\'Y}/Ý/g" $file
18  sed -i "s/{\\\r{u}}/ů/g" $file
19  sed -i "s/{\\\r{U}}/Ů/g" $file
20  sed -i "s/{\\\v{c}}/č/g" $file
21  sed -i "s/{\\\v{C}}/Č/g" $file
22  sed -i "s/{\\\v{d}}/ď/g" $file
23  sed -i "s/{\\\v{D}}/Ď/g" $file
24  sed -i "s/{\\\v{e}}/ě/g" $file
25  sed -i "s/{\\\v{E}}/Ě/g" $file
26  sed -i "s/{\\\v{r}}/ř/g" $file
27  sed -i "s/{\\\v{R}}/Ř/g" $file
28  sed -i "s/{\\\v{s}}/š/g" $file
29  sed -i "s/{\\\v{S}}/Š/g" $file
30  sed -i "s/{\\\v{t}}/ť/g" $file
31  sed -i "s/{\\\v{T}}/Ť/g" $file
32  sed -i "s/{\\\v{z}}/ž/g" $file
33  sed -i "s/{\\\v{Z}}/Ž/g" $file
34
35  echo "Done"
```

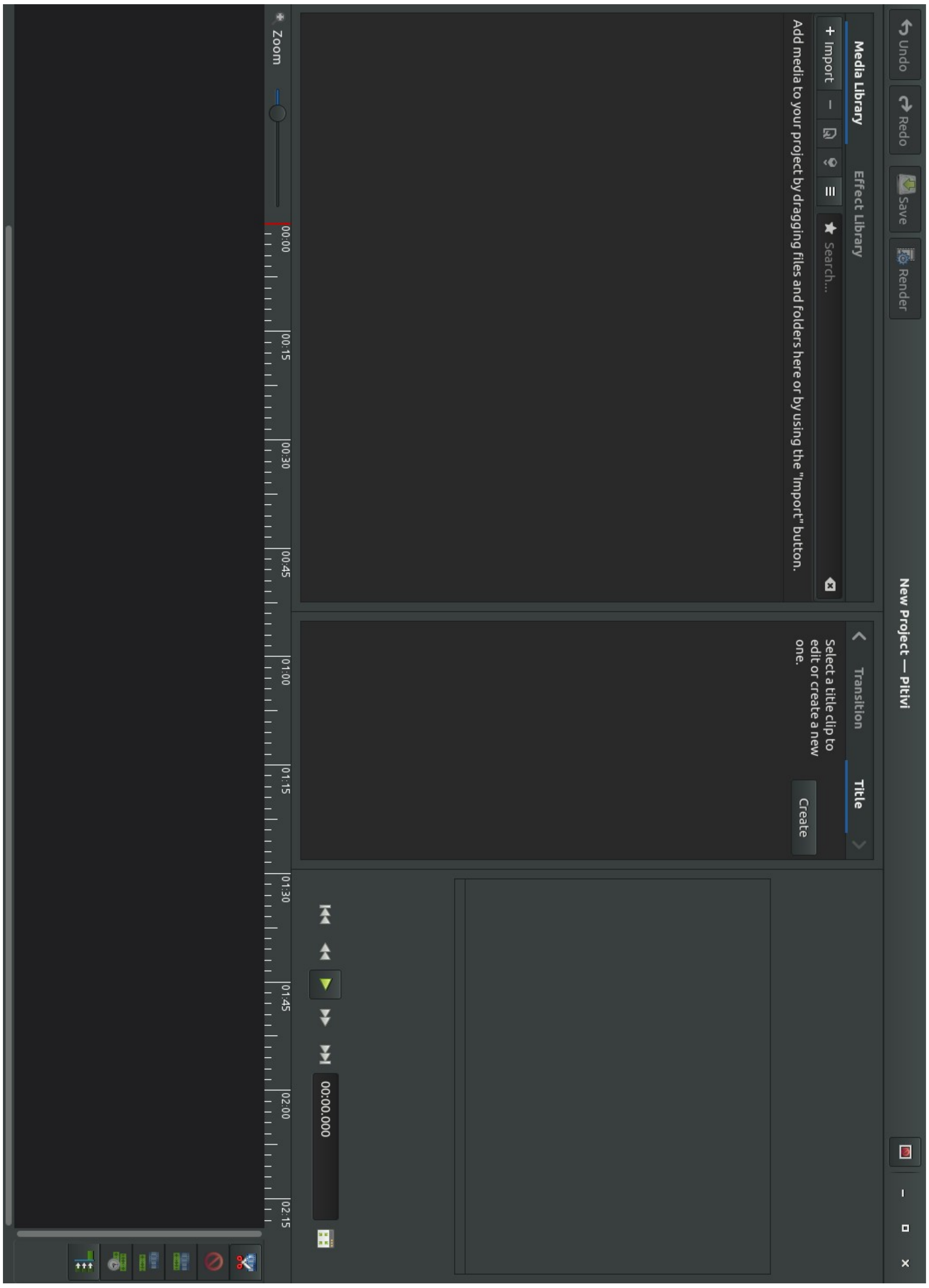
This is a script I put together to automate conversion of diacritics, as expressed in the syntax of the Latex markup language ({'a}) into its literal form (á). The first line indicates that the following contents of the file should be interpreted as a Bash script. Line 3 asks a user to input a path to the file, whose contents should be converted. Line 4 reads the path and saves it as a variable called "file". The following lines proceed uniformly: Sed, a program designed for automated editing of text files, is called with an argument "-i" which instructs it to save the edits directly into the file from which it reads (the default behavior is to only print the edited text into the terminal, not affecting the file itself). Between the quotation marks, a search and replace pattern is specified. The "s" at the beginning indicates the use of substitution function from Sed. The forward slash ("/") after that is a separator. After that comes the pattern that should be replaced (e.g. "{\'a}" – note that there are more backward slashes than what the literal pattern contains; this is so because backslash has a special meaning in the Bash metalanguage and to be taken literally, it must be "escaped" by two more backslashes, therefore, the three backslashed get eventually interpreted as one literal backslash). Then there is another forward slash as a separator, followed with the pattern that should replace the previous one (e.g. "á"). Then another forward slash as a separator, followed by "g" indicating that the substitution should be performed globally, that is, should involve all the lines in the file (the default behavior is that substitution applies to only one line at a time). At the end of the line, the contents of the variable "file" ("$" indicating that it is a variable) are given to Sed to direct it to a file in which the substitution should be performed. Line 35 prints "Done" to indicate that all of the substitutions were performed.

Given the set of 28 substitutions that should be performed repeatedly in a file, it is easy to imagine how tedious such task would be to perform by hand, which is what direct manipulation interfaces require. However, with a textual interface, a number of individual tasks (28 substitutions times number of replaced occurrences) could be condensed into one task of running a script. The enforced sequentiality involved in opening a file, searching for a pattern, replacing it, searching for another one … and eventually saving the modified file, was replaced by a single task that appears to be indivisible (by running the script, every substitution is performed), but whose internals could be rearranged by an informed user.

Appendix 2: Pitivi Architecture



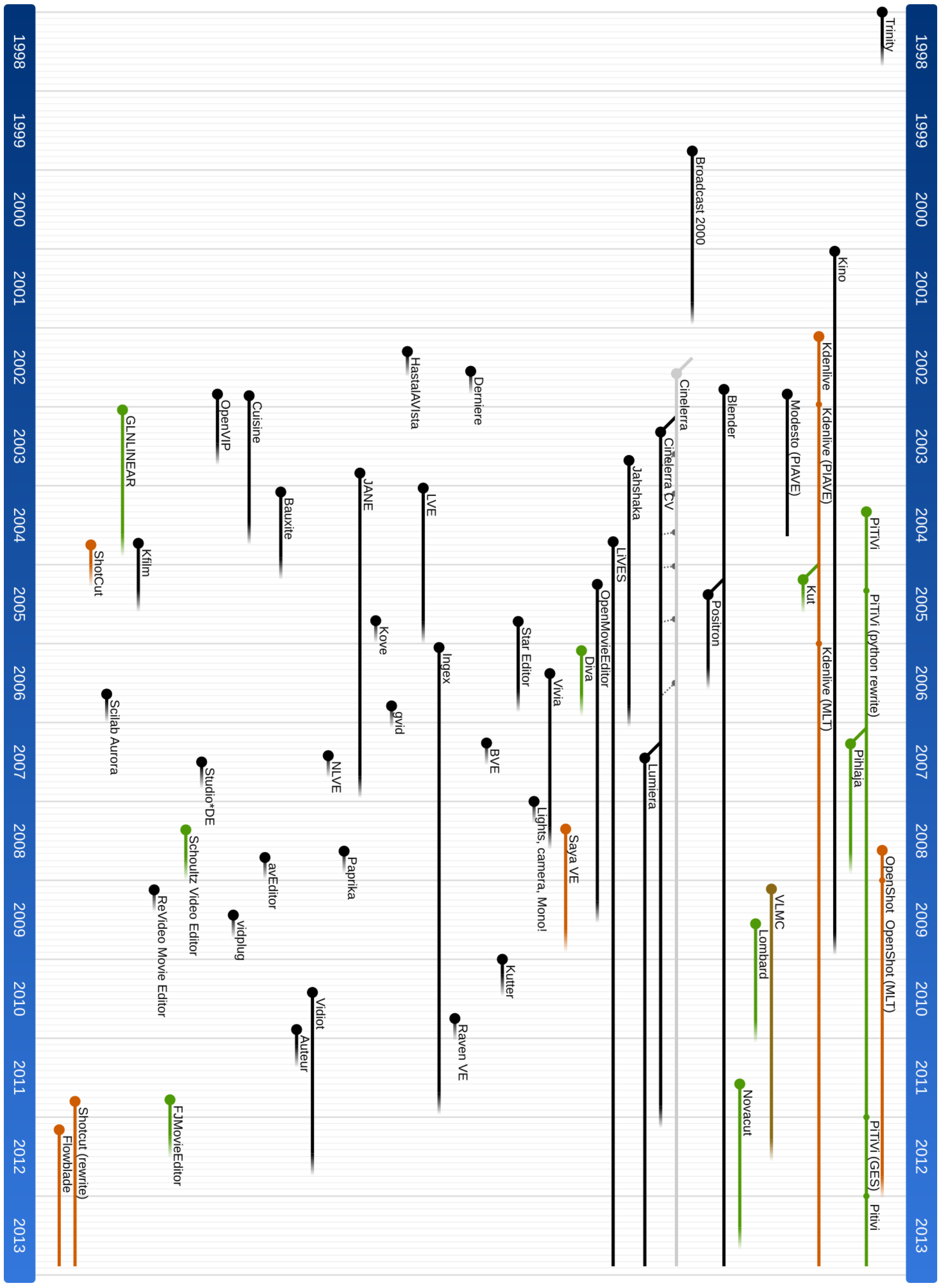
Appendix 3: Pitivi Interface



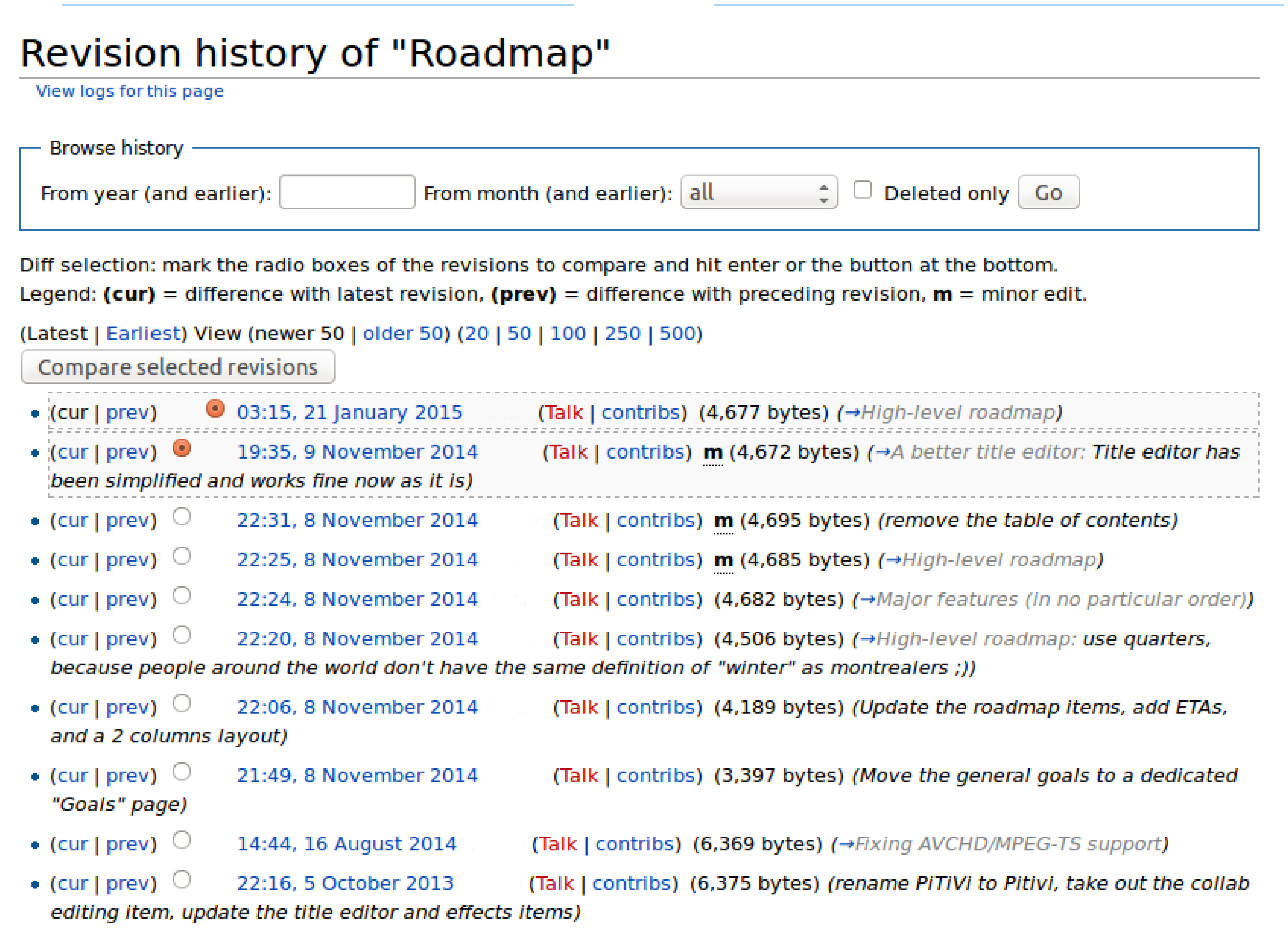
Add media to your project by dragging files and folders here or by using the "Import" button.

Select a title clip to edit or create a new one.

Appendix 4: Overview of Projects Developing Video Editors



Appendix 5: History Log of a MediaWiki Page



Revision history of "Roadmap"

View logs for this page

Browse history

From year (and earlier): From month (and earlier): all ☐ Deleted only Go

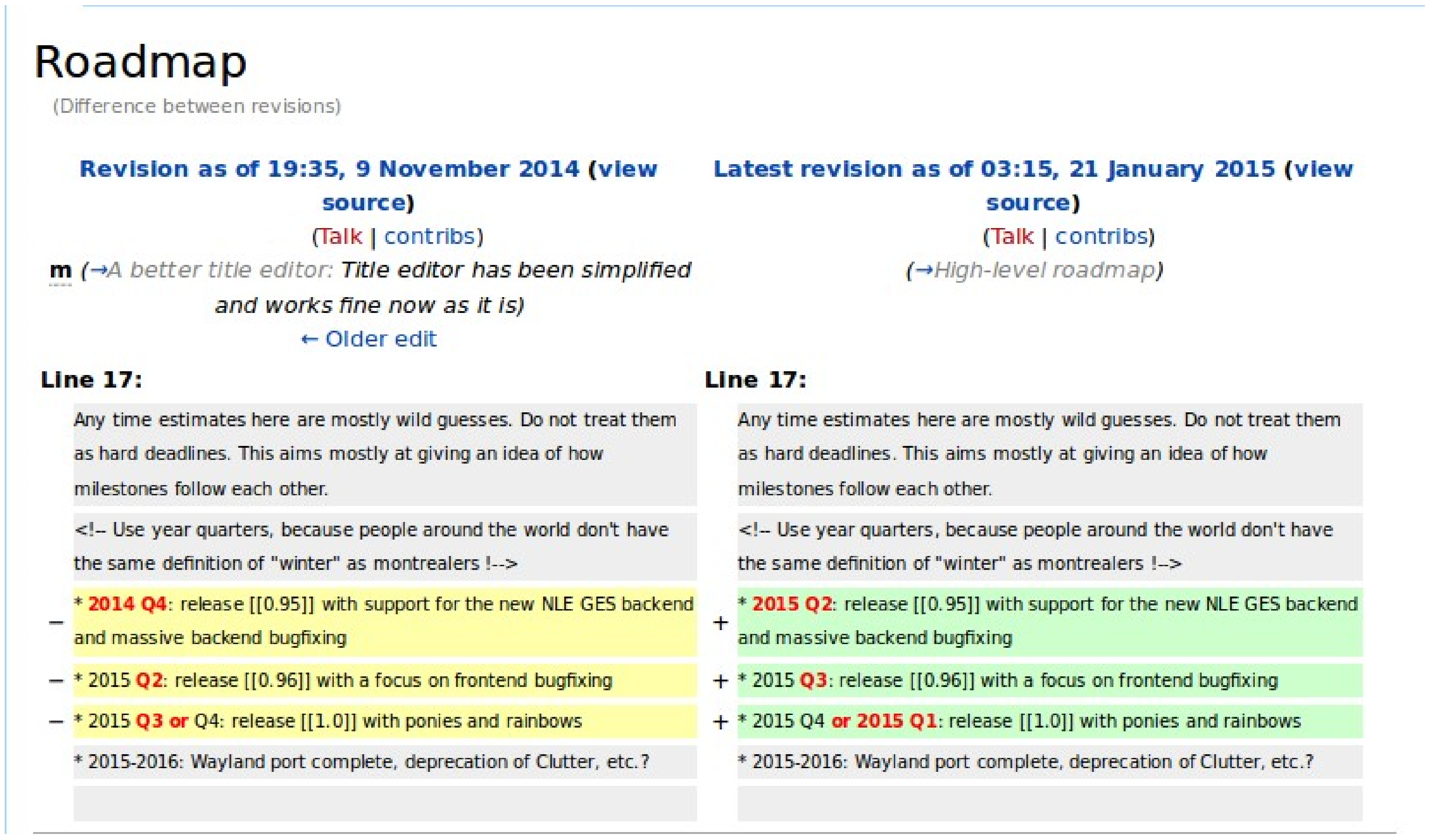
Diff selection: mark the radio boxes of the revisions to compare and hit enter or the button at the bottom.
Legend: **(cur)** = difference with latest revision, **(prev)** = difference with preceding revision, **m** = minor edit.

(Latest | Earliest) View (newer 50 | older 50) (20 | 50 | 100 | 250 | 500)

Compare selected revisions

- (cur | prev) ● 03:15, 21 January 2015 (Talk | contribs) (4,677 bytes) *(→High-level roadmap)*
- (cur | prev) ● 19:35, 9 November 2014 (Talk | contribs) **m** (4,672 bytes) *(→A better title editor: Title editor has been simplified and works fine now as it is)*
- (cur | prev) ○ 22:31, 8 November 2014 (Talk | contribs) **m** (4,695 bytes) *(remove the table of contents)*
- (cur | prev) ○ 22:25, 8 November 2014 (Talk | contribs) **m** (4,685 bytes) *(→High-level roadmap)*
- (cur | prev) ○ 22:24, 8 November 2014 (Talk | contribs) (4,682 bytes) *(→Major features (in no particular order))*
- (cur | prev) ○ 22:20, 8 November 2014 (Talk | contribs) (4,506 bytes) *(→High-level roadmap: use quarters, because people around the world don't have the same definition of "winter" as montrealers ;))*
- (cur | prev) ○ 22:06, 8 November 2014 (Talk | contribs) (4,189 bytes) *(Update the roadmap items, add ETAs, and a 2 columns layout)*
- (cur | prev) ○ 21:49, 8 November 2014 (Talk | contribs) (3,397 bytes) *(Move the general goals to a dedicated "Goals" page)*
- (cur | prev) ○ 14:44, 16 August 2014 (Talk | contribs) (6,369 bytes) *(→Fixing AVCHD/MPEG-TS support)*
- (cur | prev) ○ 22:16, 5 October 2013 (Talk | contribs) (6,375 bytes) *(rename PiTiVi to Pitivi, take out the collab editing item, update the title editor and effects items)*

Appendix 6: Display of Differences Among Two Versions of a MediaWiki Page

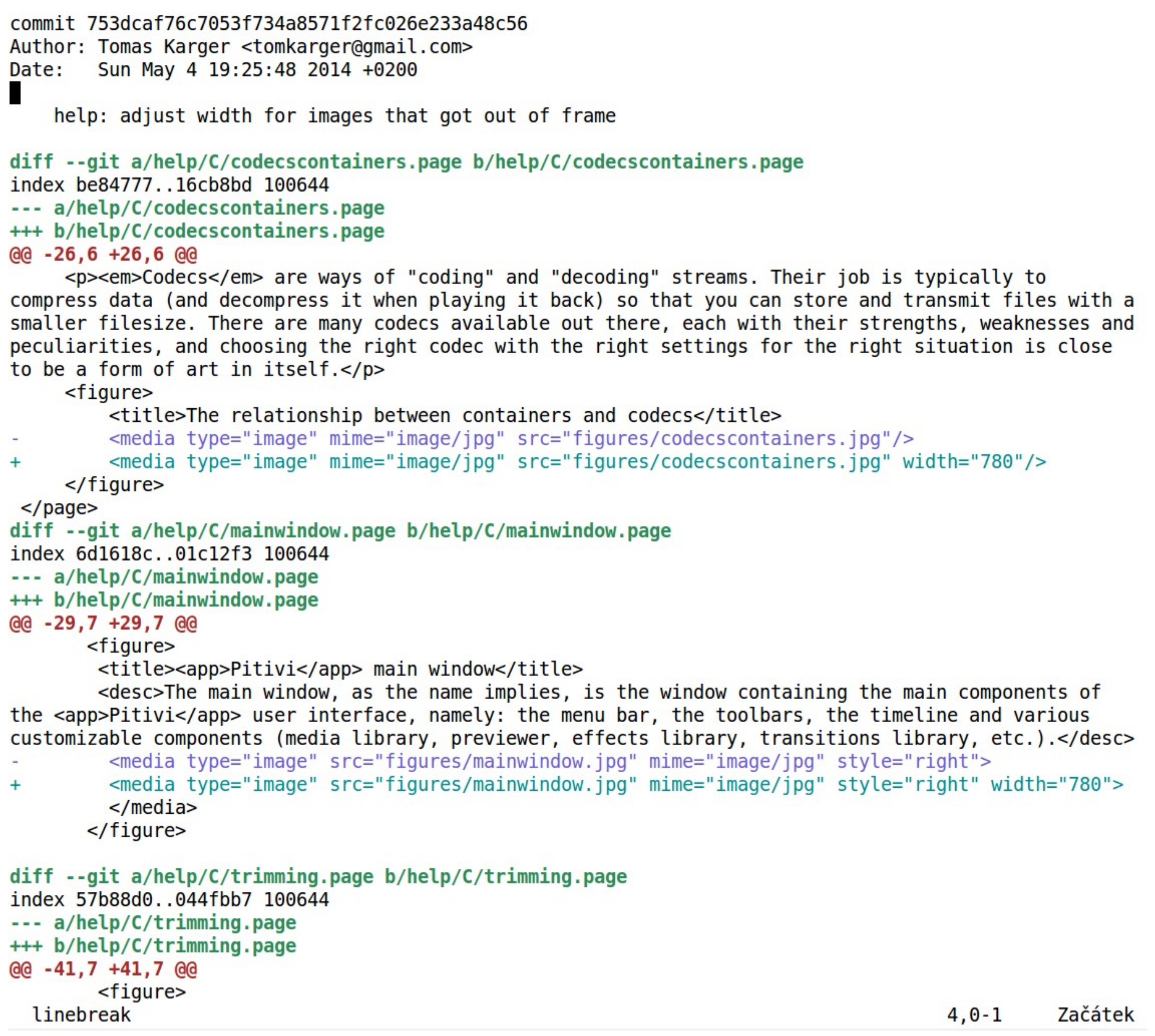


Roadmap

(Difference between revisions)

	Revision as of 19:35, 9 November 2014 (view source) (Talk | contribs) **m** (→A better title editor: *Title editor has been simplified and works fine now as it is*) ← Older edit		**Latest revision as of 03:15, 21 January 2015 (view source)** (Talk | contribs) (→High-level roadmap)
	Line 17:		**Line 17:**
	Any time estimates here are mostly wild guesses. Do not treat them as hard deadlines. This aims mostly at giving an idea of how milestones follow each other.		Any time estimates here are mostly wild guesses. Do not treat them as hard deadlines. This aims mostly at giving an idea of how milestones follow each other.
	<!-- Use year quarters, because people around the world don't have the same definition of "winter" as montrealers !-->		<!-- Use year quarters, because people around the world don't have the same definition of "winter" as montrealers !-->
−	* **2014 Q4**: release [[0.95]] with support for the new NLE GES backend and massive backend bugfixing	+	* **2015 Q2**: release [[0.95]] with support for the new NLE GES backend and massive backend bugfixing
−	* 2015 **Q2**: release [[0.96]] with a focus on frontend bugfixing	+	* 2015 **Q3**: release [[0.96]] with a focus on frontend bugfixing
−	* 2015 **Q3 or** Q4: release [[1.0]] with ponies and rainbows	+	* 2015 Q4 **or 2015 Q1**: release [[1.0]] with ponies and rainbows
	* 2015-2016: Wayland port complete, deprecation of Clutter, etc.?		* 2015-2016: Wayland port complete, deprecation of Clutter, etc.?

Appendix 7: A Git Commit



```
commit 753dcaf76c7053f734a8571f2fc026e233a48c56
Author: Tomas Karger <tomkarger@gmail.com>
Date:   Sun May 4 19:25:48 2014 +0200

    help: adjust width for images that got out of frame

diff --git a/help/C/codecscontainers.page b/help/C/codecscontainers.page
index be84777..16cb8bd 100644
--- a/help/C/codecscontainers.page
+++ b/help/C/codecscontainers.page
@@ -26,6 +26,6 @@
     <p><em>Codecs</em> are ways of "coding" and "decoding" streams. Their job is typically to
compress data (and decompress it when playing it back) so that you can store and transmit files with a
smaller filesize. There are many codecs available out there, each with their strengths, weaknesses and
peculiarities, and choosing the right codec with the right settings for the right situation is close
to be a form of art in itself.</p>
     <figure>
         <title>The relationship between containers and codecs</title>
-        <media type="image" mime="image/jpg" src="figures/codecscontainers.jpg"/>
+        <media type="image" mime="image/jpg" src="figures/codecscontainers.jpg" width="780"/>
     </figure>
 </page>
diff --git a/help/C/mainwindow.page b/help/C/mainwindow.page
index 6d1618c..01c12f3 100644
--- a/help/C/mainwindow.page
+++ b/help/C/mainwindow.page
@@ -29,7 +29,7 @@
       <figure>
        <title><app>Pitivi</app> main window</title>
        <desc>The main window, as the name implies, is the window containing the main components of
the <app>Pitivi</app> user interface, namely: the menu bar, the toolbars, the timeline and various
customizable components (media library, previewer, effects library, transitions library, etc.).</desc>
-        <media type="image" src="figures/mainwindow.jpg" mime="image/jpg" style="right">
+        <media type="image" src="figures/mainwindow.jpg" mime="image/jpg" style="right" width="780">
         </media>
       </figure>

diff --git a/help/C/trimming.page b/help/C/trimming.page
index 57b88d0..044fbb7 100644
--- a/help/C/trimming.page
+++ b/help/C/trimming.page
@@ -41,7 +41,7 @@
        <figure>
  linebreak                                                                     4,0-1      Začátek
```