Recruitment, training, communication and Open Source: A case of health information systems

Master thesis
60 credits

Anders Bjarne Skjelten Gjendem
andegje@ifi.uio.no

May 16, 2008
Contents

1 Introduction 3
  1.1 The Action Research Project 3
  1.2 Motivation 4
  1.3 Research objectives 5
  1.4 Structure of this thesis 5

2 Literature 7
  2.1 Open Source 7
    2.1.1 The Free Software Foundation 8
    2.1.2 The Open Source Initiative 8
    2.1.3 Free Software versus Open Source software 8
    2.1.4 Open Source licenses 8
    2.1.5 Open Source communities 10
  2.2 Recruitment 11
    2.2.1 Motivation 12
    2.2.2 Joining scripts 15
    2.2.3 Feature gifts 17
    2.2.4 Contribution barriers 17
    2.2.5 Balancing growth 20
  2.3 Training 20
  2.4 Communication 22
  2.5 Sustainability 23

3 Method 26
  3.1 Action Research 26
  3.2 Action Research in the field of information systems 27
  3.3 Research approach 28
    3.3.1 HISP 28
    3.3.2 OpenMRS 28
    3.3.3 Training 29
    3.3.4 Development and participation 29
    3.3.5 Meetings 30
    3.3.6 Interviews 30
3.3.7 Observation ........................................... 30
3.3.8 Possible method-related limitations ................. 31

4 Empirical material .................................. 32
  4.1 Background ......................................... 32
    4.1.1 HISP ............................................. 32
    4.1.2 OpenMRS ......................................... 32
  4.2 DHIS .................................................. 33
    4.2.1 INF5750 - Open Source Software development frameworks in global networks ............... 33
    4.2.2 Import/export ................................... 34
    4.2.3 Teaching assistant for the Open Source Software development course ...................... 36
    4.2.4 Documentation ................................... 37
    4.2.5 OSHCA conference in Malaysia .................. 38
    4.2.6 Fieldwork in Vietnam ............................ 39
  4.3 OpenMRS - GSoC .................................... 44
  4.4 Licensing ............................................ 46
    4.4.1 DHIS and licensing ............................... 46
    4.4.2 OpenMRS and licensing ......................... 47

5 Discussion ......................................... 48
  5.1 Recruitment ......................................... 48
    5.1.1 Trajectory ....................................... 50
    5.1.2 Obstacles as a filter ........................... 52
    5.1.3 Feature gifts and contribution barriers .......... 53
    5.1.4 Balancing growth ................................ 55
  5.2 Training and mentoring ............................... 55
    5.2.1 Code review ...................................... 57
  5.3 Controversies and communication ....................... 59
    5.3.1 Resisting change ................................ 61
    5.3.2 Developer types ................................ 63
  5.4 Developer motivations ................................ 64
    5.4.1 Common training ................................ 65
    5.4.2 Real life experience of the system ............. 65
    5.4.3 Cross-domain usage .............................. 65
    5.4.4 Implementers ..................................... 66
    5.4.5 Knowledge transfer ............................... 66
    5.4.6 Contributing to other OSS projects .............. 68
    5.4.7 Central nodes ...................................... 69
    5.4.8 Multi-project clustering ......................... 69
  5.5 Sustainability ........................................ 71
  5.6 Open Source communities ............................. 72
6 Conclusion

7 Appendix

7.1 Open Source license distribution

7.1.1 SourceForge.net

7.1.2 FreshMeat

7.1.3 Google Code

7.2 Open Source licenses

7.2.1 Dual Licensing
Abstract

In this thesis, I compare Free Vertical Domain Information System (FVDIS) projects, represented by the District Health Information Software version 2 (DHIS 2) project and OpenMRS, with traditional open source projects. The comparison focuses on recruitment, developer training, communication and sustainability.

The thesis was written using an Action Research approach, where I took part in both projects - as a developer and teaching assistant in DHIS 2 and as an intern in OpenMRS.

After comparing existing literature on open source software development with my experiences from OpenMRS and DHIS 2 project work, I have concluded that there are differences between FVDIS projects and traditional open source software projects related to recruitment, developer training, communication and sustainability, and these differences should be taken into account when planning and executing FVDIS projects.

Notably, the relatively limited user groups of FVDIS applications cause challenges for recruitment compared to traditional open source projects. DHIS 2 and OpenMRS takes a more organized approach to recruitment to reduce this problem. These approaches to recruitment - mandatory participation through a university course and interns via Google Summer of Code - mean that developer training has to receive a higher priority as compared to open source projects.

Based on the DHIS 2 and OpenMRS cases, I believe that handling recruitment and developer training in a fashion appropriate for FVDIS projects is critical to the sustainability of such projects.
Acknowledgements

I would like to thank Knut Staring for guidance in writing this thesis. Your detailed feedback and ideas have been greatly appreciated.

Are Wold, Lene Ostby, Jon Are Rakvaag and Inger Margrethe Saunes deserves a big thank you for keeping me going and providing feedback.

I would also like to thank my travel-mates Margrethe Store, Eivind Berg and the Vietnamese team for a great experience abroad, as well as the rest of the HISP/DHIS 2 and OpenMRS teams for interesting discussions and friendship.
Chapter 1

Introduction

This thesis is based on the development of two open source health information systems and my participation in these. I will present my findings related to capacity building and sustainability in open source software projects.

1.1 The Action Research Project

I have been involved in the Health Information Systems Programme (HISP) network, and specifically with the District Health Information Software (DHIS) version 2, since the autumn of 2005. The HISP network is a global research and development network with focus on information systems and open source software for public health care, aimed at developing countries. The DHIS 2 project is an effort to port an existing open source health information system to a new development platform. The development occurs in a global context with development and test implementation nodes situated in a number of countries around the world.

Development of the DHIS started in South Africa in 1996 [6], and is still ongoing with the current version being DHIS 1.4. DHIS is deployed in a number of countries and supports a large user base worldwide. The DHIS 1.4 software is based on Microsoft Access, and requires Microsoft Windows to run.

I have been involved with porting the DHIS 1.4 functionality to the new version of DHIS. The District Health Information Software version 2 aims at being cross-platform to support implementations on both Microsoft Windows, Linux or any other major platform. An important sub-goal of this is that DHIS 2 is able to run in a completely open source and free environment, potentially removing many obstacles to free sharing of the application for use in many contexts, as well as promoting open collaboration around the software development. The idea of open licenses thus extends beyond the need to remove the cost, though that is
also important for public sector organizations in developing countries. The development of DHIS 2 started in the autumn of 2004 at the University of Oslo [31, p6] and is licensed under the BSD license (see 2.1.4 on page 10). Currently, the main development effort on DHIS 2 is performed in Norway, Vietnam and India.

HISP works closely with other organizations such as the OpenHealth project in the World Health Organization (WHO) [21] as well as with complementary projects such as OpenMRS - an open source medical record system.

OpenMRS has a similar background to DHIS. In 2004, Burke Mamlin and Paul Biondich from the Regenstrief Institute were hired as consultants to scale up an existing Microsoft Access-based system in western Kenya [28]. They designed and developed the AMPATH Medical Record System (AMRS) in response to this task. A year later, they learned that they shared similar approaches and data models with a project from Partners in Health. The collaborative effort named OpenMRS was started.

In the summer of 2007 I became involved with the OpenMRS development team as an intern through the Google Summer of Code program, which aims to recruit developers to the open source community. Together with developers and mentors from OpenMRS I took part in the development of synchronization functionality for the OpenMRS software.

1.2 Motivation

While working with HISP and OpenMRS, I noticed that the projects and communities surrounding them differed from open source as I previously knew it. For several years I have been using open source software both for studies, privately and to gain experience in operating common server software. Because of this, I have followed various projects over time, lurking around their communities as a user, primarily observing their work. After working with HISP and OpenMRS for some time, it became apparent to me that there are differences between HISP/OpenMRS as free vertical domain information systems (FVDIS), and the way I previously viewed open source.

By free vertical domain information systems I refer to open source organizational information systems catering to vertical domains such as health care. These can be contrasted with horizontal infrastructure software like web servers and browsers [42, p2].

I also believe the vertical domain of these projects might affect their recruitment, training and communication, and though the literature on open source has expanded greatly over the last decade, this phenomenon has received limited attention (see e.g. Scacchi’s [38] review of recent research results and methods in open source software development, with
a notable exception being Fitzgerald and Kenny [9]).

**Hypothesis 1:** There are structural differences between free vertical domain information systems (FVDIS) and “traditional” OSS projects

**Hypothesis 2:** These differences affects recruitment, developer training and communication, and thus the sustainability of projects

### 1.3 Research objectives

**Research objective:** Explore the difference between traditional OSS projects and how OSS development and capacity building is affected in FVDIS developer organizations.

I will base the definition of “traditional OSS projects” on Scacchi’s [38] review of recent research results and methods in free/open source software development.

Specifically I will try to answer the following questions:

**Research question 1:** What are the differences and similarities in socialization in FVDIS type projects, as compared to traditional open source projects.

- Specifically, I will look at recruitment, developer training and communication patterns and the interaction between these.

**Research question 2:** How do the differences affect the sustainability of such projects?

The discussion will be based on data collected through my participation in two open source software projects:

**DHIS 2:** Developed import/export functionality for the DHIS 2 software to facilitate data transfer between implementation sites, and worked as a teaching assistant in an open source software development course using DHIS 2 as the case.

**OpenMRS:** Contributed to the development of data synchronization functionality between OpenMRS servers.

### 1.4 Structure of this thesis

- Literature
- Method
• Empirical material
• Discussion
• Conclusion
Chapter 2

Literature

This chapter presents the literature background of my research. Existing literature on socialization will be presented in connection with my research objectives, focusing on the traditional view of OSS project organizational structure, recruitment, training, communication and sustainability.

2.1 Open Source

Open source software is software where the human readable source code is made available and meets the Open Source Definition [23]. However, the software is rarely placed in the public domain by the developers, where it is not covered by copyright and anyone could do anything with it [48, p84]. In the absence of a corporate organization to act as an ordering device, open source software development is usually based around an intellectual property regime, in the shape of licenses, where the licenses are the major formal social structure surrounding the development [48, p85, p179].

Raymond states that the ideology of the open source culture is fairly complex. He comments that all members of the open source culture agree that freely redistributable software that can be modified to fit changing needs is a good thing and worthy of significant and collective effort [36]. He continues to say that, however, the reasons individuals and various subcultures give for this belief vary considerably.

A huge amount of open source projects exist. Though most fail, many of them are successful according to Fogel [10, p1]. He adds that «We tend not to hear very much about the failures. Only successful projects attract attention, and there are so many free software projects in total that even though only a small percentage succeed, the result is still a lot of visible projects» [10, p1]
2.1.1 The Free Software Foundation

The Free Software Foundation (FSF) was founded by Richard Stallman in 1985 to support the free software movement and the GNU Project. In its early days it started out as an employer of free software developers working on writing free software. As the free software movement grew stronger it now focuses its resources on legal work and structural issues related to the free software movement. FSF is the publisher of the most widely used license for free software - the GNU General Public License (GPL)

2.1.2 The Open Source Initiative

The Open Source Initiative (OSI) was founded in 1998 by a group of persons, among them Eric S. Raymond. OSI uses the term open source software as an alternative label to FSFs free software, thus OSI is often considered an offshoot of FSF, taking a more pragmatic rather than philosophical approach. Both organizations maintain lists of Open Source licenses, with only minor differences between them. Stallman describes the relation between FSF and OSI like this: «We disagree on the basic principles, but agree more or less on the practical recommendations. So we can and do work together on many specific projects» [40]. Examples of such co-operation are the attacks by Microsoft on the GPL license (http://en.wikipedia.org/wiki/Halloween_document), and the SCO lawsuit (http://en.wikipedia.org/wiki/SCO_v._IBM) against the Linux kernel.

2.1.3 Free Software versus Open Source software

Free Software and open source are closely related. The term Open Source was derived from free software movement, but focuses on the practical benefits of open source rather than the ethical aspect of Free Software. In the term Free Software, free is emphasizing the freedom to do as one wishes with the code versus the software being without cost. In most cases free software is also free as in gratis/libre due to the nature of the General Public License (GPL), which causes some confusion. Several other less intrusive open source licenses have been created after the GPL, like the Mozilla Public License.

2.1.4 Open Source licenses

The Open Source Initiative has published a definition of what a license may and may not state to comply with the Open Source Definition [23]. Some of the terms are:
• Free redistribution
• Access to source code
• No discrimination
• Must allow modifications and derived works

Similarly, the Free Software Foundation points to the Free Software Definition [14] and its 4 kinds of freedom for the users of the software:

• The freedom to run the program, for any purpose
• The freedom to study how the program works, and adapt it to your needs
• The freedom to redistribute copies so you can help your neighbor
• The freedom to improve the program, and release your improvements to the public, so that the whole community benefits

OSI maintains a list of approximately 50 Open Source approved licenses. In the appendix I present and compare data retrieved from two major websites for hosting and listing software. Based on these we get a fairly good impression of the distribution of the licenses used.

Choosing the correct license for a project is essential. Lawrence Rosen argues that «Software licensors choose their licenses based upon their particular philosophy and business models, and they intend their licenses to define the expected behavior of their licensees around the world. Because philosophies and businesses differ, so do licenses» [37]. Rosen among others also divide the licenses into two general categories and explain the difference between them: «(1) Those licenses that allow derivative works of the software to be used and distributed as part of proprietary programs, and (2) those that require derivative works to be distributed under the same open source license as the original. The BSD license is the archetype of the first kind, what I call academic licenses. The GPL (version 2) is the most popular of the second kind, the so-called reciprocal or copyleft licenses. The license philosophies of these two categories of licenses are inherently irreconcilable, the first being compatible with proprietary derivative works and the other not, but they each make a valid point about the freedom to copy, change, and use software» [37].

The two most relevant open sources licenses are described in the following sections. Several other common open source licenses are described in the appendix (see 7 on page 82).
**BSD - Berkeley Software Distribution**

The BSD license is a license with very few restrictions compared to most other licenses. The BSD license was first written by the Regents of the University of California, Berkeley and used for the Berkeley Software Distribution - a Unix-like operating system - BSD. There is no copyright or copyleft - where the term copyleft is the permission to reproduce, adapt or distribute the work as long as any resulting copies or adaptations are also bound by the same copyleft licensing scheme. BSD is regarded as “copycenter” - you’re free to copy it as you wish, with or without modifications. As long as the license is distributed along with the software in both binary and source code form, and the name of any derived works differ from its heritage, the usage of the code has no other limitations. BSD code can thus be taken, modified and used in proprietary settings without any requirement to provide changes back to the forked project. The BSD license accepts linking from code with a different license. The BSD license is compatible with the GPL license, and approved by both OSI and FSF.

**MPL - Mozilla Public License**

The MPL was written by Mitchell Baker when she worked as a lawyer for Netscape, and later released as version 1.1 as she worked for the Mozilla Foundation. The MPL is a weak copyleft license. Any files already licensed under the MPL must remain under the MPL even after modification, but MPL licensed files may be combined with other non-MPL licensed code without affecting those. The MPL is approved by both FSF and OSI.

### 2.1.5 Open Source communities

The Linux kernel project started by Linus Torvalds in 1994 has become one of the most well-known open source projects and has as its size and impact increase attracted lots of media attention. The repeated interest from media over time and a substantial user base has over time ensured that many other mature open source projects running on the kernel have emerged and created other developer communities around them. Open source projects are often thought of as loosely connected groups of developers “scratching their own itch” - trying to solve problems and improve software important to themselves. Other motivations may be to gain other types of personal value - like increased knowledge in some area that they find interesting.

As the number of projects increases, projects with a different background have emerged too. Several big software companies have open-
sourced their products, examples of that is the Mozilla Project. Netscape Corporation chose to open source its Netscape Communicator browser suite in 1998 [3] as what is today known as the Mozilla Project and its Firefox browser. At the time, Netscape was in a fierce battle with Microsoft Internet Explorer, and Netscape decided that a broad-based development effort within the software development community was the way to go to produce a future browser as a shared resource. Similarly the project today known as OpenOffice started out as StarOffice, developed by the German company StarDivision. StarDivision was acquired by Sun Microsystems in 1999 and open sourced under the LGPL license in 2000, aiming at reducing the market share of Microsoft Office [49].

Both of these projects started out as proprietary software projects that at a later stage were open sourced. Though both of them are regarded as fairly successful today, there has been challenges with involving the community of existing open source developers with pre-existing projects with a certain degree of organizational backing. Mitchell Baker in the Mozilla Foundation admits that it may possibly be true that the Mozilla project was not a true open source project during the initial time, because of the tight relation with Netscape employees and their contributions, and Netscape managements long involvement in the project. She especially points at the management and that they lived in an intensely uncomfortable setting as the control of the project was moved over to mozilla.org and out of their hands [3, p5].

It would be easy to conclude that open source software quality might not be very good as it's in most cases developed for free, and thus probably not by professional software developers. A review from 2002 by Stamelos et al. on a sample of 100 open source C projects included in a large linux distribution, did show that the structural code quality was better than one might expect of open source, but lower than the quality implied by the business-standard they compared it to [41, p56]. In the projects with large adoption and high maturity it is probably closer to the truth that most of the open source developers work in normal day jobs developing proprietary software and contributes to open source projects in their spare time, or are about to become professional developers, if they are not full time employees in a project like Mozilla, sponsored by a company. Raymond wrote that «Every good work of software starts by scratching a developer's personal itch» [35]. These days, the itch may very well be your employers’.

2.2 Recruitment

To support the long-term survival of OSS projects, recruitment of new developers is a key issue. Ducheneaut explains that «Socialization of
newcomers is particularly crucial for the Open Source movement. (...) as successful OSS projects mature, their technology grows more complex and only a few people who have been actively involved in their development fully understand their architecture. If these key members were to leave, the project would eventually falter. This makes socializing new members an essential ingredient in the long-term survival of OSS projects» [8, p326]. Depending on the setting, this may be more important for some projects. A university setting may provide access to many new potential contributors, but also represent challenges when it comes to keeping participants around for a longer period of time, as their motivation may be based on other factors than contributing to the project. For instance, students may have to participate in the development as part of mandatory assignments.

2.2.1 Motivation

Hars and Ou identified two broad types of motivation: Internal factors, and external rewards [19, p1]. Understanding developers' motivation for participating in open source projects is interesting as it may have implications for the recruitment process, developers' training and their contributions to projects.

Internal factors

Internal factors are rooted within the individual developer. It is argued that open source software developers are motivated by their own hobbies and preferences. Other rewards are based on increasing the welfare of others, rather than monetary incentives [19]. Harsh et al. describes the intrinsic motivation of developers as «(...) the feeling of competence, satisfaction and fulfillment that arises from writing programs» [19, p3].

Hars and Ou expects that «open source programmers with intrinsic motivations will spend more time and effort in open source projects» [19, p3]. They point out that developers' needs may not necessarily be linked to the needs of the users of the software. In cases where these two groups are not identical, incorporating the needs of the users may be a problem [19].

The second internal factor Hars and Ou discuss is altruism. They define altruism as «doing something for another at some cost to oneself» [19, p3], an opposite to selfishness. By providing open source software to others, the developers give of their time, energy and opportunity costs and thus belong to the altruistic category [19].

Hars and Ou also presents a variant of altruism that they label “community identification”, where developers identify themselves as part of a community, and align their goals with the community’s interests. Hars
and Ou state that those developers «(...) may treat other members of the community as their kin and thus be willing to do something beneficial to others but not to themselves» [19, p3].

**External rewards**

Open source software is for the vast majority of developers something they do not get compensated for directly. However, indirect rewards may be obtained through increased marketability and skills, or by selling related products and services [19].

Hars and Ou propose that developers may view their participation in open source software projects as «(...) an investment from which they expect future returns» [19, p3]. Different categories of such returns was identified [19, p3-4]:

**Revenues from related products and services** An approach in the open source software business is to sell services like hosting solutions, training, support and commercial consulting.

**Human capital** Increasing personal skills, capabilities and knowledge by learning and practicing in open source projects may lead to better job opportunities.

**Self-marketing** Using open source projects to prove capability and skills. Hars and Ou point out an implication of this; the self-marketing may lead to commercial software vendors showing interest and thus the best developers and most productive minds may disappear into commercial software development.

**Peer recognition** In open source software rapid and constructive feedback is common. Hars and Ou state that «Feedback always has a positive effect - it shows the programmer that people are using their contribution» [19, p4]. They conclude that «feedback is self-reinforcing: it encourages the author to spend additional effort to perfect his code» [19, p4].

Many open source projects were started based on a personal need for some software, or for specific software functionality. Hars and Ou state that «it shows that participants of open source projects may act rationally after their own self-interest» [19, p4].

They also state that complex products with few dependencies are more likely to be sold commercially than provided for free, using the example of Sendmail and its commercial add-on products.

The last implication of personal need Hars and Ou mention is the alignment of users and developers. They state that «both are interested
in improving the functionality; both are willing to invest in improvements» [19, p4], and then point to the fact that current license agreements in the traditional software business «(...» prevents customers to invest in their software by making modifications and by sharing the improvements with others» [19, p4].

Scacchi’s research into the developers’ motivations revealed similar findings. Through his exploration of FOSS research, he identified a wide range of possible motivations [38, p8]:

- Fun, personally rewarding, provides a venue where they can improve technical competence that may not be possible within their current job or line of work
- Building trust and reputation
- Achieving “Geek fame”
- Being creative
- Giving and being generous with one’s time, expertise and source code
- Maintain and improve software development skills

Building up social capital and recognition can also be achieved through becoming a central actor in a social network of software developers interconnecting multiple FOSS projects [38, p8]. Scacchi refers to Hars and Ou stating that “60% or more FOSS developers participate in two or more projects, and on the order of 5% participate in 10 or more FOSS projects” [38, p8]. However, a core group of developers control the architecture and direction of a project, and those typically contribute the vast majority of the source code.

In addition to the mentioned motivations, research examined by Scacchi suggest that «(...) individual FOSS developers often benefit from higher average wages and better employment opportunities (at present), compared to their peers lacking FOSSD experience or skill» [38, p14].

Hars and Ou [19] found that the participants’ motivations were more complex than expected, and that the external factors had greater weight than the internal factors. A key factor was the personal need of the
developers, though building human capital and self-marketing was also important.

Hars and Ou found that «Hobbyists and students are the most internally motivated. Salaried and contract programmers, in contrast, seek to sell related products and services» [19, p7]. Their results demonstrated that a surprisingly high number of developers are paid for their open source development efforts, and that «These developers are most concerned with self-marketing and fulfilling personal software needs» [19, p7].

2.2.2 Joining scripts

Von Krogh et al. define a joiner as someone who is eventually given access to the code repository, and “joining script” as «the level and type of activity a joiner goes through to become a member of the developer community» [47, p1227]. These joining scripts are implicit and did not exist in writing, but were nevertheless found important. They also point out that joining has a cost to any would-be developer in the project, as they need to emerge from a much larger group of list participants.

They propose that «Participants behaving according to a joining script (level and type of activity) are more likely to be granted access to the developer community than those participants that do not follow the project’s joining script» [47, p1229].

Based on previous research, they state that to be able to do that, one need to «demonstrate some level of technical expertise as well as understanding of what the community expect in terms of behavior, in order to make a contribution to technical development» [47, p1226-1227].

The level of activity is one key element to the joining script. Messages posted to the developers mailing list indicated that a significant period of observation often was needed before someone felt they could contribute to the technical discussion [47]. This period is commonly known as “lurking”.

The type of activity is central to a joining script. In two stages, they studied the developers' initial mail to the developer list as well as the type of activity a joiner underwent before being accepted. None of the joiners Von Krogh et al. observed proposed unsolicited “new” technical suggestions in their initial contact which might indicate that it is wise to start out humbly. Findings show that «Overall, the community welcomed people who announced their interest and indicated their skill levels, but they expected new participants to find their own tasks to work on» [47, p1228]. However, because of the need for a significant knowledge level in many cases, many of those who announced their interest never became developers.
Analysis of the mailing list identified that 9 out of 22 categories of mails turned out to be significant in distinguishing joiners' from non-developers' activity. Among joiners, the relative numbers of mails offering bug-fixes, containing general technical discussion, reporting bugs, and expressing repeated interest to contribute was significantly higher than with non-developers.

Von Krogh et al. identified three kinds of people among the joiners, the ones who:

- Noticed a problem, fixed it and provided the solution back to the community. The kind of person that gets accepted.
- Expressed interest in helping out, and ask for a task to start working on. These tend not to do anything.
- Finds the project interesting but wants to make large changes to it. According to the interviewee they tend to start fighting with core architects, who decide not to follow the suggestions.

Ducheneaut identified a 6-level trajectory of roles that developers tend to evolve through over time [8, p349]:

1. Peripheral monitoring of the development activity
2. Reporting of bugs and simultaneous suggestions for patches
3. Obtaining CVS access and directly fixing bugs
4. Taking charge of a "module size" project
5. Developing this project, gathering support for it, defending it publicly
6. Obtaining the approval of the core members and getting the module integrated into the project's architecture.

The initial 3 levels of this trajectory is similar to the joining script Von Krogh et al. describes. New developers monitor the project, contribute bug reports and patches, before possibly being granted access to the code repository at some stage.

Ducheneaut propose that the initial lurking period (level 1) can be shortened by «(...) highlighting the most dynamic and controversial conversations (...), these usually reveal the underlying socio-technical structure of a project during "trials of strength"» [8, p358]. It is suggested that one should start out humbly by contributing to preexisting problems before moving on to more significant accomplishments, typically level 4 [8, p349]. Ducheneaut found that «Most contributors stop at the bug reporter stage, or did not evolve at all» [8, p353]. In his case 4 out of 50 newcomers evolved into the developer role.
2.2.3 Feature gifts

During the initial period of working with a project developers may provide feature gifts to the community. Von Krogh et al. propose that «Feature gifts by newcomers are related to their specialization in open source software projects» [47, p1233]. They compare this with a social exchange where «(...) one can assume that individuals form relationships to maximize rewards and minimize costs, and that gifts are a part of this process» [47, p1233]. They also state that their «(...) findings and analysis confirm an evolving idea in the literature; open source software innovation hinges on contributors giving gifts in the form of code, in this case features» [47, p1233]. As well as the learning benefits obtained by contributing the gift, they found that it increased early specialization of newcomers.

They also found that the contributed feature gifts were provided on the basis of prior knowledge and experience that the developers had acquired elsewhere. Thus, the cost related to the contribution was relatively low to the newcomers [47].

2.2.4 Contribution barriers

In a blog post Greant [16] claimed that the different generations of OSS projects have had their own set of challenges and strengths. However, a common need among them is the availability of participants with some resources. These resources consist among other of «(...) programming skill, thick skin, spare time and a problem to solve» [16]. He points out that this excludes whole classes of people from participating in online communities. He claims that «Even today, where open source and free software is fairly commonplace, people are often intimidated at the prospect of getting started in a given FOSS community. Many projects don’t have an easy point of entry and a novice asking the wrong question may get a rude reception» [16].

The idea Greant presented to help relieve this problem was to set up «(...) a gratis, open-to-all, highly visible, community-driven, multi-day coaching session designed to help people get a solid start participating in, starting and learning key techniques from free culture, free software and open source communities» [16]. Greant found support for this idea among experienced open source contributors he talked to. They wanted an approach that was a fast and non-threatening way to get started for the novices, to avoid the intimidation by the complexity of the projects.

The approach taken by Greant has similarities with the Google Summer of Code (GSoC) project. One of GSoCs goals is to «Help open source projects identify and bring in new developers and committers» [15]. In the case of GSoC this is done by paying students to work on third party
open source software projects together with mentors provided by these organizations. The mentors provided are potentially useful to reduce these barriers. However, many of these students are highly skilled and motivated. As these are picked from a large number of applicants, they likely do not represent the ones who need these mentoring resources the most.

Von Krogh et al. propose that «In an evolving software architecture of open source software projects, contribution barriers of modules (modifying and coding, variation in computer language, plug-in, and independence) are related to the specialization of newcomers» [47, p1233].

They examined specialization by analyzing the target of the code submissions, which maps to which module was changed. They regarded “high” specialization as modules changed over time by a developer as opposed to “generalization” which were indicated by multiple modules changed by a developer. The contributors often apply existing domain knowledge to the project. Most of the new developers started out by contributing to either the “build and install scripts” module or the “clients” module, typical entry-level modules. On the other hand a small group of developers showed a “generalist” or low specialization tendency by contributing to most of the modules; typically known as “core developers” [47]. Benefits of rotating developers among jobs was identified [47], these were:

- Broadening the understanding of a project
- Increased sensitivity to coupling of tasks
- Better management of interfaces

Von Krogh et al. propose a construct they term “the contribution barrier”. The barrier pertains to four items, erected by complex open source software technologies [47, p1231], and is interesting both with regards to recruitment and developer training:

**Ease of modification:** «ease of modifying and coding module» Refers to the complexity of the source code and the level of difficulty of the used algorithms in order to achieve the desired goals. Typically relative to the developers' previous experience, domain knowledge and professional skills.

**Language flexibility** «the extent to which the potential developer can choose the computer language used to code for the module can vary» Some languages can be complex and difficult, while others can be fairly simple, flexible, and in common use. A common and popular language may attract a large number of potential contributors.
Flexible module framework  «ease with which to “plug” the module into the architecture» Clearly defined interfaces between modules reducing the need for detailed knowledge of inner workings of other modules.

Coupling  «the extent to which a module is intertwined or independently working from the main code» Modules with low coupling, that work independently and may be used optionally and/or alternatively is positive. Breaking a module with low coupling may not necessarily break the rest of the system. Developers consider the barriers for contributing to such modules to be lower.

Von Krogh et al. also propose that «Feature gifts by newcomers are related to contribution barriers in an open source software project» [47, p1234]. They observed that a specific gift from a newcomer lowered the contribution barriers of those newcomers that came after, which affected recruitment and developer training.

Another common barrier erected to resist unwanted change is the concept of code review. A formal approach was found in Ducheneaut’s case where they practiced code reviews at the end of module development, rather than continuously throughout development. The module would be scrutinized by the entire community before core members finally delivered a verdict of acceptance or rejection [8, p351]. To successfully get the changes accepted and integrated, the participant has to pass the foes: «(…) the “foes” here are the entire network, designed to resist change, which must be weakened in strategic areas and eventually reconfigured if a participant’s contribution is to be accepted» [8, p353].

Ducheneaut view open source software development as politics by other means. He suggest that black-boxing the relationships between actants is used as a trick to fortify a hybrid network. Anyone who would like to challenge the current state of the project will thus have to uncover the relationships first. He suggest that this can be done by probing the network to reveal its structure: «By asking simple questions about the current state of the project, a participant can see from the responses he obtains who is connected to a particular artifact, and what the nature of this connection is» [8, p354].

Contribution barriers erected by not giving away a projects inner workings and details was used as a selection process. Ducheneaut explains that «(…) by adopting a somewhat distant attitude, the project leaders make sure that they do not have to constantly “hold the hand” of newcomers and waste an inordinate amount of time introducing them to the subtleties of software development. A certain amount of selection is necessary, if only to allow the core members to focus on their tasks (…)» [8, p357]. He concludes that these obstacles function as trials and
rites of passage for the newcomers and is put in place to ensure that these individuals are a good fit for the project.

In the case of a conflict or controversy barrier, one needs to assemble a network of allies first to overcome it [8, p344]. Ducheneaut says that «Indeed statements regarding a controversy are weak if they are left alone. To make a statement stronger, it needs to be connected to what others have said beforehand. This way anybody opposed to the solution offered has to attack not only the solution and its provider, but also a string of other propositions and assertions made by others beforehand» [8, p344].

2.2.5 Balancing growth

Recruiting developers is important, but it seems there is a balance between recruiting and efficiency. Large projects like the Linux kernel project is organized like a dictatorship, with one strong leader on the top making the main decisions and a crew of trusted developers helping him manage the project. An article based on the OpenMRS collaboration states that they have «(...) tried to keep the core group of developers relatively small, especially at the early stages of collaboration» [28, p529]. Scaling up potentially hurt collaboration as challenges with coordination emerge. The OpenMRS project experienced «(...) tension between bringing more developers on board and getting overwhelmed by the number of voices» [28, p531].

2.3 Training

Different open source projects have different approaches to training, with natural variations due to their setting. Open source participants have to master a wide range of skills to be able to successfully participate in the community. FVDIS emerging from a university setting may have training of developers as an important bi-effect and even sub-goal of the project. Other open source projects have different approaches to training, as Ducheneauts findings reveal.

Greant proposed a list of potential topics for some training sessions he wanted to set up for new open source contributors, both developers and non-developers, showing the range of topics. His list contains among others the following [16]:

- Revision control systems
- Licensing
- How to use and contribute to Wikis
• Project hosting sites and tools
• Filing and tracking bugs, patches
• Documentation
• Localization of documentation and software

Greant’s list contains many of the same items as Fogel’s list of what a project needs as a minimum standard set of tools: A web site (project hosting site), mailing lists, version control (revision control), bug tracking and real-time chat. Also, the project must decide on a license [10, p33, p47-48].

In addition to this, some level of coding skills is often needed. Ducheneaut found in his case analysis that «...it looks as if participants come already equipped with good programming skills, and learn instead how to contribute meaningfully to a fairly large-scale project such as Python» [8, p352]. However, he stated that participants in open source communities need skills that go far beyond purely technical knowledge. In his case, they need to learn the political process of how to participate and how to build an identity that will help get them get their ideas accepted and integrated, not only the individual process [8, p352].

Ducheneaut’s study revealed that there was no evidence of coaching in the project: «There is also little evidence of explicit coaching or teaching from established experts. Instead the participants have to discover by themselves what the norms of participation are» and «It is interesting to remark that I found no evidence of coaching of any kind from the project’s members: the acts of finding bugs, reporting them, and proposing a solution to them all stem from the participant’s initiative» [8, p352].

The setting Ducheneaut operates in is quite harsh, which is not surprising in a mature project. He found that the Python project is not a place for novices to learn about computer science as that knowledge is assumed. Rather, what one has to learn is «how to participate and how to build an identity that will help get his ideas accepted and integrated» [8, p352].

Tommerholt states that capacity building is a way to reduce “gaps of understanding” [46, p98]. He refers to training (capacity building) of community members as a way of increasing tool and framework knowledge and thus reducing communication barriers. This capacity building has been practiced in different ways in various nodes. In Vietnam, Norwegian master students have given general program lectures [46, p98], assignments as well as occasional hands-on/pair programming sessions. In Ethiopia, a Vietnamese master student ran practical sessions, before teaching more abstract concepts [30]. In Norway, the training has been
held through the open source software development course with a combination of lectures and project group work with guidance from HISP developers. The various approaches might be partially explained through differences in the educational systems as discussed by Overland [34, p87, p91, p108]

2.4 Communication

Communication is an integrated and important part of open source software development. The politics presented by Ducheneaut is tightly knit to communication, as it is needed to build the necessary network of allies. Ducheneaut found that «(...) the list of active members is always shifting because developers have differing free time, availability, and interests. To work with this large and dispersed group, you’ll have to learn who’s the right person to answer a question, how to convince the other developers of the usefulness of a patch, how to offer helpful criticism, and how to take criticism» [8, p339].

As 60% of FOSS developers participate in two or more FOSS projects, and 5% of developers participate in 10 or more FOSS projects [38, p8] this indicates «(...) that there is a growing social network of alliances across multiple FOSS development projects» [38, p25]. Communication between projects can then span multiple project communities, through developers serving as “social gateways”, increasing the project’s social mass [38, p25]. This communication going through a central actor in the network between multiple FOSS projects «(...) is a way to accumulate social capital and recognition from peers» [38, p8].

As most open source projects are distributed, it is common to also have public e-mail archives, Wiki-based documentation and meeting summaries etc. Most participants in FOSS projects engage in these online discussions and use them as a central way to observe, participate and contribute in public to discussions of topics of interest. In addition to this, Scacchi also states that «these people also engage in private online or offline discussions that do not get posted or publicly disclosed, due to their perceived sensitive content» [38, p14].

Tommerholt [46] explored communication models in global software development and how they affect the development process. Over two years he observed the internal workings of the HISP network and found that there appeared to be a lack of coordination of the network itself. Tommerholt suggested that a composite communication model consisting of many kinds of coexisting communication models [46, p97] was the best fit to deconstruct the communication pattern in an organization like HISP, with different teams which may have different internal communication structures [46, p100].
Tommerholt compares HISP to the communication structure of the “standard” Free/Libre Open Source Software (FLOSS) projects, and found some similarities [46, p86]:

The communication is:

• Mostly electronic
• Open and public
• In a forum accessible by all (mailing lists, issue tracker, wiki etc.)
• Many-to-many enabled

Though, in addition to this, he found that it also operates in a different tradition, where some communication is kept in private between recipients in a one-to-few or few-to-few pattern rather than many-to-many as in the traditional FLOSS view.

For communication to be effective using the FLOSS model, he states that the physical infrastructure must be strong enough to allow effective use of tools [46, p100]. In some of the HISP project nodes this is not the case today, as power outages and slow, unstable Internet access is a frequent and common problem. He also suggests that «The model may also be ineffective if the development involves different teams with a strong internal identity» [46, p100] as the FLOSS model focus on the individual participant communicating with a community, rather than as a member of a team with own priorities and activities.

Finally he points to the barrier of communicating through a public media and the related social cost as not everyone is comfortable with communicating with a large group of people. Related to this is also the problem of communicating in a second language, which is likely to increase the barrier further. The language barrier is referred to as one of several “gaps of understanding” [46, p98].

2.5 Sustainability

Scacchi’s review of open source research, which contains studies sampling data ranging from 400 up to 40000 projects at Sourceforge.net, revealed a power law distribution common to large self-organizing systems. A few large projects have a critical mass of 5-15 core FOSS developers that lead the projects, and are again surrounded by dozens to hundreds of other contributors in more peripheral roles. These projects that sustain such critical mass, are those that gathers the most attention, downloads and users [38, p45].

Scacchi states that «Overall, FOSS systems co-evolve with their development communities» [38, p47]. This means that a project with few developers will generally not be sustainable and produce a viable system,
unless it grows past its critical size and reaches 5-15 core developers. However, the reality for most open source projects is that the vast majority of them are small, lacking critical mass and are thus unlikely to thrive and grow [38, p45].

A possible way to grow into a sustainable size could also be to form an alliance with related projects: «Multi-project clustering and interconnection enables small FOSS projects to come together as a larger social network with the critical mass needed for their independent systems to be merged and experience more growth in size, functionality, and user base. It also enables shared architectural dependencies to arise (perhaps unintentionally) in the software components or sub-systems that are used/reused across projects» [38, p37].

In the world of open source, forking of a project may occur. This usually happens at times when a minority of code contributors challenge an established project. This may be due to different goals, the original project disagreeing with the direction some developers want to move in and etc.. Scacchi suggest that «(…) projects tend to embrace incremental innovations such as evolutionary mutations to an existing software code base over radical innovations» [38, p46] due to radical innovations often needing to be maintained in a separate version. A separate version means a potential loss of critical mass of other FOSS developers. He thus concludes that incremental mutations tend to win over time [38, p46].

To reach and sustain a critical mass as developers join and leave a project, it will need to take care of newcomers. Ducheneaut confirms this: «Socialization of newcomers is particularly crucial for the Open Source movement. (…) as successful OSS projects mature, their technology grows more complex and only a few people who have been actively involved in their development fully understand their architecture. If these key members were to leave the project would eventually falter. This makes socializing new members an essential ingredient in the long-term survival of OSS projects» [8, p326].

So far, the focus has been on a sustainable developer community. The concept of “critical mass” is however often used in relation to users, as a significant factor of network growth. In the open source setting of health where, in most cases, the developers does not equal the users of the system the community of users is equally important to the sustainability of the projects. Hanseth et al. argue that in addition to the size of the network, one should also consider the heterogeneity of its elements [18].

Hanseth et al. state that in large-scale, networked technologies like telecommunication, their «(…) value for each user increases with the total number of users that are using the technology» [18, p3]. This means that as the network grows larger, the technology catches momentum and starts growing through a self-reinforcing process [18, p3]. Hanseth
et al. says that in the case of telemedicine networks, richer models of the critical mass model is needed as users, developers and institutions are not equal in this regard. They continue focusing on the heterogeneity of the user community. This equality-mismatch also occurs in the health network setting. Hanseth et al. suggest that to reach critical mass, one should identify the users being willing to adopt the technology first, and then those willing to adopt to it as second, and so on, and enroll these first [18, p5-6].

Hanseth et al. suggests bootstrapping the network to reach the critical mass. They suggest that this is easiest done by starting with «(...) motivated and knowledgeable users who possess the necessary resources. A use area that doesn’t depend on a large network, which is low in complexity and criticality, and which doesn’t require radical organisational change will be an optimal starting point» [18, p10].
Chapter 3

Method

This chapter presents the research method utilized during my research and a description of my approach.

3.1 Action Research

The research approach applied for this thesis is within the action research framework. Action research is an iterative process where the researcher and the organization collaborate towards a common goal, and is suitable for real world situations. Knowledge gained during the process cycle is applied, thus linking theory and practice.

Action research can be divided into a two stage process [4]:

**Diagnostic stage** Collaborative analysis of the social situation => theories formulated.

**Therapeutic stage** Collaborative change experiments => changes introduced and the effects are studied.

Traditionally the researcher is placed outside the organization being studied, not taking active part in the project studied. The method tries to ensure an objective study. Action research takes a more active approach, integrating the researcher and the organization being studied.

Within the action research framework both qualitative and quantitative methods can be used. The context of the project determines which methods are most suitable in any given case. Action research has three key elements that must be present in the research [17]:

- Research
- Participation
- Action
Susman and Evered's five-phase model is used by Baskerville in the action research iterative process [4]. Together the stakeholders in the process (researcher and organization) will apply the following steps:

**Diagnosing** Identify primary problems that cause a desire to change.

**Action planning** Specify methods suitable to improve the identified problems and the targets for the outcome.

**Action taking** Implement and apply the methods, make changes.

**Evaluating** Identify whether the methods applied relieved the problems.

**Specifying learning** An ongoing process of gaining knowledge from the previous steps.

### 3.2 Action Research in the field of information systems

Action research has emerged as a research method in the information systems field after initially being introduced in social and medical sciences in the mid-twentieth century. In the late 1990s action research began to grow in popularity with information systems research [4]. Action research has gained acceptance at the same level as quantitative studies [2]. MIS Quarterly reported a change towards qualitative methods by the mainstream of researchers in a special issue on “Action Research in Information Systems” [34].

Qualitative research is largely exploratory, while quantitative research hopes to be conclusive. To gather information, qualitative research usually rely on four methods [29]:

- Participation in the setting.
- Direct observation.
- In-depth interviews.
- Analysis of documents and materials.

For action research to succeed over time a certain scale is necessary. For the target organizations to find the research valuable it should cover their needs at a useful/realistic level. Small projects with a time limit, limited focus, pilots or projects supported by donor organizations may not be valuable for the target organization and sustainability of the project may thus be a problem after the research period is over [34].
3.3 Research approach

I have been involved with two open source software development organizations. Different roles within these organizations over time is useful as the roles as student/intern, developer and teaching assistant have provided me with different perspectives on the projects I have taken part in.

3.3.1 HISP

The HISP project has traditionally used the action research method. As action research is based on participation, all HISP master students in Oslo have been working in the field for a period. This has been practiced to both improve the software, gain experience with it deployed in the field, and to support research towards a degree.

The HISP node in Oslo is developing a new version of the DHIS software, the DHIS 2. This work is coordinated by two PhD students in Oslo, working closely with other coordinators in the HISP network and implementers in the countries where the software is being used. Most of the development has occurred in the Oslo node of the project, but other networks in India and Vietnam has contributed with both local modifications and increasingly also core components.

The author has been working with the DHIS 2 project in various ways since taking part in a course at the University of Oslo related to open source software development and taught by core HISP members. As a participant in the university course, I looked into reporting and GIS tools suitable for the project.

Later on; I returned to the project as a developer working in Oslo. Eventually, I took on both development and deployment tasks as I went abroad to work with the Vietnamese team for 9 months.

As I returned to Oslo, my role then evolved into becoming a teaching assistant in the mentioned open source software development course, working with newcomers with a varying level of developer experience, and generally little experience with open source development.

3.3.2 OpenMRS

I have also been involved with another open source project during this period. A project named OpenMRS (Open Medical Record System) formed in 2004 operates in the same health informatics field, but is complementary to DHIS as its focus is on medical records rather than DHIS’ focus on reporting.

I originally applied to Google Summer of Code 2007 (GSoC) hoping to work on integration of these two complementary projects. However, I
ended up as an intern on OpenMRS working on synchronization of OpenMRS servers with a small group of experienced developers from OpenMRS. This work allowed me to experience the role as student again, but in a separate project with more experienced and partly full-time professional developers.

Working alongside the other accepted students, communicating frequently through both chat and mailing-lists gave me good impressions and first hand experience of how this organization in the GSoC setting was mentoring students compared to the HISP project.

I have later continued lurking around the OpenMRS community, observing the continued development on the project mostly from a distance, hoping to continue my support to the project when time allows. Over the course of a year I have been monitoring the development as both an integrated part of the community, as well as in a lurker-like role, giving me a basis for understanding how the project works.

### 3.3.3 Training

Training was an ongoing part of the field work in Vietnam. There was cases of training end users at the district-level together with fellow students and local project members in Hue, and together with the developers in Ho Chi Minh City. Both cases required close cooperation with the local team, as they often had to provide on the fly translation.

While working with the developer teams in Vietnam, we held informal training sessions when needed. In some cases the local team had more knowledge and experience, and in other cases the Norwegian students possessed more experience. When a new employee was hired, I was assigned to train the employee in the usage of the DHIS 2 system, communication tools, software development tools, related frameworks and Java development on the project over a one month period.

Back in Oslo, I worked as a teaching assistant on the open source software development course. This work involved training of master level students in software development tools used on the project, as well as frameworks and methods used in the DHIS 2 project.

### 3.3.4 Development and participation

I worked closely with both the other Norwegian students on the team as-well as the local employees with development. Requirements from users were discussed both locally and globally. For long periods we were working on different tasks, but discussions within the team were frequent. Timezone issues made the local team valuable and the core developers in Oslo less accessible. Local problems was preferably solved locally, fol-
lowing the action research method of cooperating closely with the locals and getting their feedback and opinions quickly.

3.3.5 Meetings

During the field work, formal meeting between project- and local representatives was held. These meetings concerned our presence and work focus for the first period. We also had a meeting where the software was presented and the possibility of developing some requested functionality slightly outside the scope of the DHIS 2 software was discussed. These meeting provided experiences related to both politics as well as local needs and requirements.

After returning to Oslo, less formal planning meetings were attended, where developers discussed progress, technical issues, specific module issues and plans, and which features to focus on.

Similarly, less formal meetings were frequent in OpenMRS as the development team working on the synchronization functionality used it to synchronize work, discuss progress and issues. A formal code review meeting was also attended with our code as the subject. This provided experiences related to a much more formalized software development environment than what the DHIS 2 project practices.

These meetings have given me new knowledge on how the organizations work internally, as well as technical and domain knowledge necessary to work on the systems.

3.3.6 Interviews

Informal interviews in the form of questions to both private and open mails on mailing lists, the use of instant messaging software, voice/video conference tools and normal informal conversations has been used to gather information. Day-to-day communication gave valuable information and impressions of situations and happenings. A focus on working closely with the projects through all phases has made this a natural approach.

3.3.7 Observation

Action research does not usually use observation, as the researcher is supposed to work closely with the project. Still, observation has been used to study other participants’ progress and the open communication between them and the organizations.
3.3.8 Possible method-related limitations

The intense participation in the projects, and especially on the DHIS 2 project, sometimes makes it hard to “step out” and view events in a larger context. In general, the Vietnamese people and team have great respect for western foreigners, and as such may have influenced them in ways that were not supposed to happen.
Chapter 4

Empirical material

4.1 Background

4.1.1 HISP

HISP is introduced in the introduction found in 1.1 on page 3. Previously the project and its history has been covered in detail in several articles and theses. Tommerholt [46, p36-47], Nordal [31, p27-35] and Overland [34, p21-24] cover the HISP network, and the various software produced by the HISP network. Tommerholt also introduces the basic domain of the DHIS software, and gives an overview of the nodes in India and Vietnam [46, p39]. Braa et al. [7] describe the start-up and strategy of the project in Africa. They focus on the need to organize networks of action, scaling and spreading the health information system approach into several nodes in order to attain sustainability.

4.1.2 OpenMRS

OpenMRS is introduced in the introduction found in 1.1 on page 4. The history of the collaborative effort OpenMRS is covered by Mamlin et al. [28] where they explain how OpenMRS can serve as a foundation for electronic medical record development in developing countries. The basic design of OpenMRS, the problem it attacks and how it attempts to solve it is described briefly by Wolfe et al. [50]. Seebregts et al. [39] describe the work and progress of the OpenMRS implementers’ network.
4.2 DHIS

4.2.1 INF5750 - Open Source Software development frameworks in global networks

The INF5750 - “Open Source Software development frameworks in global networks” course at the University of Oslo was my initial point of contact with the HISP project as I signed up for the course in the autumn of 2005. It is a course aiming at providing students with real-life coding experience using software development frameworks that are up-to-date. The course uses the HISP project and the DHIS open source software project as the basis for the course. The initial focus is on the student's understanding of modern software development and frameworks, teaching them how to correctly use version control software to keep track of changes in code on a software project. Modern build and project management tools are used for organizing the code into different semi-independent modules that can be combined and built in different combinations as needed. Current frameworks for persistence management, dependency injection of objects, transaction support, and presentation layer frameworks are all introduced as well.

Working in groups

An important part of the course is the focus on group work. Students are divided into groups depending on their interests, and work together for about 3 months on their assignments. Many of the software requirements come from abroad, and with all the users of the software located mainly in Africa and Asia, and developers situated in many of the same countries many of the groups will have to communicate with their counterparts around the world, providing some real life experience with global software development.

The group I was part of was assigned to a GIS task. Visualizing data on a map is important for analyzing data and reporting. The group based its work on another open source project, uDig, which again is built upon the Eclipse framework. The group decided to try to develop a plugin for uDig so that interaction between uDig and DHIS 2.0 would be possible, utilizing the data from the DHIS 2.0 database, preferably through the DHIS 2.0 API. Several institutions provide satellite image data for free, e.g. NASA. These could be used in combination with other digital maps of relevant areas. Layering our existing map data with satellite images turned out to be slow and CPU demanding. Also the integration of the data into uDig turned out to be hard to get working in an acceptable way in the limited time we had available. We eventually were able to provide a demo where an organization unit could be selected and highlighted.
together with some data, but coloring the different shapes on the maps turned out to be harder than expected as the necessary functionality was not implemented in uDig at the time.

The work on uDig was separate from DHIS 2.0 itself, meaning that we did not get as much experience with the Java frameworks used in the main DHIS 2.0 application during the group work part of the course. The course did however provide initial training in some of the relevant technologies through mandatory assignments in the first part. These covered some basic unit testing, version control, basic dependency injection with the Spring framework and the Maven project management tool. The assignment used the standard 3-layer architecture of the system while we did development on the data access and business layer. The graphical user interface was provided to us as a Swing based application rather than the web interface that DHIS 2.0 uses. As the focus of the course was open source Java frameworks and a tight schedule for the course limits the number of technologies that can be successfully taught, this was a good approach. Most of the groups working on new modules would then learn the frameworks used in the user interface during their group work, while also extending their knowledge at the data access layer with Hibernate rather than with the mock-up in-memory data access layer as used in the mandatory assignments.

During group work, and project work in general, when I worked closely with others over time, I experienced that project participants tended to exhibit very different working styles. Some developers fixed problems they found themselves, other developers preferred to simply report them, even though they had the competence to fix them and suffer from the problem themselves, while some developers presented suggestions for redesigning the solution completely rather than doing a simple fix, or suggested features that were outside the scope of the project. This last type of project participant, the “visionary”, usually ended up either giving up, or, on some occasions, start developing the solution outside the project, and promote his solution over the existing one. The “contributors”, on the other hand, usually fixed any problems found themselves, without causing major fights. In my experience, the core developers and other regular developers fit this category. The last type of developers mentioned, the “non-self-going”, have in general not been found in the project for any substantial time.

### 4.2.2 Import/export

#### Case

The DHIS 2.0 application is a very data-centric application where the collected data is supposed to be available for aggregation upwards in the
organization hierarchy. As the software is designed as a web application that can be installed on a central server and used from many different locations, synchronization/import/export of data between servers could theoretically be avoided. Unfortunately using the software on a single server is problematic in most real life cases for various reasons. A country-wide setup on a single server covering all levels in the public health system requires significant processing power and bandwidth. As the DHIS 2.0 software is mainly targeted at countries where important infrastructure for such a scenario like electricity and Internet access are expensive, has limited availability and often are unreliable; there is a significant need for methods to synchronize data between servers. Another target for import/export functionality would be to be able to upgrade and collect data from older Microsoft Access based DHIS 1.3 and DHIS 1.4 systems. As Internet connections was not available at all places the import/export functionality had to support different types of sending and receiving data. In the case of DHIS 2.0 this was a file based system where the users themselves had to handle the transmission of the data through e.g. e-mail, USB drives, CDs and the like.

Solution

Together with another developer on the project located in Norway we started developing a system for handling exporting and importing of data from one instance of a server to another. Over time, we split the work between us so that the other developer focused on export, while I handled importing of the data. Both parts of the system were developed with plugin functionality for easy addition of new export and import formats at a later time for usage with other related software e.g. for data analysis. Significant time was spent working on performance issues on both import and export. As the data amounts can be significant, even if it's done e.g. monthly, the amount of data collected can consist of millions of values. A test data set used as a benchmark based on a single state in India contained about 1.8 million data entries - which caused both problems with timeouts in the system, and the software running out of memory and crashing. A lot of research went into solving memory constraints by streaming the data to file systems, compressing the data on the fly and keeping sessions alive. Since none of us had developed the core systems handling web requests, session, transactions and timeouts, this was a big challenge.

Similar problems was faced with the import code. Limits to file size of uploaded files had to be changed, and lots of work in the end went into making the import code stream objects straight from the uploaded file and through the matching and validation of imported data. This was a valuable experience, where some interesting workarounds had to be
written to be able to iterate streaming objects while keeping file access open without timing out sessions. Also on the multi-user side of things this was a challenge as the upload was not synchronized between different concurrent users. In most cases import would be a “blind” operation without user interaction. In the case of collisions between data with differing values, and in the case of someone wanting to manually control and possibly edit data during import through the user interface this became a challenge again - especially on the large use cases this would be very hard to get working properly in the combination with streaming objects. After months of work, that part still did not perform acceptably. Later, most of the import/export process was replaced by a solution that stored objects temporarily in the database after uploading the data and then processing them later.

Trying to solve these performance problems was however a very valuable experience, as we learned a lot from each other on optimizing and evaluating performance. This work was performed in a traditional open source setting with two peers located separate and in different time zones. We solved this by using asynchronous communication tools like the mailing-lists, as well as working long nights for one of us to align our working time better and utilize instant messaging tools.

4.2.3 Teaching assistant for the Open Source Software development course

The core members of the HISP network organize and run the Open Source Software development course at the University of Oslo. The course teaches core frameworks in the Java world like Hibernate, Spring, JUnit, WebWork, Velocity along with version control software and build/management tools. The work as a teaching assistant consists of different types of work in this course. One of the main work tasks are evaluating mandatory assignments from students and giving feedback on their work, focusing especially on explaining the parts of an assignment the student might have gotten wrong. In the first part of the course there tends to be a lot of new frameworks and tools to learn for most students, and giving basic “crash courses” into the practical use of tools is one of the main tasks to get the student up and running as quickly as possible.

As the individual part of the course ends, the students are divided into groups to work on new features, research or other tasks suitable for a group of students. In that phase the teaching assistants’ work is mostly to support the groups in their work, answering questions, helping spot bugs and point out possible routes to go forward to solve a particular problem. As the course is used as the main recruiting area for new developers on the project from Oslo, keeping an eye out for students with an interest for the project and connecting them more permanently
to the project is important.

4.2.4 Documentation

Two of the DHIS developers started a project where they aimed at creating and maintaining documentation for the DHIS 2.0 software. The documentation is located at the HISP/DHIS Wiki site. It is probably not a coincidence that this project was started by developers who were not part of the project when it started, but joined it later on and did not have the overview of the system enjoyed by the initial developers.

The following goals was set for the documentation project:

- To provide a clear and unambiguous access to documentation for users, administrators and developers.
- To provide a clear and unambiguous distribution of responsibility for documentation.
- To organize the maintenance of documentation.
- To evaluate and implement tools for documentation.

The documentation was divided into different groups for different target users/goals:

- Users
- Administrators
- Developers
- System documentation
- Mailing lists

Most of the work was done over a relatively short period of time. After those developers left the projects the work more or less stopped even though several parts of the documentation is still lacking. During the period documentation was frequently added and updated I did however notice that one change/update/addition often triggered other changes. The reason for this I believe is that e.g. changes listed on the developers’ dashboards on the website and non-mandatory e-mails that some of the developers had signed up for with summarized lists of changes triggered more changes. Whenever some of the documentation was updated, people was checking what was added or changed, and contributed with corrections, additions and comments to the material. This work does seem to need a certain amount of traction and interested developers to be maintained over time. The OpenMRS project seem to have
understood this even more than the DHIS developers as they are aggregating all changes/information into a single RSS feed where developers are able to filter the information they are interested in and stay up to date without actively having to monitor dashboards, mails, forums and mailing lists. This is probably an area where the DHIS project could learn from OpenMRS, as the information is more spread out, and interested parties have to sign up in several different systems/mailing lists to stay updated on all relevant changes.

4.2.5 OSHCA conference in Malaysia

The Open Source Health Care Alliance (OSHCA) is a non-profit organization that provides a collaborative platform and forum to promote and facilitate Free/Open Source Software in Health Care. In May 2007 OSHCA organized a conference in Kuala Lumpur, Malaysia where HISP was represented. During the conference many different projects and companies held presentations of their software and projects. At the time I was not aware of most of these projects even though they operate within the same field of IT and health.

The conference was organized into several different tracks where the attendees could move freely among the tracks to take part in the most useful presentations from their perspective. Some of the organizations had technical presentations of their software and quick introduction courses aimed at developers into the programming languages used, like Ruby on Rails. Baobab's presentation of their special modified hardware aimed at health facilities in Malawi was most interesting, along with the possibility of playing with a prototype of the One Laptop Per Child (OLPC) computer. Other presentations covered legal issues with software and public health.

I attended a lot of presentations of different projects and software systems operating within the same field as the HISP project, and as most of these projects were unknown to me I found it very useful. Speaking to and hanging out with other developers in the same situation from around the world truly made this both a useful and inspiring conference. I have later stayed in touch with a handful of these contacts as the cooperation between DHIS and OpenMRS and other projects in the OpenMRS sphere has increased. Contacts made during the conference was especially useful in the early stages of my Google Summer of Code project work, as I then already had established contact with some of the community members surrounding OpenMRS.
4.2.6 Fieldwork in Vietnam

Before the fieldwork in Vietnam, the other students and I had limited experience with DHIS 2. We had all attended the Open Source Software Development course (see 4.2.1 on page 33) and through the different projects we were assigned to, we had gained various experience with the software. However, in my case, most of the work involved third party software, and in general we did not have extensive knowledge of the inner workings of DHIS 2.

For the first two weeks of the stay, one of the HISP coordinators joined us to attend meetings with local officials involved in the implementation of the project. When we arrived, we had already talked a bit with some of the students that had worked on the project in Vietnam before us. However, we did not know the details as to what they had achieved or what they worked on during their stay. A minimal introduction to the project’s history in Vietnam was given by the coordinator. At the time, we knew little about the previous cooperation with a consulting company in Vietnam, as well as the cooperation with one of the local universities and the lectures that had been given there on previous visits by Norwegian students. The focus during these first two weeks was mainly on practical issues related to travelling, office locations, and being introduced to the local developers and implementers at the two locations in Vietnam where the project had a presence.

Distributed development

With distributed development several issues naturally appear. Coordination and communication barriers increase as distance, time and possibly language becomes issues. During the fieldwork in Vietnam I was mainly working on the DHIS project with developers located in Norway, India and Vietnam. Mailing lists on the project are supposed to take care of most of the communication internally on the project, but the limitation becomes apparent when response is needed quickly. Due to different time zones communication takes time, and may slow down work significantly as developers with the right knowledge might not be available at the time of need. That was especially a problem during the second half of my stay in Vietnam, as the developers working from Vietnam did not always know what was needed to solve a particular problem. In the first semester that was less of a problem as we were more developers with the same background working on similar issues. The issues with time zones was more or less solved by changing sleeping habits and staying awake late at night while sleeping in the morning local time to stay closer synchronized with the most relevant developers. As the Google Summer of Code project started in the end of my stay in Vietnam this became
even worse to manage as mentors for the project was located in the US.

**Local teamwork**

**Facilities**  As soon as the team arrived in Hue, we started working with one of the local developers on the project on a daily basis. The plan was to set up office at the local health ministry headquarters and work closely with the mid-level users there as we continued to develop and implement the software in the districts reporting to the head quarter. In reality, this required authorization from the central ministry of health as allowing foreigners access into the offices of the health service was looked upon as sensitive. This access was first granted about 3 months into the projects, out of the 4 planned. In the meantime we had set up office at our hotel and kept working out of our rooms there. This solution did not give us the close relationship with the local users we had hoped for, but the local developer came in on a fairly regular basis several times a week to our hotel-office and worked with us there. As the security clearance was given, some of the work was moved to the local health office but as both internet access and electricity was even more unstable there than in the hotel, and there was no air condition in the office, it unfortunately turned out to be more productive to work from the hotel.

The initial plan was to get DHIS 2.0 up and running in the 9 different health facilities around the district. As transportation had to be requested from the local health service, this turned out to be too much in the beginning, and it was narrowed down to the 4 nearest facilities where we were less dependent on available transportation.

**Tasks**  The first couple of months working in Hue on the project we were 3 developers from the Norwegian node working together at the same location, though on slightly different tasks. My initial focus was on converting the existing DHIS 1.4 database into the current DHIS 2.0 milestone database. As there was no such existing tool at the time, only a few outdated scripts, much time was spent on experimenting with importing and exporting data on different formats and in different ways. As the Vietnamese alphabet uses some different characters it was important to find a method that retained the correct text without losing information. Another major reason for finding a recipe for this was that the DHIS 1.4 would still be running at the same time in other district, thus this was not a one time operation but one that would most likely be done several times in order to keep the databases updated until all the locations was running the same, compatible, version of the software.

One of the other developers working on the project in Vietnam fo-
cused on a validation module, for validating input data statistically against previous entered values, or hard coded values. I was involved in debugging it during development, and suggesting possible development paths. This module was supposed to give feedback to the users whether or not the values seemed sensible, and the idea was to avoid common typos as the users of the software sat with the paper reports, reading a column and typing at the same time.

**Approaches to learning**  As all of us were fairly new on the project at that time, we tried to help each other out with the tasks, learning from each other and trying to understand how everything was working by studying code from other parts of the projects that performed similar things. At the time there was very little updated documentation, and the group of developers that had developed most of the software was all very much into the code and how it all worked. For us as new developers on the team, with relatively little experience with the technologies and frameworks used on the project, this was challenging. Due to the time zone differences access to experienced developers was not available until in the evening and during night-time.

In the beginning of the stay the three of us were assigned different main tasks to work on. As time went on we worked more closely together. The reason for this was that none of us felt that we knew the source code base well enough, and thus ran into different kinds of problems frequently. We spent significant time trying to understand how the different modules, frameworks and classes interacted together. Especially the developer that was assigned to work on data validation and I experienced that working together, trying to understand how the different layers worked, and where and what to change was a better path to go than struggling mostly alone. Especially as we seemed to have a good understanding of different parts of the system, we complemented each others knowledge well. In some periods much of the coding was done as pair-programming on one computer, working on one task at a time.

**My approach to learning**  The approach I took to understanding how DHIS 2.0 worked internally was to begin in one end with a limited number of technologies - preferably similar to technologies I had previously used. In my case I chose to go through the web interface layer and validate the code using the HTML and CSS validation tools made available by the World Wide Web Consortium (W3C). These tools parses the code and reports on sections breaking the standards. A common problem in many web applications is that developers tend to not spend enough time validating their code, but trusting that as long as the output generated by the code looks correct, it probably is. I spent quite some time
systematically going through the web interfaces in the various modules, locating and tracking down standard violations and then trying to solve them. Fortunately, many of the errors and warnings reported were similar and could in some cases be fixed by a simple search & replace. Other sections required larger structural changes and logic. Especially on modern web pages this can be more tricky than it sounds, as the code in the browser can be manipulated after it’s sent back from the server, meaning that new errors can be created after the page is sent back from the server. Graphical user interface testing frameworks exist, but so far haven’t been prioritized. Unfortunately, manual testing requires more developer resources than standard unit tests that are written once and run often, automatically. The fact that various browsers tend to implement certain features of the web standards differently also makes this complicated. On some occasions you might have to trade off the correctness of the code and rather use what works best in the actual browsers. Due to the way the web interface is designed, we have seen issues, especially with Internet Explorer 6 and the layout based on the CSS standard. Internet Explorer 6 does not display DHIS 2.0 correctly, and the use of the browser was officially discouraged due to some major layout rendering problems. Another part of the system where such problems was experienced was with the JavaScript implementations. As users in the field in Vietnam requested a way to enter data faster, we chose to implement a feature that automatically moved to the next input field when the users hit the Enter key. In this case, the JavaScript code to handle the input keys and change focus to the next input field had to check for which browser was used, and identify the Enter key correctly based on that.

By starting with a very limited number of technologies involved, and then moving down layer by layer and framework by framework I found it easier to understand how the application worked technically. As HTML and CSS were well-known technologies to me, I could then focus more on JavaScript and the code that generated the output containing these three technologies: Velocity. As Velocity is a fairly simple tool for combining Java code with e.g. HTML output, it was then natural to focus one layer down, into the Java business layer, and how Java, Velocity and the output was combined by the WebWork configuration files handling the flow in the user interface. An advantage of approaching the project with a limited number of frameworks, and from one end to the other is that the developers can start to deliver useful code and bug-fixes at an early stage in the process, compared to starting out doing work that requires changes in all layers in the system, which would require understanding of many more frameworks and their interaction before you’re able to deliver useful output to the project.

Another approach that I have used previously and found efficient is
doing the opposite, learning one and one technology from scratch thoroughly and combing them in a simple application separate from DHIS 2.0. This approach would probably have increased my understanding of the different frameworks at an earlier stage, but would most likely require a lot more work and a good, simple test application where all the relevant frameworks could be tested together. The disadvantage is that the code generated is not immediately useful to the project, but would probably leave the project with developers having a better and more complete understanding of the frameworks and their interaction. This might be valuable for a project as the developers will be more likely to be able to architect good future solutions.

The learning experience in the DHIS 2 project differed significantly from what I had experienced earlier. The DHIS 2 project was already in heavy development at the time I got involved with it. A large number of frameworks that I was unfamiliar with, as well as a different code style and code hierarchy made it a challenge. In all earlier projects I had been working on, I was involved from the beginning, and thus knew the working details and technology involved. In addition, few of them had a scope similar to that of the DHIS 2 project.

Training local developers

During the stay in Vietnam we shared offices with the local team, either at a hotel-based office, at the local health service or in one of the hospitals. Most of the local developers working on the project had already received training from other Norwegian students in the form of lectures at their university and through their day-to-day work at the office and knew most of what was needed to perform their jobs. However, some Vietnamese developers were not co-located with the Norwegian students who had visited previously, or joined the project after the Norwegians left. The training of the developers conducted during our stay was performed on an on-demand basis. This training was not only provided by the Norwegian students, but also by the Vietnamese in fields where their skills were superior to ours, e.g. in software related to designing reports.

Both through work and training we experienced that language was a barrier. The working language on the project is English, however, often both the Norwegian and Vietnamese team spoke in their respective languages between themselves. This was discussed in the project group, and we tried to stick with English in most cases as to maximize the learning effect from the stay for both teams. However, in some cases using English was too slow, limiting our performance and understanding, so it was practiced more as a rule of thumb rather than strict, enforced rule. In the second part of my stay it was practically impossible to enforce it, as some of the developers hardly spoke any English at the time.
During our stay, it became apparent that the Norwegian and Vietnamese educational systems differ significantly. While the Vietnamese educational system focuses more on the individual student and learning “facts”, the Norwegian system focuses more on problem solving and group work. In open source software development one usually has to solve practical problems, and thus one of the challenges was to get the teams to work together as a group and find solutions. An example of this was the way a new report was added to the system. Rather than making a solution for dynamically locating the right report out of a number of possibilities, the Vietnamese team had chosen a very straightforward solution, where new reports were continuously added to a list of reports, hard coded in the system. This caused extra work as the system had to be recompiled and changed frequently as new reports were added. No one had seemed to notice that a small change in the code could save a lot of work.

**Training end users**

On several occasions the team in Vietnam, including both local and the Norwegian developers performed end user training sessions. This was performed in their own work setting, training a single user at the time over a period of time. As the end user performed testing of new functionality, the training was split up and updated as needed, and feedback from the user brought back for the developer community to improve on.

Group session training was conducted for larger groups as well. On another implementation site, the local development team held hands-on training sessions for a group of approximately 15 users gathered from hospitals around the implementation site. This training session lasted for a full day. Due to a limited number of computers available, users worked in groups, rotating the tasks.

**4.3 OpenMRS - GSoC**

I was introduced to OpenMRS through Google Summer of Code. I had previously heard about the application, but my knowledge about the project was limited. As OpenMRS complements DHIS 2 in terms of functionality, and in many cases potentially contains the data the aggregation in DHIS uses, it was in DHIS’s interest to integrate closely with OpenMRS. Similarly, OpenMRS lacked the aggregation and reporting functionality offered by DHIS 2. I was encouraged along with several other developers to apply for a Google Summer of Code position on the OpenMRS project. The initial application detailed the suggested integration of the two projects, as proposed by one of the HISP coordinators involved with
DHIS 2. As OpenMRS received a large amount of applications they did however prioritize functionality more relevant to their core functionality, and thus the application was not accepted.

I was then asked if I would be interested in working on one of the other projects outlined by OpenMRS, namely synchronization functionality. This project overlapped partially with previous experience I had from handling import/export, and I accepted as that would also increase HISP’s knowledge about OpenMRS, with the hope of easing the possible integration of the projects at a later time.

At the time when the Google Summer of Code working period officially started, I was still located in Vietnam. My mentor was located in the USA, along with many of the other developers and core OpenMRS people. The beginning of the period was spent researching serialization frameworks and solutions to various problems related to synchronization, for instance time-stamping.

Consisting initially of my mentor and myself, the team grew in size over the summer, as the size of the task and ambitions increased. The planning and development was performed mostly by a 4 person strong team. Due to the size of the project, late start on the coding, and slower progress than expected, the project continued beyond the schedule of the Summer of Code program. At the end of the period we did have functionality that more or less worked on simple use cases. However, another 6 months has been put into the development by parts of the team, and after almost a year the functionality seems to be almost complete and stable enough to be deployed for production use.

Most of the Summer of Code interns along with many of the mentors and developers used Internet Relay Chat (IRC), mailing-lists, wikis and blogs to communicate during the development. Internally on the synchronization project, as well as on the weekly development meetings where mentors and core developers usually were present, instant messaging software and voice-over-IP software were used to communicate. The synchronization team usually had voice-chats 1-2 times a week during the development and conducted other communication as needed.

The Google Summer of Code program utilizes a mentor-based approach to the projects. All developers are assigned a main mentor and usually a backup mentor as well. The idea is that the mentor should help the project along, provide domain knowledge and give feedback on code and progress. In the case of the synchronization project the mentor was heavily involved in the development along with several other developers. There was little mentoring at the code-level. However, the shared domain knowledge and experience with synchronization, planning and development on fairly large projects was the most useful outcome of the mentoring process for me. Working with experienced developers was challenging in the way that one does not necessarily feel competent to
voice one’s opinion. In addition, it was most useful to be part of a project where there was a real need for planning and thinking things through, communicate and synchronize development over several time zones in a truly distributed environment. In addition to the mentors, core developers, developers and other interns were available for each other to answer questions on mailing-lists and through instant messaging software.

Out of the interns accepted by OpenMRS, some have stuck around the OpenMRS community, though most of them are no longer taking active part in the development today.

4.4 Licensing

4.4.1 DHIS and licensing

The DHIS project is licensed using the new BSD license, meaning there is no copyleft and that it may be used in a proprietary setting. The reasoning behind this is that the DHIS core software was supposed to be very general and not only fit the health setting where the DHIS project itself is located. The core of DHIS could possibly be used in any setting where you need to register data on different levels in an organization hierarchy with aggregation possibilities. Some of the early developers might have seen other possible usages for the DHIS core software in other settings at a later stage, and thus wanted to keep the possibility of using the software in proprietary settings later open [32]. The DHIS software is developed mostly in an academic setting where openness and sharing of resources have long traditions.

DHIS developers discussed licensing on the mailing list before deciding which license to use for DHIS 2.0. A lead developer [32] raised several questions on the mailing list regarding the legal issues involved with the different libraries and bundling the software with other libraries used by the project with several different licenses and argues for a liberal BSD license. A coordinator [44] argued for liberal licenses like the BSD, Apache License or MIT License, pointing to the fact that a lot of the DHIS code is very general and may serve other purposes outside the public health domain. The South African based project leader of the DHIS 1.3 and 1.4 software released under the LGPL license, argues that «my opinion is that we should move to using the "full" GPL, i.e. that we expect derived products to also be open source’d» [20]. One of the lead developers points out that few people were involved in the discussion, observing that «Anyone who wish to get a good overview of the alternative OSS licenses for DHIS2 need to put a lot of effort into it.” and that «Only a couple of people got involved when mails on the matter were sent to the mailing lists. Those of us that got involved had a relatively good under-
standing of the different license models, but we mainly represented the Norwegian development team. There was an attempt to get more people involved, when an overview of the different licenses was posted on the development mailing list, but it got little feedback» [31, p61]. This observation aligns well with the position I found myself in at the time, feeling that I didn’t have enough experience in the project and knowledge about licensing to provide well thought through feedback and opinions on the matter.

4.4.2 OpenMRS and licensing

The OpenMRS project has chosen to create a legal organization to hold the rights to the software produced by the developers working on the project. The legal organization behind OpenMRS, OpenMRS LLC. has chosen to license its code under its own license, the OPL - OpenMRS Public License [25] despite the general wish from OSI to not create any duplicative licenses [22], but rather try to find a suitable pre-existing license. The OPL is based on the Mozilla Public License 1.1, and the changes from the MPL are summarized in section 6.3 in the OPL. The changes applied are among other things trying to specify in more detail who the initial developer was, and removing the MPL 1.1 section 13 - “multiple licensed code”, removing any possibility of portions of the code being made GPL compatible under the current license. FSF states that the MPL «...has some complex restrictions that make it incompatible with the GNU GPL. That is, a module covered by the GPL and a module covered by the MPL cannot legally be linked together. We urge you not to use the MPL for this reason. However, MPL 1.1 has a provision (section 13) that allows a program (or parts of it) to offer a choice of another license as well. If part of a program allows the GNU GPL as an alternate choice, or any other GPL-compatible license as an alternate choice, that part of the program has a GPL-compatible license» [11]. However, they are still be able to license the project under another license themselves as the copyright owner.

Like the MPL, licensing OpenMRS under the OPL license allows any third-party developed modules or extensions may be kept proprietary as the viral nature of the OPL only covers what is defined as modifications, which in this case are modifications or code based upon code already licensed under the OPL. Any new files containing only code not based on OPL licensed code may then be kept under a separate license [26]. OpenMRS tries to keep track of all the licenses of involved third party libraries in its lib.properties file [27].
Chapter 5

Discussion

This chapter presents my findings related to the previously introduced literature on socialization and sustainability in OSS projects.

5.1 Recruitment

The DHIS project is to a large extent developed by students working on the project as part of their master thesis. Over time, students leave the project and new students become involved. Ducheneaut points out that «Socialization of newcomers is particularly crucial for the Open Source movement. (...) as successful OSS projects mature, their technology grows more complex and only a few people who have been actively involved in their development fully understand their architecture. If these key members were to leave, the project would eventually falter. This makes socializing new members an essential ingredient in the long-term survival of OSS projects» [8, p326].

The DHIS project tries to transfer knowledge between the developers by recruiting new student developers through a university course. The course taught the basic tools and frameworks used and involved the students in the development and research of new features at a very early point, there’s little time for the “lurker”. One of the ideas was to find students with an interest in the project and with technical potential to take part in the project as a regular developer and over time evolve into a core developer. Rather than letting the students locate the project themselves and start providing patches there was a more active approach to recruitment of potential developers than the typical open source project unable to “force” testing of new potential developers through a course setting. A fairly stable group of core developers and project managers provide stability and domain knowledge that can be shared with the newcomers. The open source software development course has been the main recruitment base for the DHIS 2 project, in addition to developers hired by
implementation nodes. In traditional open source software, developers are often recruited from the user base. In the case of vertical domain information systems as DHIS 2, with a limited number of implementation sites and thus users, the pool of potential developers is limited as well. This may impact sustainability (see 5.5 on page 71).

OpenMRS and their core group of professional developers does not have the same natural recruitment base as DHIS, though several universities and students have located the project and become developers of modules. During 2007 OpenMRS got involved with Google and the Summer of Code program as a mentoring organization hosting projects for students and supplying mentoring resources. OpenMRS published a list of potential project and additionally accepted suggestions for other projects submitted by students applying for a position in the Google Summer of Code 2007. The number of applications for a position received surprised the project leaders; «We are pleased to announce that we received 134 eligible proposals! Applications came from not only 13 states within the US, but also 34 other countries as well, covering six of the seven continents» [5] and 11 of these were accepted due to limitations from Google Summer of Code and mentoring capacity.

To support the socialization in the network of distributed developers OpenMRS encourages the developers to be logged on to their chat room. The chat room is used as a less formal communication form where questions can be asked both between students and core developers and where anyone can join in on the conversation with suggestions to technical discussions and informal discussions about other things than the project. This form of communication seems to lower the barrier of communication between both students and developers and is sometimes used to prepare or pre-check if a questions should rather be pointed to the official developers mailing-list for future archival and documentation of the question and response.

Using the chat channel is also useful for observing other developers interact, discovering who is working on what, learning the culture in a project and how to give and receive criticism. Ducheneaut found descriptions of this in the Python projects developer documentation: «(…) the list of active members is always shifting because developers have differing free time, availability, and interests. To work with this large and dispersed group, you’ll have to learn who’s the right person to answer a question, how to convince the other developers of the usefulness of a patch, how to offer helpful criticism, and how to take criticism» [8, p339].

An automated analysis of the DHIS code by ohloh.net\(^1\) reveals that

\(^1\)The same automated analysis of the OpenMRS code base does not seem to represent the complete picture as only a handful over developers are listed compared to
out of the 25 developers that have committed code to the main project, a small core group of 3-4 persons performed most of the actual development. These numbers were very similar to what Tommerholt [46, p59] found in his commit log analysis. The rest of the contributors provided minor modules or fixes, normally over a short time span compared to the group of core developers. This aligns with Ducheneaut and his findings in the Python project.

5.1.1 Trajectory

Developers tend to change roles in a project over time as their knowledge and interests may change. Ducheneaut [8, p349] identified 6 levels in this trajectory:

1. Peripheral monitoring of the development activity
2. Reporting of bugs and simultaneous suggestions for patches
3. Obtaining CVS access and directly fixing bugs
4. Taking charge of a “module size” project
5. Developing this project, gathering support for it, defending it publicly
6. Obtaining the approval of the core members and getting the module integrated into the project’s architecture.

Lurking

The initial level is commonly known as “lurking”: «Messages posted on the development mailing list and interviews with core developers indicated that often a significant period of observation (lurking), ranging from a couple of weeks to several months, was needed before someone felt they could contribute to the technical discussion» [47, p1227].

The lurking period can be shortened by «(. . . ) highlighting the most dynamic and controversial conversations (...), these usually reveal the underlying socio-technical structure of a project during "trials of strength"» [8, p358]. OpenMRS displays recent mails and blog posts at their front page, and in great detail on their “feed” page (http://feed.openmrs.org/) - which gathers data from mailing lists, forums, Wiki page modifications the overall community of active developers. This may be explained with the extensive use of branches, the fact that only the “trunk” branch is analyzed, and the use of the “alpha” branch as the day-to-day development branch for an extensive period. In addition, merging of branches into the main code-base is primarily done by one core developer.
and etc. into one stream of information. As much of this information is tagged with keywords - a “tag cloud” called “What people talk about” is displayed on the page highlighting the most used keywords. This makes it easy to spot which areas are the most active, though not necessarily controversial, and is an easy way to navigate through the data to find related information. In DHIS there is no such display but sorting out the longest threads from the mailing list archives are doable but time consuming. Both projects utilize bug tracking software that can be used to sort issues and requests with high priority and big impact, which are good pointers in general, but again not necessarily controversial.

Ducheneaut suggests that the long lurking periods could probably be reduced given appropriate resources, accelerating the influx of new and useful ideas into the project without increasing the core members’ workload [8]. OpenMRS, and their involvement in Google Summer of Code probably did speed up the lurking period for some of the new interns through the use of mentors and access to core developers. My impression is that the close cooperation with the mentors from the beginning, got many of the interns going quickly, and most of the basic questions was spared from the core developers.

Through the OpenMRS IRC channel and the general socialization period defined by the Google Summer of Code program, the socialization process was initiated before the official work period. This period could probably be viewed as a type of lurking in the open, getting to know the rest of the community, and a bootstrapping process to the rest of the development period. Ducheneaut suggests to use computer tools to make the project’s socio-technical structure more “readable” to aid this bootstrapping [8, p357].

In the case of DHIS and OpenMRS which recruits students/interns there’s little time spent on lurking before participating more directly in the projects. Most of the developers recruited to the DHIS project are recruited fresh out of universities, or while still at a university. A reason for skipping this step is that the domain where the project is situated is not necessarily of major interest from students. The primary motivation of students is to learn the technical side of things and practicing software development in an open source setting.

The crucial role of “starting out humbly” by contributing technical solutions to already existing problems (levels 2 and 3) before moving on to more significant accomplishments (level 4) is emphasized. The approach the associated master students take on the DHIS projects differs. Some start out by looking in the bug database containing already identified bugs and areas that needs for improvements and then start fixing those. That approach may gradually increase the domain and technical knowledge needed, as many minor bugs have limited impact on the project as a whole and are a good approach to understanding how the
different modules, technologies and frameworks works together.

Another and more common approach on the DHIS project is to start working on a specific, major need of the project. This is typically a module or major addition to an existing module. Examples of this are the integration of what was initially created as a “min/max limit module” to verify input data based on previously entered data or a set limit, and the work done on import and export functionality, rewriting the existing module and adding new formats while improving performance. These tasks are fairly major and may feel overwhelming until the necessary domain and technical knowledge is gathered. This is a natural side effect of a project in an educational setting rather than a project based on the developers personal and private interest in the project where one normally would assume work on a task you have a personal interest, and possibly domain knowledge. The DHIS setting is comparable to a business setting where your employer have a need and contributing to an open source based project is a side effect caused by the employers need.

The identified levels in the trajectory may or may not be followed, some developers jump directly into level 4 starting out on a module sized project. In the case of DHIS this is an acceptable approach, at least when it's spawned from a need identified by the project. However, Ducheneaut reports that «Most contributors stop at the bug reporter stage, or did not evolve at all» [8, p353]. This is not directly comparable to DHIS as few start at the bug reporter stage, however the finding that many does not evolve from their initial level is valid in DHIS too. Ducheneaut found a ratio of 4 out of 50 newcomers evolving to developers on the Python project. Similar ratios can be found on the DHIS project; after the initial open source software development course that most new contributors attend initially only a few keep interest in the project and evolve into a role of regular developer on the project. To my knowledge no code has been contributed to the project from outside this recruitment structure, and to a very limited degree has any contributions been made after the students finish the course, or their thesis on the project. This indicates a lack of socialization and limited personal interest in the projects domain.

5.1.2 Obstacles as a filter

The approach of not giving away all the inner workings and details of a project is explained by Ducheneaut to be used as a selection process: «(…) by adopting a somewhat distant attitude, the project leaders make sure that they do not have to constantly "hold the hand" of newcomers and waste an inordinate amount of time introducing them to the subtleties of software development. A certain amount of selection is neces-
sary, if only to allow the core members to focus on their tasks (…))» [8, p357]. Repeated questions are sought to be avoided by gathering documentation on Wiki pages with frequently asked questions along with other documentation. Referencing existing documentation rather than customizing an answer every time the same or similar question pops up is then fairly easy and less time consuming for the rest of the community, and it’s a clear hint to new developers to look through the basic sections first before asking the most common questions. These can be seen as obstacles. If the developer is not able to figure out basic things himself or search existing documentation before asking, he may not be worth spending too much time on for the project, and thus has failed a “test”: «Moreover, the obstacles put in the path of newcomers function as trials and rites of passage that are important to ensure these individuals are a good fit for the project» [8, p357].

In my experience, asking peers on your own level or other developers working on your project first comes naturally. Learning from each other is a good way for new developers to both practice their skills and socialize with other developers in the same situation. If a group teacher or mentor is available, such resources should be contacted first, while contacting core developers and other key contributors should be a relatively late step in the process. In many cases it’s a bigger problem that developers avoid utilizing available resources and rather spend too much time trying to solve a problem themselves rather than asking someone who are likely to be able to help at an early stage in the process.

5.1.3 Feature gifts and contribution barriers

Feature gifts are a common occurrence in open source, and seem to be present in vertical domain information systems too. In one of the cases I observed, code related to serialization of objects into XML was donated by a developer shortly after joining a project. The reasoning was to save work, and reuse the developers existing work to get a head start. In addition, this code was well-known to the developer, making the transition into the project easier due to the familiarity with some of the code. According to Von Krogh et al. feature gifts relate to their specialization in open source projects, and individuals want to form relationships to maximize rewards and minimize costs [47, p1233], which align with the findings.

A developer on the DHIS 2 project suggested to manage the documentation for the project, as it at the time was out of date and lacked proper organizing. Another developer decided to respond with additional help and the documentation project was formed. At the time, the second developer was fairly new in the community and had limited knowledge to the system. As the contributed work to the documenta-
tion project was not provided shortly after joining, or within the domain knowledge of the document, it can hardly be argued as a feature gift. However, there are similarities with the findings of Krogh et al. on the evolving idea that open source software innovation hinges on contributors giving gifts in the form of code [47, p1233]. In this case, the contribution to the lacking documentation would be valued just as high as a code-based gift contribution, if not higher.

The documentation project also had another effect on the project, as it helped lower the contribution barrier to the project. Von Krogh et al. suggested that feature gifts by newcomers are related to contribution barriers of those newcomers that come after [47, 1234]. Out of the four items they suggested pertained the contribution barrier [47, p1231], “easy of modification” is the most relevant related to the documentation project, as it helped reduce the complexity of the source code by documenting the inner workings and relations between the different parts of the project, easing the access to information for later newcomers. In addition, the documentation project would also lower a second barrier, the “Flexible module framework” barrier, as module interfaces and approaches were documented more thoroughly than in the API documentation.

Ducheneaut [8, p257] suggested that contribution barriers were used purposely as a selection process, to avoid project leaders having to constantly “holding the hand” of newcomers. Von Krogh et al. found that potential developers announcing their interest in the project was generally welcomed to the community [47, p1228]. However, he found that many of those who announced their interest never became developers on the project. In the case of vertical domain information systems, the general interest from new developers does not seem to be a problem, thus there is little need to “black-boxing” relationships, and hiding project details. Newcomers seem to be generally greatly appreciated, as the vertical domain suggests, the interest is generally not as broad as in traditional “horizontal” projects.

In addition to the contribution barriers Von Krogh et al. proposed, many open source projects seem to create an additional barrier on purpose. Both projects in this case have barriers in the form of code reviews. DHIS 2, through the commit mailing-list, sending out a copy of the changes to all registered developers. The review is then performed on a peer-level basis by developers with time and interest in checking the changes, meaning there is no formal structure for this process in the project, and the feedback is given after the change has happened. However, the version control system utilized, mean that changes can be reverted at a later stage. In the case of OpenMRS, the review process has become formalized. As in the DHIS 2 project there are code style rules etc. to be followed, however major incoming changes are reviewed fol-
allowing a process defined on the website, with checklists for code style and common mistakes. This process could be seen as a barrier to contributors, as they will have to follow all standards to pass through the system. It may also be viewed as a way to align the community, and make sure everyone is on the same course, adjusting the direction of those in need of it.

5.1.4 Balancing growth

Even though recruiting developers is viewed as important, one of the cases revealed that the project was worried about the size of the core developer group [28, p529-531]. It was stated that there was tension between bringing more developers on board on the project, and getting overwhelmed by the number of voices. In this case, it seems to have changed as time has moved on. Seemingly, the situation may change as structure, code-base etc. stabilize allowing easier coordination between groups of developers. In 2007 and 2008, OpenMRS participated in Google Summer of Code as a mentoring organization. As GSoC is designed to bring new developers into the open source world, recruitment of developers obviously has changed somewhat since the article was published. However the view on keeping the number of core developers low may not have changed. Currently, neither of the vertical domain information systems studied seems to have problem with too many voices, though both are past the initial critical phase at this stage, and with module frameworks development can easily be modularized and separated into smaller units, reducing the number of involved people on the various sub-projects.

5.2 Training and mentoring

The DHIS project bases much of its short term recruitment on student from local universities attending courses on open source software development. The students are introduced to common tools and methods for working on open source projects like version control systems, mailing lists for development communication, Wiki based documentation systems etc.. Introductions to these tools are taught by the main teachers and lab assistants during the first part of the course to give everyone attending an equal standing before project work in groups are started. During the course the students are given feedback on their usage of the tools, though that is in most cases given the same way as any other open source project would give feedback; through the systems themselves.

The main difference identified here is that there’s a period in the beginning where the student developers are coached by the lab assistants.
on both tools and programming frameworks. The students are grouped and introduced to a project they will be working on during the course, and during that period they’re being coached by the lab assistants. This close and personal followup is natural in a course-setting where the developers are located mostly at the same place and meeting regularly in person.

For students remaining on the project after the course, there’s no organized mentoring system, though most students by then know their way around the project, and also have contact with their ex-lab assistants and other developers through the regular channels like the developer mailing lists and in some cases through chat-tools and personal meetings.

DHIS also has developer nodes located elsewhere in the world, and new developers joining the project at these distributed nodes they too need training to get up to speed with the project. This has been handled in different ways, some of the developers have received on-site training by other master students working in the field, this has happened in both India and Vietnam and has been moderately successful. Developers from both Ethiopia, India and Vietnam has also visited other development nodes for shorter or longer time to receive training and work together to increase the knowledge and understanding of the different frameworks and tools used in the development.

Experience from OpenMRS’ participation in the Google Summer of Code is a bit different; all projects available had mentors assigned to them, or a mentor was assigned when the intern project was accepted. The mentors was following up their intern monitoring work and providing domain- and technical knowledge as needed. As most interns accepted in 2007 was spread around the world meetings in person was not common. Unlike in the DHIS setting the interns accepted into the Google Summer of Code program did have some prior knowledge of tools and frameworks. Due to the huge amount of applications that the OpenMRS project administrators could choose among, they could thus select interns with a high likeliness of completing the project successfully. The mentoring work, in my experience, was adjusted as needed, but was mostly on a domain level rather than on a technical or code level.

Both these projects break with Ducheneaut’s findings from the Python project as he states that «There is also little evidence of explicit coaching or teaching from established experts. Instead the participants have to discover by themselves what the norms of participation are» and «It is interesting to remark that I found no evidence of coaching of any kind from the project’s members: the acts of finding bugs, reporting them, and proposing a solution to them all stem from the participant’s initiative» [8, p352]. However, coaching at the level Ducheneaut mentions is
not common in either of the two projects, as mentoring the act of finding bugs and proposing a solution has not been observed, however the act of how to properly report an identified bug is.

The differences identified between the DHIS and OpenMRS mentoring is likely a response to the difference in location of the students/interns. In the case of DHIS they’re usually not located alone, but can relay on some support from their peers. The interns of OpenMRS have to relay on mentoring resources located elsewhere. The university setting of DHIS breaks with Ducheneauts findings in Python; «Python-dev, therefore, is not a place for novices to learn about computer science - this knowledge is assumed. What the newcomer has to learn is how to participate and how to build an identity that will help get his ideas accepted and integrated» [8, p352] as some of the purpose of the DHIS project is to educate developers, and especially the developers in the field so that they later on will be able to continue work on the project and create modules, modifications and bug fixes without relying on the main node in Oslo to do so.

The common need among the projects is developers that are able to identify what work is needed themselves and work individually or in groups as needed. Von Krogh illustrates this with an e-mail request: «I am a Java engineer [...] I have been working for five years, and I really would like to help. Give me something to do.» and comments that «This person tends not to do anything» [47]. As for the developers’ motivation Hars and Ou reported that close to 71% answered «improving my programming skills», close to 52% «because I can build a network of peers», but also that some answered «learn English and teamwork» [19, p6]. The last group mentioned is also present in the DHIS project; several of the project members that over time quit the project has reported that they now don’t get to practice their English skills as much as before and that leaving the project has affected those skills. Ducheneauts points out in relation to Hars and Ou’s findings that «From my analysis it looks as if participants come already equipped with good programming skills, and learn instead how to contribute meaningfully to a fairly large-scale project such as Python.» [8, p352] while most developers on the DHIS project are fairly inexperienced developers at the time they join the project.

5.2.1 Code review

The levels in the trajectory covering defending the new feature and the implementation of it has parallels in both DHIS and OpenMRS. In the case of DHIS major features are developed in separate branches and integrated with the rest of the system when the developer feel that the code is more or less ready for use, containing at least the most basic
functions necessary to operate properly. During development most of the frequent contributors monitor mails with changes sent by the version control system automatically on every change, containing all the changes. This often spawn feedback on the code, especially in the cases where the code does not follow the projects specific coding style, the change is not followed by a proper log message explaining the change, or in cases where code is deemed to be highly inefficient, unclear, not well documented or even incorrect. This is an invaluable feedback system that provides developers with tips on how to improve their code. It is probably the closest one comes to mentoring on the DHIS project after the initial course-phase. As all developers with access to the code repository are in practice able to make changes to any part of the project this is also the closest one comes to peer reviewing code developed by regular DHIS developers on DHIS. It's often a case of commit first and ask later. Changes have been reverted on several occasions due to the code breaking code standards, leaving the project in an unusable state, or the changes just not being acceptable for various reasons. The finished project usually does not receive such treatment before being finally integrated with the system. The focus is on giving feedback as quickly as possible.

Student projects are treated differently, and in most cases never make it into the main project. Part of the reason for this is due to the nature of the projects. They are often researching possible approaches to add certain functionality and either end up being thrown away or re-implemented by regular developers on the project. An example of this was a module implementing “target” functionality, the ability to set a target value for something measurable in the system, by a group of student developers. The projects was fairly successful, well written and documented, and got included after one of the core developers inspected the module. After some modifications and bug fixes over the next few weeks it was properly included in the main project.

Other projects are known to have more thorough review systems and Ducheneaut found that the Python project was doing this in the end-phase of the module development rather than continuously throughout the development: «Moreover, the output of the work on this module will be evaluated during a rite of passage, where the entire community scrutinizes what has been produced and the core members finally deliver a verdict of acceptance or rejection» [8, p351]. The OpenMRS project has structured itself differently. Even though developers with access to the code repository are able to make changes to every part of the project, the rules state this is not allowed. Minor changes and bug-fixes that are extremely unlikely to cause any discussion may be committed directly to the main project. All other changes are going into a development branch, preferably through a branch or as a code modification uploaded
in a special file format to the bug tracking system for review by one of the core developers.

Unlike many other open source projects, credit to the developers are only given in the log messages and never stated in the code itself, as the code is officially donated to the organization for legal reasons. OpenMRS was also trying to enforce a new rule of in-depth peer reviewing new major changes from branches. This work has led to a detailed description of the review procedure, including checklists over what to look out for, templates for review feedback and a proposed plan for architectural reviews [33].

A review session may be conducted like a conference call where all participants are looking at the code, explaining the functionality from top to bottom while going through what happens where and how while receiving feedback on both structure, code and documentation. This ensures that the core developers are properly aware of how the solutions was meant to work and spreads the knowledge among more developers than those working on it initially. In the case of the Google Summer of Code projects where some of the mentors were external from the core developer group this may prove useful to keep knowledge about the code in-house.

5.3 Controversies and communication

A controversy in the development of DHIS was caused by a rather minor change of how to visualize the toggling of the left menu in DHIS. Based on real world experiences one of the developers came with some feedback to improve the visual look of the toggling feature. At the time the menu toggling was visualized with plain text symbols: «The “<” symbol for closing the left window can look as a “failed” parsing of some text in the HTML code. How about using “«” and “»” or some sort of icon for this job?» [45]. This message caused a minor discussion about whether to choose an icon rather than symbols and benefits/drawbacks for either solution. A few months later no result of the discussion had appeared in the code repository and a new message was sent to the mailing list: «Hi; I observed a few months back a mail thread (http://www.hisp.info/archives/dev/msg01879.html) regarding the toggle-function and the use of <, « or icons. It seems there was a kind of conclusion to it, but I can't actually see that anything changed. I've now changed it to using two icons (http://tango.freedesktop.org/static/cvs/tango-icon-theme/16x16/actions/go-first.png and a mirrored one). If nobody complains before I'm back from dinner I'll just commit it. I think it looks a lot better this way. If the 6 extra pixels is too much we can find some other icons.. or later change to the pro-
posed solution» [13]. Obviously from the mail this was looked upon as a minor change from the developers side, but still this mail sparked a very long discussions on the development mailing list with fairly strong opinions about the many different solutions available and variants of possible icons.

Ducheneaut refers to Latour stating that «To reach the point where the controversy can be resolved, a network of allies has to be assembled first» [8, p344]. In this case a network of allies was not assembled at the time of the mail and the interest in this change was also severely under-rated by the developer. There were no statements supporting the controversial use of the icon originally designed for a different purpose. «Indeed statements regarding a controversy are weak if they are left alone. To make a statement stronger, it needs to be connected to what others have said beforehand. This way anybody opposed to the solution offered has to attack not only the solution and its provider, but also a string of other propositions and assertions made by others beforehand» [8, p344].

Rather than trying to force acceptance of the contribution and setting a time limit, a better approach would have been to go back in time in the code base and identify stakeholders and contributors to that specific section of code and first discuss possible changes with those developers. Talking with the actual users and implementers in the field to identify their views upon this would also have been wise, as these persons teaches the users and will have views and experiences from user training on what works and does not work. This information did appear on the mailing list at the end of the discussion and was the actual turning point of the discussion, before a decision was forced through by one of the project administrators so that developers could resume work on what the administrators viewed as more important features [43].

Similar examples of controversial code contributions has happened in other cases too. The performance of a critical module in DHIS; the Datamart module; had been discussed over time. The module is supposed to run aggregation rules on the data to prepare for analysis of the data. With the amounts of data gathered at certain sites in real life this operation regularly could run for hours and hours, and even through nights. This was not acceptable for the implementers that needed the data available for analysis rapidly to provide the necessary information and reports to their bosses. DHIS is built and organized to be independent of the underlying storage system through Data Access Objects. Database independence is achieved through the use of the Hibernate framework. One of the developers and implementers located in the field suggested that he could make a better performing module with pure SQL avoiding the cost and complexity introduced by the frameworks. This of course violates the data store independence approach the project had tried to maintain, and would make it much harder to change data base
systems later. As at least two different database systems was in use (MySQL and PostgreSQL) and the introduction of a hard coded MySQL based datamart would mean that another version for PostgreSQL would have to be maintained too. The work on the module was started regardless of this issue, and the module was added to the source code repository and changes made to replace the existing module with the new one without getting real acceptance for this on beforehand. When other developers also realized that the new module also had other problems, did not always compute the right results and was a bit buggy there was some discussions about the module.

This event is a parallel to what Ducheneaut observed in his studies of the Python project: «Qualitatively, this proposal is quite different from simply fixing bugs. In proposing to add his module, Fred is not addressing a pre-existing controversy: he is creating one. To add his module, the very fabric of the project’s hybrid network has to be challenged: relationships between people, between artifacts, and between people and artifacts would have to be changed to accommodate the addition of a new piece of software. And as was the case with bug fixes, Fred needs to build up a network of allies to support this effort - a fact that Guido makes plainly visible by saying that Fred needs to explain why his module is better» [8, p346]. The developer did not take the time to build up a large enough network of allies before introducing the new module to the DHIS project. Some support was gathered by the implementers facing the problems in the field, but as this change would cause extra work for the developers over time and was seen as a sub-optimal solution he did not gather the necessary support and broad consensus from the developer community but rather ended up facing even more reluctance from developers that had to fix errors due to the introduction of the new module.

The new module also did not follow the agreed upon code standards for the DHIS project. It was thus viewed as something that should not have been committed in its current state, but should rather have gone through formatting, code cleanups etc.. However; this event did seem to push the development of the original module forward, as the competition increased. The increased focus and work on the original module after this event ensured that over a couple of months the speed was improved to the point that the module could compete with the speeds of the MySQL specific module introduced.

5.3.1 Resisting change

This practice ensures that new changes are properly accepted by the most important and central of the projects’ members. Open source software networks are designed to resist change, unless it’s a change that the
network can accept; «Open Source Software development is politics by other means» [8, p353]. From the developer under reviews' point of view «(…) the “foes” here are the entire network, designed to resist change, which must be weakened in strategic areas and eventually reconfigured if a participant's contribution is to be accepted.» [8, p353] meaning that his solution must be a good one, well documented and that he must be able to convince the network, and especially members central for his feature, of the need of such a change to gain support for the solution. Being able to convince the relevant members means that you first need to identify them.

Networks of developers that are communicating outside the normal communication structure may make this harder to reveal. As an example many of the developers on the DHIS project are located at the same nodes and interact both in person and through other private communication channels. These discussions may never turn up in the official communication channels, and if they do, often just the conclusion does and not the underlying reasons, this applies specially for strategic discussions. Tommerholt found in his study of communication in the HISP network that it had similarities with FLOSS projects' communication structure, where the communication is electronic and open. However, he also noted that it operates outside the traditional FLOSS tradition with some communication being on a one-to-few or few-to-few level. Partially this can be explained due to the co-location of some developers, where it is easier to make decisions internally in a smaller subgroup of developers. Unfortunately, in addition to being discussed in private fora, these decisions may never be revealed to external developers if they're viewed as minor, or not of general interest to others. A second reason behind private communication may be what Scacchi refer to as communication that is publicly disclosed due to their perceived sensitive content [38, p14], for example strategical decisions by coordinators. Physical infrastructure may be an issue to projects operating in remote locations. The teams stationed in Vietnam frequently, up to several times a day, experienced loss of Internet connection, as well as frequent electrical dropouts for hours. Working with slow Internet connections on remote servers located on the opposite side of the earth can be painfully slow, and make communication near impossible at times. Tommerholt found these physical infrastructural problems to render use of communication tools ineffective [46, p100]. As these discussions are held in private, they're obviously hard to reveal unless you're part of them, as such I have little insight into the inner workings of other vertical domain information system projects and their communication strategies. However, the impression from my experience with the other vertical domain project is that this strategy partially exists there too. The work is divided in subgroups of developers which make minor decisions in-
ternally. Larger proposals seem to be documented on wikis as well as on public mailing-lists to a larger degree, as the awareness of the issue seems higher among the participants, however it is hard to say whether my observations are representative for other organisations. In general, all information flowing in the community is sought to be made available through standardized methods like Really Simple Syndication (RSS), aggregating the information from blogs, mailing-lists, wikis etc. into one information feed.

In the case of the code in DHIS the developers are encouraged to tag the code with their name so that it’s easy to track who’s been working on a certain file through annotations. As the code normally is committed by the developer in person, the change logs will also reveal who has had an active interest in a certain piece of code, making it fairly easy to identify stake-holders. The “protection” of the project found by Ducheneaut thus seems only partly valid for the DHIS project: “A “trick” used to fortify a hybrid network is to “Black-box” the relationships between actants: the process through which connections were established in the first place is hidden from view, so that anyone who would like to challenge the current state of the project will have to uncover these relationships first» [8, p353].

The project also has listed the project leads along with the core developers on its webpage. OpenMRS on the other hand tries to actively reveal its structure on their website presenting the different working groups, their tasks and at least their key members. «This page coordinates efforts between the many interested groups in the OpenMRS universe. The idea is to identify informal “Working Groups” for each area of development (programming, not organizational or geographic), providing the opportunity for everyone to get some idea what everyone else is doing. Participation is of course not mandatory, but simply encouraged. This is loose, community driven project management» [24]. Another frequently used approach is to probe the network to reveal its structure: «By asking simple questions about the current state of the project, a participant can see from the responses he obtains who is connected to a particular artifact, and what the nature of this connection is» [8, p354].

5.3.2 Developer types

Von Krogh et al. identified three kinds of people among the joiners in his study. I identified similar kinds of people during my work with the vertical domain information systems. The “contributor” category, primarily containing regular committers and core developers seem to match the findings of Von Krogh et al., specifically the description of the joiners who, when noticing a problem, provided a solution back to the community. The other two categories I found, the “non-self-going” and “vision-
ary” also fit with the kinds that “tend not to do anything” and those who tend to fight with core architects, and not follow their suggestions. This indicates that vertical domain information systems see the same kinds of joiners, and thus developers, as other FLOSS projects.

5.4 Developer motivations

The classic view of FOSS developers are that they are themselves users of the software that they develop, be it either as some type of desktop software or library they need as part of e.g. their work [38, p5]. Scacchi refers to previous research covering the complex motivations that FOSS developers have; why are they willing to allocate time, skill and effort to work on FOSS projects [38, p8]?

- Fun, personally rewarding, provides a venue where they can improve technical competence that may not be possible within their current job or line of work
- Building trust and reputation
- Achieving “Geek fame”
- Being creative
- Giving and being generous with one’s time, expertise and source code
- Maintain and improve software development skills

As both the DHIS and OpenMRS software is within the health domain, few of the developers are actually medical personnel that use the software as end users. Some of the people in the core of OpenMRS project are actually medical doctors and do see patients regularly. Thus they have very important knowledge from both sides of the table; as developers, and the medical knowledge and experience from real life experience of the use of such software. This is key knowledge to successfully develop such specialized software. As the software is aimed primarily at third world countries and is not used in the developers’ home countries, there still is a gap between the traditional style of FOSS development where the developers are the end users.

As the open source software development course is the main recruitment source for the DHIS 2 project, the motivation of the students is interesting. The motivations cited in the literature - improving and maintaining their software skills, being creative and partially providing a venue for improving technical competence not available in their current “job” (courses), all match with experience gathered from working as a group assistant on the course.
5.4.1 Common training

In the case of DHIS the core developers are even further away from the field as most of them are students with little or no knowledge about the medical field. There has been attempts to increase the knowledge among the developers in several ways. One of the initiatives was to arrange a common 2 week intensive course for both medical personnel, both local and foreign, and software developers at the University of Oslo. The course was covering both theory and practical group work. Setting up the groups with a mix of developers and medical personnel and giving them real-life-like tasks of analyzing situations, making relevant reports and presenting as they would be on county-level. As a participating developer on the course it felt highly useful and increased the understanding of the usage of the software a lot. A key to developing user friendly and fairly complex medical systems correctly in the first place is to understand the usage of the system and the needs of the users.

5.4.2 Real life experience of the system

Another key element in the process of increasing the knowledge of the developers is the goal of the HISP project that all developers should go abroad and work on the development and/or implementation of the system where it is in real use, or being prepared to be used. As such, all developers get some level of real life experience with the system in the field. This practice has ensured that students end up working in a range of countries (on several different versions of the system) from India, Vietnam, Ethiopia, and Cuba, coming back with different experiences and having faced a wide range of different problems, from cultural to technical.

5.4.3 Cross-domain usage

The DHIS software was initially designed to be very flexible and independent of the health domain. The idea was that the application of the system would be wider, and possibly be useful for a wide range of domains; as hierarchical reporting and data analysis is a common need in most fields. The software is so far in reality not used for other domains as far as we’re aware. The thought however was that it could possibly attract developers and resources from other domains to help develop the general functionality of the system faster, This could also possibly attract core developers with a personal need for the system, thus making them more likely to take part in the projects over time. One of the initial core developers did have this personal motivation when he started designing and working on the project, but so far has not applied it to other
5.4.4 Implementers

Most of the well-known FOSS projects have two levels of stakeholders; users and developers. These two are in many FOSS cases the same [38, p5]. OpenMRS and DHIS are both infrastructural software projects where, as explained earlier, the users and developers in most cases are not the same. These two projects do however have another level in between the users and developers commonly referred to as implementers. The key roles of the implementers are to facilitate the installation of the software, setup of the medical terms and reports, training end users at different organization levels, and communicating needs to the developers. The role as a middleman/facilitator between end users and developers communicating needs between these is quite important. End users seldom turn up on the developer mailing lists with issues and needs, however the implementers are frequently observed as contributors to these lists. As earlier mentioned, most developers on the DHIS team have no medical experience, while the users have medical experience but little or no computer/developer experience. The implementers often holds a little of both, and especially the understanding of what the users need in the field. They thus often play an important in translating the needs of the users into requirements that the developers understand and implements.

5.4.5 Knowledge transfer

A goal of the DHIS 2 project is that the project should be developed as much as possible by developers from the countries using the system. The advantage of this is that they’ll be able to do modification, maintenance and addition of modules/features themselves and as such being less dependent on the initial core developers and research project initiated at the University of Oslo. This transfer of knowledge is a security for these countries too, knowing that everything is open and that they do have the knowledge necessary to use the system independently of the initial developers. The exchange of developers where both Norwegian students go abroad, and developers from abroad travel to some of the other nodes for shorter or longer stays are important in this matter. The developers work together and exchange experience and knowledge. Knowing and having met the other developers on the project seems to increase the feeling of being a group working together against a common goal, and decrease the fear of using mailing lists and other common resources that might initially look intimidating and scary. As the experience with technologies and frameworks is getting better and the
knowledge of these increasing, there has been a noticeable transfer of
where core contributors are coming from. As many of the initial core
members located in Norway has left the project over time, new contrib-
utors have stepped up and turned into key contributors to the project,
suggesting that this knowledge transfer is starting to happen.

In a project like Python knowledge transfer is important as with all
systems, however, special care may be necessary in the case of verti-
cal domain information systems. Both the HISP network and OpenMRS
has implementations around the world. These require local adjustments
and features to fit the need of the users and their information needs.
Compared to Python with a wide range of uses, the knowledge transfer
in these vertical domain information systems becomes more important
as the projects have responsibilities for the implementations, as well as
the local adjustments. The specialization of these systems means there
are fewer developers to share this knowledge, and less space for making
mistakes in this process. An example of this could be the weak knowl-
dge transfer of previous work related to DHIS 2 in Vietnam. A group
of students I was part of was supposed to continue the work in Viet-
am. We had no/little overlap with previous developers, and only talked
briefly about the situation and previous achievements in the node. How-
ever, one of the HISP coordinators followed us for the first two weeks
of the stay, to guide us during the initial period. Most of the knowledge
transfer focus was on practical issues as transportation, offices and cul-
ture, which in the beginning of such a stay is very relevant. However, as
the dust settled and we were left to continue the work, it became appar-
tent time and time again that the knowledge transfer had been too weak
on the detailed history of the implementation and tasks performed in
Vietnam. We were not aware of the details of an earlier cooperation with
a local consulting company. Similarly little was known about the status
of the DHIS 2 test implementations in Vietnam, especially in the south.
Our involvement on this test was limited due to not knowing the details
about the implementation and goals until late in the process. Similarly,
the involvement with a cooperating local university also grounded to a
halt, as we were not aware of the level of cooperation and their expecta-
tions until late in our scheduled stay.

As the literature on the project increases, the broad history of the
project is becoming increasingly accessible to new students, as there
is less need to contact previous developers on the project, potentially
easing the knowledge transfer.

Knowledge transfer in distributed nodes pose challenges. In the case
of DHIS this knowledge transfer seems to be limited by the physical
infrastructure to some degree. As access to electricity and a stable Inter-
net connection can be problematic, the information flow to other phys-
ically separate nodes is somewhat limited on occasions. Slow response
times from servers located elsewhere in the network makes the usage of knowledge transfer tools like mailing-lists and wikis less attractive. Adding to the challenge, language barriers are a major factor. Understanding and learning technical information explained in a secondary language such as English was challenging in the case of DHIS 2. Vietnamese are used to study literature in Vietnamese, and reading technical web sites and APIs proved challenging on occasion. Language skills vary from developer to developer, though generally improved over time. However, as new and inexperienced developers unfamiliar with working in an English speaking project join, this challenge seems to remain fairly constant. Flexibility in the working situation when it comes to language, live translations by peers, and close communication worked well in the case of DHIS 2.

5.4.6 Contributing to other OSS projects

A number of developers on the projects have been observed working in parallel on other OSS projects. Some of the developers in the community around OpenMRS is mainly working on their own OSS project based on the database architecture created by OpenMRS, but with a different user interface and target. They have also been active in other OSS projects related to graphics functionality they need in their main project. Similarly, some common code for XML processing originating from the DHIS project has been separated out and is intended to be offered as a FOSS library by two of the DHIS developers. One of the first core developers also created a plugin to the Maven software to handle automatic installation of the software on the application server Jetty which is available through the Maven 2 plugins Mojo-project.

The fact that developers from both projects both create and participate in other OSS projects is fairly common. Scacchi [38, p8] refers to a study where it was found that at least 60% of the FOSS developers participated in two or more FOSS projects. They also found that 5% of the developers participated in 10 or more FOSS projects. This indicates «(…) that there is a growing social network of alliances across multiple FOSS development projects. Project contributors who span multiple FOSS project communities serve as “social gateways” that increase the ongoing project’s social mass, as well as affording opportunities for inter-project software composition and interoperation» [38, p25]. In the case of DHIS these numbers are probably too high; most of the student developers on the project have relatively little programming experience and usually the DHIS project is their first participation in an OSS project. Between the DHIS and OpenMRS project the number of developers with knowledge about both systems are increasing, but at this point in time the outcome is still not significant. As it is still fairly early in the process
this is likely to change and the ties between the organizations will probably become closer over time, as the social relations and network most likely grows stronger.

5.4.7 Central nodes

«Becoming a central actor (or node) in a social network of software developers that interconnects multiple FOSS projects is also a way to accumulate social capital and recognition from peers» [38, p8]. Especially in the last case of the Maven-Jetty plugin this is relevant, as it both interconnects two OSS projects itself, but is also used by many other projects. OpenMRS in itself seems to turn slowly into a central node in a network of health software developers as more and more health related OSS projects start integrating with the project. Scacchi points out that «The density and interconnectedness of this social networking characterizes the membership and in-breeding of the OSS movement, but at the same time the multiplicity of projects reflects its segmentation» [38, p50].

There might be several reasons behind their involvement in several OSS projects. One main reason observed from participation in the two projects and following the community is that the projects the developers contribute to are projects or libraries/functionality that is needed in one of their other projects. The second reason which is not as obvious may be as Scacchi observed; to accumulate social capital and recognition from peers. He also mentions research pointing out that financial rewards, in terms of higher salaries for conventional software development jobs [38, p9, p14] may be important to some, or at least a positive bi-effect. How relevant this is in the DHIS project is unclear, as most come fresh out of school. Though, a couple of the early contributors and core developers on the project have built themselves a solid reputation and have central roles in the Java developer community in Norway. In general showing that you contribute to OSS projects is viewed as positive among employers as it shows motivation, skills and that you’re a self-going type of person. When most students have little experience to point at, this is invaluable, and goes for all contributing developers, not only the core developers.

5.4.8 Multi-project clustering

It’s a fact that the vast majority of OSS projects hosted at e.g. SourceForge.net appear to be inactive, many of them without any published software, or a long period of inactivity. Most projects have only one or two developers contributing to the project [38, p49] and never gain momentum to grow into a sustainable community. Many of these project
have a limited scope which may not be of interest to many other developers, or they may be a yet another implementation of some software, where there is already present other solutions that people prefer to utilize. Some of these projects could potentially be good candidates for merging into larger multi-project clusters, interconnecting them into larger units. Examples of this would for example be code-related projects clustering with communities surrounding media-player software, small desktop applications clustering with desktop environment projects etc. «Multi-project clustering and interconnection enables small FOSS projects to come together as a larger social network with the critical mass needed for their independent systems to be merged and experience more growth in size, functionality, and user base. It also enables shared architectural dependencies to arise (perhaps unintentionally) in the software components or sub-systems that are used/reused across projects» [38, p37].

There are signs that this clustering is starting to happen between OpenMRS, DHIS and other projects like Baobab, EpiHandy and others, with OpenMRS as the central node. The software developers on the different projects work with many of the same problems and within a similar domain, health. As this domain is a relatively minor domain in the world of FOSS software, a larger community should be positive and encouraging for the developers working within it. The ability to discuss with and meet other developers working within the same field is important for any community.

As mentioned earlier, Scacchi points out that the clustering also enables shared architectural dependencies to arise, perhaps unintentionally. DHIS and OpenMRS is based on many of the same frameworks and developed with the same programming language. These choices was taken independently years ago by both organizations, but as both organizations have similar goals of database independence through the use of Hibernate, architectural choices etc. both ended with many of the same design and technical solutions. As both projects are fairly mature at this point in time, changing core frameworks is likely to require huge developer efforts. Minor changes and adjustments have however occurred. While evaluating potential frameworks for e.g. JavaScript/AJAX-solutions ideas and experiences have been exchanged at least informally. Since there is a goal over time to integrate the projects closer, there’s a focus from the DHIS side on not drifting apart technologically.

Both OpenMRS and HISP, with their central developer nodes operate with organizational support structures, in the shape of universities and non-profit organizations. Both organizations operate within the same vertical domain, with users and developers separate and implementers in between, and external backing from third party stake-holder organizations like WHO, Partners in Health and Regenstrief Institute. These orga-
nizations’ funding and support, the partially course-based training, and partially professionalized developer organizations suggests that FVDIS organizations are more professionalized, with support structures, than traditional open source projects.

5.5 Sustainability

The core developer resources in the OpenMRS project are contributed mainly from two external organizations; the Regenstrief Institute, and Partners in Health. The two organizations are both committed to the health-care domain and have provided developer resources to the OpenMRS project since 2004. Unlike traditional FOSS projects where developers spend their spare time working on projects they have personal interest in, most of the core developers are full-time employees of who work on the project for a living. As the core developers have OpenMRS as their main focus, this means that they are able to contribute to and focus on the project over time. A small turnover of core developers that know the processes involved with their work well, leads to a common understanding of project structure, quality of code, standardization and best practices for code and documentation in the project that emerge over time and stabilize; a community of practice.

Developers on the DHIS project are mostly students enrolled in master or Ph.D. programs. Most students affiliated with the project spend about a year on the project before they leave university. This means that students join and leave the project at a higher frequency compared to the case of OpenMRS. Most students focus their work on a small limited feature due to the short project time-frame and steep learning curve before one is able to contribute to the project in a meaningful way. Most DHIS student projects focus on integrating a new feature in the software, integration with an external application or similar. The focus is on the material outcome; producing something that works in the short time-frame, resulting in a varying level of documentation. Best practices and standardization of code is important, but takes time to get used to, and even more time to evolve in a direction that all can agree upon.

Evolving into a core developer in the OpenMRS project is likely to be time-consuming as there is little turnover of (informal) core developer positions. In the DHIS project the positions of the developers are changing at a higher frequency. There are thus better possibilities for developers to evolve into a core position if the necessary skills and understanding can be proved through contributions.

Comparing the two organizations and their developer communities it seems likely that long term participation encourages and improves structure, documentation and standardization of code and processes. A
well documented project with a solid structure and standardization of processes and code is likely to have a more solid foundation for a long term sustainable community, as standardized processes lowers the entry barriers and amount of new processes to learn when changing tasks on a project. This ensures that more community members are able to pick up the pieces and carry on work on different parts of a system. Sustainability of an organization is also in many situations dependent upon external relations with other organizations. Long term relations between developers, clear structure and good documentation should be positive factors for long term sustainable relations in such situations.

5.6 Open Source communities

The developers of the DHIS software differ from the “scratch an itch” mentality. Most of the active developers at any given time are Master level students at universities with no special medical background or interest. This points out that there might be other advantages and gains involved too. The most obvious are that the involvement is the basis for most of their research related to the Master thesis. Other attractive elements may be the possibility to work on real-life systems with real users and requirements, rather than specialized and limited software that in many cases will never be used outside the setting of their thesis. Working with technologies that is very relevant in the business world and thus increasing their own market value and knowledge might also be a big motivation in such scenarios. Working in an international setting, there is also very good possibilities for the developers to visit some of the other nodes in the network and work closely with different levels of users of the system and gather requirements and feedback directly from the grass-root level.

There are full time paid developers in the HISP/DHIS network, but most of them are working on specific features and implementation relevant to their location. These developers are typically stationed somewhere in the public health system in their respective countries and also do work related to support and implementation. In periods there has been developers working full time on the core of the DHIS software, but most of the work is done as part of a thesis, spare time work or in student project groups at the university.

Developers in the OpenMRS network are quite different from DHIS. OpenMRS is a multi-institution, non-profit organization led by Partners In Health and Regenstrief Institute, Inc.. OpenMRS has a core of approximately 5-7 full-time developers. These developers are mostly professional software developers working for the Boston-based philanthropic organization Partners In Health and Regenstrief Institute - a medical in-
informatics research institute in Indianapolis. Another way in which OpenMRS have tried to increase the awareness of the project and the recruitment of new developers to their project is by participating in Google Summer of Code.

The OpenMRS community have also attracted external developers. One example is one of the Google Summer of Code mentors, which works for a medical equipment company. His day-to-day job is within the health information system domain, and got involved through his knowledge of Partners in Health. He stated that he had previously worked on synchronization issues for other companies in health information systems sector, and was looking for a relevant, technically sound open source project with a decent install base to contribute to.
Chapter 6

Conclusion

The research objective of this thesis was:

**Research objective:** Explore the difference between traditional OSS projects and how OSS development and capacity building is affected in free vertical domain information system (FVDIS) developer organizations.

This research objective was summarized in two research questions:

**Research question 1:** What are the differences and similarities in socialization in FVDIS type projects, as compared to traditional open source projects.

- Specifically, I examined recruitment, developer training and communication patterns and the interaction between these.

**Research question 2:** How do the differences affect the sustainability of such projects?

Through the work with this thesis I have found that recruitment in FVDIS organizations, represented by the HISp network and OpenMRS project, differs significantly from recruitment in traditional open source software projects. FVDIS projects do not have a large user base with potential developers to recruit from, due to the fact that end users of FVDIS projects are normally domain specialists (doctors and nurses in the cases of HISp and OpenMRS). Software developers are not automatically in the target user group, as would be the case with more widely used software such as a web browser. In the DHIS 2 project, developers are recruited from participants on an open source software development course at the University of Oslo, as well as from partner universities. OpenMRS currently bases its external recruitment on Google Summer of Code students.
Developer training is largely absent in traditional open source projects. In OpenMRS each summer intern was assigned a mentor who had the task of providing assistance and domain knowledge as needed. Developers on the DHIS 2 project usually receive their initial training through a university course, where group assistants are responsible for supporting the students. Assuming that FVDIS organizations in general have support structures like HISP and OpenMRS, training in FVDIS projects is likely superior to training given in traditional open source projects.

Large traditional open source projects usually operate in a distributed fashion, as developers are recruited from the project’s user base, which is usually geographically spread out. HISP and OpenMRS operate largely in a semi-centralized fashion. For instance, HISP developers are gathered in geographically distributed nodes. Information and discussion within one node was often not made visible to other nodes. Large open source projects with an evenly distributed community are likely forced to conduct more of their communications via channels visible to all developers, such as mailing-lists. The amount of time spent lurking seems to be reduced compared to traditional open source projects, as recruitment of new developers is not necessarily driven by interest, and mentoring resources are made available to the newcomers. Due to the need for active recruitment to vertical domain projects, contribution barriers are generally sought to be minimized.

Communication with end users differs as well, as most of the end user communication flows through an extra layer of implementers. These are necessary to bridge the gap between developers and users. In general, users of traditional open source software would be regarded as computer literate and able to install and maintain the software on their own. The software users in Vietnam required assistance with these tasks, and implementers were also necessary to perform end user training.

The issues of recruitment, developer training and communication heavily impacts the sustainability of the DHIS 2 project. Unlike traditional open source projects, HISP and OpenMRS rely on organized activities such as the university course, collaboration with other universities and Google Summer of Code to sustain recruitment and long term development. Developer training is crucial to maintaining recruitment, particularly in the case of HISP since the project depends on recruiting developers from a university environment. Students are not likely to take an interest in the HISP project if no training is given. As vertical domain projects tend to rely on a narrower developer base, often with time limited engagements, they are likely to be more vulnerable to problems caused by key developers leaving the project, and lack of proper knowledge transfer procedures.
Sustainability can also be negatively influenced by poor communication. As we began our HISP fieldwork in Vietnam, it became apparent that we did not have enough information on the activities of the developers who had worked there previously. This reduced both effectiveness and motivation among new developers, and thus affected the sustainability of the project.

Finally, I believe that these differences between FVDIS projects, as represented by HISP and OpenMRS, and traditional open source projects have to be taken into account when planning and executing FVDIS projects. Failure to do so may negatively impact the sustainability of such projects.
Bibliography


Chapter 7

Appendix

7.1 Open Source license distribution

FreshMeat publishes aggregated statistical data about license usage for the projects it lists on their website. SourceForge.net does not publish such details, but the OSSMole Project regularly analyzes the projects hosted and publishes these data. By writing a small program to parse the published data we’re now able to roughly compare the license distribution on these two sites. Note that some projects are multi-licensed and each of the licenses would count in this comparison, in the data gathered from SourceForge.net all licenses listed by a certain project was counted. Many of the projects are represented on both web sites.

7.1.1 SourceForge.net

112805 projects with license specified. The 12 licenses in table 7.1 on the facing page represent 95.33% of the “license usage” on SourceForge.net.

7.1.2 FreshMeat

A total of 46911 branches with license specified, 46978 branches in total by 43782 projects. The 14 licenses in table 7.2 on the next page represent 92.15% of the “license usage” on FreshMeat.

7.1.3 Google Code

Google Code is one of the newer open source hosting solutions launched in March 2005. Google Code contains both documentation and APIs for many of Googles own projects, and also their open source projects like Google Gears and Google Web Toolkit. An interesting observation is that Google Code only accepts a very limited number of open source licenses:
<table>
<thead>
<tr>
<th>License</th>
<th>Number of projects</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>GNU General Public License (GPL)</td>
<td>69884</td>
<td>61.95%</td>
</tr>
<tr>
<td>GNU Library or Lesser General Public License (LGPL)</td>
<td>12330</td>
<td>10.93%</td>
</tr>
<tr>
<td>BSD License</td>
<td>7808</td>
<td>6.92%</td>
</tr>
<tr>
<td>Public Domain</td>
<td>3371</td>
<td>2.99%</td>
</tr>
<tr>
<td>Apache License v2.0</td>
<td>2397</td>
<td>2.12%</td>
</tr>
<tr>
<td>MIT License</td>
<td>2358</td>
<td>2.09%</td>
</tr>
<tr>
<td>Other/Proprietary License</td>
<td>1518</td>
<td>1.35%</td>
</tr>
<tr>
<td>Mozilla Public License 1.1 (MPL 1.1)</td>
<td>1442</td>
<td>1.28%</td>
</tr>
<tr>
<td>Artistic License</td>
<td>1425</td>
<td>1.26%</td>
</tr>
<tr>
<td>Academic Free License (AFL)</td>
<td>1341</td>
<td>1.19%</td>
</tr>
<tr>
<td>Common Public License</td>
<td>1251</td>
<td>1.11%</td>
</tr>
<tr>
<td>Apache Software License</td>
<td>1243</td>
<td>1.10%</td>
</tr>
<tr>
<td>Open Software License</td>
<td>1168</td>
<td>1.04%</td>
</tr>
</tbody>
</table>

Table 7.1: SourceForge.net: Raw data collected by the OSSMole project, October 2007

<table>
<thead>
<tr>
<th>License</th>
<th>Number of projects</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>GNU General Public License (GPL)</td>
<td>30104</td>
<td>64.17%</td>
</tr>
<tr>
<td>GNU Lesser General Public License (LGPL)</td>
<td>3095</td>
<td>6.60%</td>
</tr>
<tr>
<td>BSD License (revised)</td>
<td>1395</td>
<td>2.97%</td>
</tr>
<tr>
<td>BSD License (original)</td>
<td>1366</td>
<td>2.91%</td>
</tr>
<tr>
<td>Freeware</td>
<td>1078</td>
<td>2.30%</td>
</tr>
<tr>
<td>Freely Distributed</td>
<td>971</td>
<td>2.07%</td>
</tr>
<tr>
<td>MIT/X Consortium License</td>
<td>809</td>
<td>1.72%</td>
</tr>
<tr>
<td>Other/Proprietary License with Free Trial</td>
<td>759</td>
<td>1.62%</td>
</tr>
<tr>
<td>Free for non-commercial use</td>
<td>732</td>
<td>1.56%</td>
</tr>
<tr>
<td>Artistic License</td>
<td>712</td>
<td>1.52%</td>
</tr>
<tr>
<td>Free To Use But Restricted</td>
<td>656</td>
<td>1.40%</td>
</tr>
<tr>
<td>Public Domain</td>
<td>534</td>
<td>1.14%</td>
</tr>
<tr>
<td>Other/Proprietary License</td>
<td>532</td>
<td>1.13%</td>
</tr>
<tr>
<td>Other/Proprietary License with Source</td>
<td>487</td>
<td>1.04%</td>
</tr>
</tbody>
</table>

Table 7.2: Freshmeat: Data collected on December 12, 2007
• Apache License 2.0
• Artistic License/GPLv2
• GNU General Public License v2
• GNU General Public License v3
• GNU Lesser General Public License
• MIT License
• Mozilla Public License 1.1
• New BSD License

These licenses are some of the most common licenses out there and represented in the top of both FreshMeat and SourceForge.net's license rankings. Limiting the choice of licenses down to these licenses is an interesting way to go, and will likely increase compatibility among projects because of the fewer and more common licenses the developers can choose between. Both OSI and FSF encourage developers to reuse existing licenses as much as possible, and Google Code are driving this one step further by narrowing the choice of the developers even more.

7.2 Open Source licenses

This section contains a short description of the most common open source licenses. The Mozilla Public License (see 2.1.4 on page 10) and the BSD license (see 2.1.4 on page 10) is presented in the literature chapter.

GPL - GNU General Public License

The GPL license was written by Richard Stallman for the GNU project. The GPL version 2 was released in 1991, and it is one of the most used Open Source licenses. Version 3 was released in June 2007. Unlike BSD, it has a strong copyleft clause that requires all derived work to be available under the same copyleft, thus ensuring that the code stays open source even when modified. This is in line with the FSFs philosophical thoughts on open source. Both FSF and OSI approve the license. Unlike the BSD license, linking from code with an incompatible license is not allowed. There have been several versions of the GPL published; the most current is version 3. The copyleft is in the GPL case triggered by the “distribution” clause. As long as modifications are kept private and not distributed it can be used in any setting.
LGPL - GNU Lesser General Public License

The LGPL license is another license published by the FSF. It’s sometimes referred to by its former name; the Library General Public License. The LGPL is a compromise between the copyleft licenses and the more permissive BSD-style licenses. The LGPL places a copyright on the program itself, but any program linking to the software is not required to be copyleft. The LGPL is suitable for libraries, though it is also used by other software. The LGPL is approved by both FSF and OSI.

AGPL - GNU Affero General Public License

The AGPL license is a license designed to close a loophole in the GPL license where the copyleft clause first triggers when the code is distributed. This is done through a clause covering network usage. This meant that privately modifying code, without redistributing it did not have to be published. This opened a loophole for application service providers to base services on GPL licensed code offering (public) services. The AGPL is compatible with the GPLv3 - AGPL licensed software can use libraries and code released under GPLv3. GPLv3 licensed code that doesn’t use the network may also be compatible with AGPL. The AGPL does not seem to be OSI approved as of December 2007.

Apache License

The Apache License was authored by the Apache Software Foundation. The Apache License version 2.0 is compatible with GPLv3, but does not have a copyleft statement. It does accept linking from code licensed under a different license. The Apache License is similar to the BSD in that the licensed software may be redistributed with modifications under a different license, but differs by requiring that a notice is supplied with the modified code stating that code licensed under the Apache license has been used. The Apache License is approved by both FSF and OSI.

MIT License (X11 license)

The MIT License was first drafted for usage by the X Window System project by Massachusetts Institute of Technology (MIT). It’s a permissive license, allowing linking from code with a different license, and it is not a copyleft license. It’s similar to the new BSD License, except for not prohibiting the use of the name of the copyright holder in promotion. The MIT License is GPL compatible and approved by both FSF and OSI.
Artistic License 2.0

The Artistic License was first written by Larry Wall and is used by e.g. the standard Perl implementation. The v2.0 of the Artistic License and the Clarified Artistic License is approved by the FSF. The license is GPL compatible. The license gives several options for developers that modifies the code and want to redistribute it, either you provide a copy back under the same terms or allow users of the modified code to do it, or it can be licensed under a different name and made sure that the modified version does not prevent the user from using the original version in any way. The Artistic License allows linking with other projects.

Open Software License 3.0 / Academic Free License 3.0

The OSL 3.0 license was written by Lawrence Rosen. The OSL is a copyleft license, but does not try to enforce that reciprocity on independently written software, meaning that it is more like LGPL than GPL. There is an almost identical version of the OSL 3.0 named Academic Free License 3.0 which does not contain the copyleft statements. Both licenses are approved by OSI, but the Free Software Foundation discourages the use of both licenses due to a problem with distributing software under the licenses: «Recent versions of the Open Software License have a term which requires distributors to try to obtain explicit assent to the license. This means that distributing OSL software on ordinary FTP sites, sending patches to ordinary mailing lists, or storing the software in an ordinary version control system, is arguably a violation of the license and would subject you to possible termination of the license. Thus, the Open Software License makes it very difficult to develop software using the ordinary tools of free software development. For this reason, and because it is incompatible with the GPL, we recommend that no version of the OSL be used for any software» [12].

7.2.1 Dual Licensing

Dual Licensing is a method used by some projects to license the same software under several licenses. The licensee of the software can choose which license to use, depending on which fit best for a certain purpose of usage. Dual licensing is often used as a business model, where open source developers may use the licensed software for free, provided that they accept a copyleft license and provide any modifications back to the community. Commercial proprietary solutions may then be licensed under a different license possibly for a fee, without any copyleft requirements. There may be several reasons for wanting to dual license your code, where license compatibility, increased market share and familiar-
ity with your products may be examples of advantages gained. Some see
this as an investment, where users that are familiar with the software
from an open source project may later choose to use it in a commercial
setting that requires a commercial license. An example is the MySQL
database which is licensed under a proprietary license and the GPL, with
a FLOSS License exception to allow linking from other OSI approved li-
censes [1].

In most cases the copyright holder is the only one able to change the
licensing terms of the software. This means that most dual licensed soft-
ware is copyrighted by a company or organization rather than a group
of individuals. As a committer on a project it's normal to transfer the
copyright to a single organization for a project to make such processes
easier to manage. If that's not the case, all committers may have to be
contacted and give their approval of such a license change.