NTNU

Norwegian University of
Science and Technology

Viable Open Source for the Consultancy Industry

Kristian Fredrik Klette

Master of Science in Informatics

Submission date: Januar 2012

Supervisor: Reidar Conradi, IDI

Co-supervisor: Rolf Sture Normann, Acando AS

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

The Norwegian communes will have to integrate their data systems with Uninett's FEIDE system over the next years. In order to do this, each commune needs a system for maintaining the users that should have access to resources protected by FEIDE. Acando AS has written a white paper for Uninett AS describing how such a system could be implemented and released as open source.

Acando has constructed a problem description consisting of two main parts, one part implementation and the other theoretical. The implementation part is to create an module for their OpenFEIDE solution that will allow regular users to easily add their own XML-based data sources to the OpenFEIDE import mechanism.

The theoretical part raises the following questions:

- Is authoring of open source software a viable business idea for consultancy agencies?
- How should software be released as open source?

The theoretical part should identify and decide upon key issues, both technical, economical and legal ones.

The final expected result is a first attempt at an open source project release, and a working prototype for the importer module.

Abstract

Open source software is growing in the market, and increasingly preferred to closed software for the increased flexibility free software provides. As a result of this more and more businesses are trying to enter this market and profit from open source software. Consultancy agencies targeting the public sector are in demand of expertise and products released as open source. As this is a new field for many companies, studies are needed on how to approach these markets with a high chance of success with regards to business models and the technological benefits that open source software may provide. The problem description raises two research questions:

- Is authoring of open source software a viable business idea for consultancy agencies?
- How should software be released as open source?

This thesis presents two main contributions for answering the research questions. The first is a set of guidelines and techniques for estimating the business viability of a of open source software venture. The second is some best practices for authoring and releasing open source by observing the successful projects that already exists. In addition to these theoretical parts of the thesis, a system for analyzing and generating XSLT-transformations for OpenFEIDE is presented.

Preface and Acknowledgements

This Master's thesis is the final part of a Master of Science degree from the Department of Computer and Information Science (IDI) at the Norwegian University of Science and Technology (NTNU).

Acknowledgements

I would like to thank my supervisor Reidar Conradi for his support, guidance and patience.

I would also like to thank all my friends in the open source community who have been very helpful in the gathering of information, and correcting me if any mistakes were made.

January 15, 2012
Kristian Klette

Contents

List of Figures

List of Tables

Acronyms

API Application programming interface 31

BAS Brukeradministrativt system (eng: user management system) 40

CBSE Component Based Software Engineering 17

COTS Commercial of the shelves software 1

EPL Eclipse Public License xxvii

FEIDE Felles Elektronisk IDEntitet 2

FSF Free Software Foundation 15

FTP File transfer protocol 27

GPL GNU Public License 14

GWT Google Web Toolkit 11

IDE Integrated Development Environment 84

IRC Internet Relay Chat 26

Chapter 1

Introduction

1.1 Motivation

Open Source Software (OSS) is software where the source code is available for the consumer to inspect, and given the right license modify and redistribute. This type of software gives the consumer greater power with regards to how it wants to use the software since there is no vendor lock-in occurring. Since OSS is in most cases provided free of charge, and the software can be modified to suite the consumers needs, OSS is becoming increasingly popular in the software community. Companies and governments are also starting to take an interest as a mean to reduce rising IT costs. In Norwegian state run facilities the adoption of OSS usage has grown from 35% to almost 60% from 2006 to 2008 according to Statistisk sentralbyrå (Statistics Norway) (SSB), This trend forces consultancies to take OSS seriously as a part of the strategies for selling consultancy contracts with the government.

For established software houses and consultancy firms OSS is a radical new way of developing and profiting from software. As OSS is fairly new in the mainstream software industry there are not a lot of well established best practices and monetizing strategies available. While the adoption of OSS component usage in the Norwegian consultancies, as shown in "Adoption of Open Source in the Software Industry" [13], is prominent, the development and maintenance of OSS are not. This is not surprising as consultancies sell services and not Commercial of the shelves software (COTS) as traditional software houses do, and often do not want the risk involved in maintaining software, unless on commission by a client.

The motivation of this study is to research some potential best practices and monetizing strategies for consultancies authoring open source software. This includes the technical, legal and business aspects of these types of projects.

1.2 Actual Context

Acando AS is a consultancy agency based in Oslo and Trondheim and works mainly against the public sector. They are in the process of creating their first OSS effort with the product Acando's open source FEIDE BAS (OpenFEIDE). OpenFEIDE is a user management system specialized against Felles Elektronisk IDEntitet (FEIDE). FEIDE is a federated identity management system that provides users with a single set of username and password they can use against all services in the public school sector in Norway. Most universities and colleges are already using FEIDE, with other user management systems. But there is a potential market in the primary and secondary schools. This schools are run by Norwegian municipalities.

1.3 Problem Statement

The problem description has two parts that the initiators of this thesis wants done. The first is a theoretical study of the practical and business aspects of authoring OSS. The other is the implementation of an import module to their open source log-in system for usage for the FEIDE platform.

1.3.1 Research Questions

The problem description raised two research questions. RQ1 is based on the business side of OSS. Can be profitable for a consultancy agency to invest in the authoring of OSS, and in return provide consultancy services to the users of the software? RQ2 is about the practical side of the authoring of and releasing OSS. How and when should the software be released in an open source setting? RQ2 includes the technical, economical and legal issues with releasing software.

RQ-1 Is authoring of OSS a viable business idea for consultancy agencies?

RQ-2 How should software be released as open source?

1.3.2 Purpose of Implementation

The initiators of the thesis would like to have an module for helping ordinary users add data sources to their OpenFEIDE solution, without having to have any technical background. The data sources are exports from internal databases (Skoleadministrative systemer (eng: school administration systems)

(SAS)-systems) and may vary in their structure and content. In the existing solution OpenFEIDE must have a predefined XSL-Transformation for the given data sources for OpenFEIDE to understand the data.

The implementation problem is how to create a module that helps these users to create such an transformation without having to know anything about the technology behind it.

1.4 Contributions

From the exploration of this thesis the following main contributions are created:

- Collected methods and tools for determine the viability of open source business ideas, including financial and legal issues.
- Collected and analyzed development practices of successful open source projects.
- Prototype implementation of XSLT generating software for the OpenFEIDE project.

The purpose of the first two contribution is to give the reader a broad understanding of how both the business and technical aspects surround open source software works. Business aspects such as business models, viability estimation, legal issues with licenses are discussed, as well as how to do open source development in a practical and efficient matter. These practices includes how to manage source code, bug tracking and user contributions in an open environment.

The last contribution aims to provide a prototype XSLT generator implementation for OpenFEIDE's import system, and an analysis of the problem area of the implementation.

1.5 Thesis Structure

Chapter 1: Introduction

Chapter 1 gives an introduction to the premises of this thesis, and what it is trying to accomplish.

Chapter 2: Background

Chapter 2 illustrates the current state of the art for open source development, from the software engineering view point and open innovation to business models.

Chapter 3: Implementation Background

Chapter 3 gives an overview of the current state of technology related to the implementation part of this thesis by describing technologies like Extensible Markup Language (XML) and XSLT, and current attempts to analyze such documents.

Chapter 4: Research Design

Chapter 4 provides the context of the study and the perspective taken for this thesis, and how the research was designed and performed.

Chapter 5: Contributions

Chapter 5 introduces the reader to the research methods used in this thesis to establish the data needed for proposing answers to the research questions. This is split into two part, one for each research questions.

Chapter 5 also includes the description of the implementation part of this thesis.

Chapter 6: Evaluation and Discussion

Chapter 6 discusses the contributions made in this thesis.

Chapter 7: Conclusion

Chapter 10 provides an conclusion made from creating this thesis in the context of the research questions.

Chapter 8: Further Work

Chapter 11 describes the missing pieces of the thesis and how they should be addressed.

1.6 Appendices

In addition to the 8 chapters, 9 appendices were created.

Appendix A and B presents case studies done on two business founded on open source software. Appendix C, D, E, F and G are case studies of open source projects created for mapping how different projects organize and run themselves. Appendix H presents a recommended path of OpenFEIDE to become a more open project, and how to get there. Appendix I provides some documentation for the implementation part of the thesis.

CHAPTER 2

BACKGROUND

For solving the research questions posed by this thesis one must have a good understanding of how the open source world works, and how it differs from traditional development. This chapter aims to explore the different aspects of Open Source development, both the technical, the legal and the business aspects.

2.1 Software Engineering

2.1.1 Background

The term Software Engineering comes from the first conference on software development held by North Atlantic Treaty Organization (NATO) in 1968 and its choosing was described by Naur and Randell [26] as follows:

> The phrase "software engineering" was deliberately chosen as being provocative, in implying the need for software manufacture to be based on the types of theoretical foundations and practical disciplines, that are traditional in the established branches of engineering.

Software engineering was not formally defined until IEEE-standard 610.12-1990 [32] released in 1990 and states the following:

> (1) The application of a systemic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.
>
> (2) The study of approaches as in (1).

The definition itself is very general, but it's content can be broken down to activities occurring in software development. These activities are concept analysis, requirement analysis, design, implementation, testing, integration

and maintenance [7]. But there is still no firm engineering practice with regards to software development. The field has however matured by developing tools and processes to aid performing the above activities. There are doubts if there will ever be sound engineering principals in software. Lyu states that software engineering is fundamentally flawed as an engineering discipline as decisions are made using human judgment and bias, and not laws and required process as with other established engineering principles [21].

2.1.2 Development Life Cycles

There are two well established life cycles in Software Engineering, the waterfall cycle and agile.

Waterfall

The waterfall model is a strict sequential life cycle bound to the activities defined by Braud [7]. The model first appeared in the paper "Managing the Development of Large Software Systems" by Dr. Winston W. Royce in 1970 as a flawed way of developing software. According to Royce this model is flawed because of the assumptions it makes on the environment of software engineering;

- All the essential requirements for future users are known.
- The requirements are not changed significantly during the process.
- The system can be developed entirely in one sequence of activities.
- The system will not change the requirements of any external system.

These assumptions boil down to that once a system has been designed it cannot be changed in the process of making it. It fairly well documented that this way of developing software was only attractive to managers as it simplified the process by reducing it to a known mechanical process, thus removing risk by making the assumptions that all situations have been though of in the design process.

The life cycle is separated into the activities shown in figure 2.1.

The first activity in the life cycle is the requirements phase. In this phase all the functional and non-functional requirements of the final product are created and specified as detailed as possible. After the requirements phase the

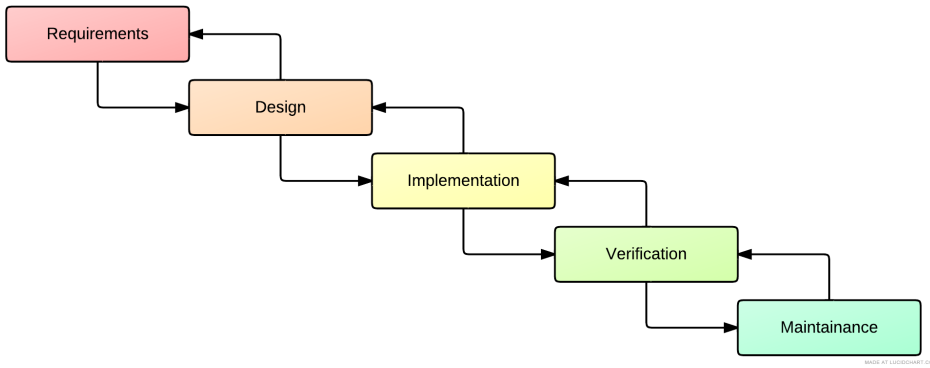

Figure 2.1: The waterfall life cycle activities

design process starts, in which the system is designed. This includes the architecture, data models, interactions, interfaces and so fort. When the system is designed it's implemented as according to the given specifications. The last phase during development is the verification stage where the system is verified to be according to the specifications. After this is enters the maintenance phase which lasts for the duration of the product life time.

Agile life cycles

Agile software development is often described as a lightweight incremental development model, as opposed to the heavyweights like the waterfall model, and often referred to as "agile". The term "agile" comes from the model's ability to handle change during the development process. This way of developing was first formalized by E. A. Edmonds in his paper "A Process for the Development of Software for Nontechnical Users as an Adaptive System" [11] in 1974, but under the name "adaptive software development". As additional models for this type of development was introduced, such as Scrum (1995) and Extreme Programming (1996), the need for a common umbrella became apparent. In 2001 the Agile Manifesto [12] was created and the term "Agile Software Development" was born. The Agile manifesto states:

> We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:
>
> - Individuals and interactions over processes and tools

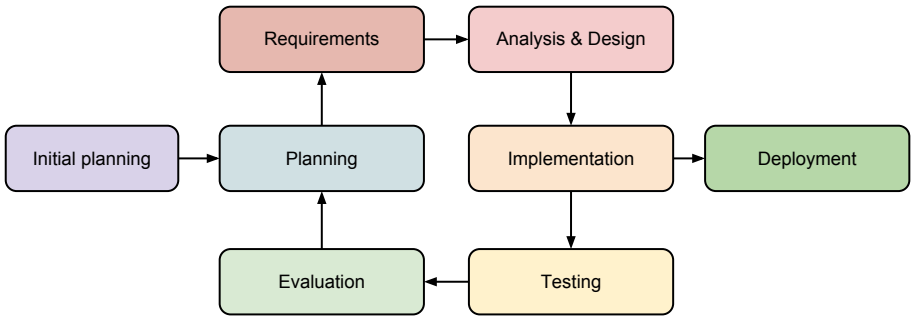

Figure 2.2: The agile life cycle activities

- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

That is, while there is value in the items on the right, we value the items on the left more.

The activities in agile are for the most part the same as in the waterfall model, but instead of running the sequence once, it loops to embrace change during the development. There are of course variants of this depending on which of the agile development models one uses, but the primary concepts are the same. The loops or iterations of each step is represented in figure 2.2

2.1.3 Software Quality

Software quality is standardized as ISO/IEC 9126-1:2001 [15], and defined as:

> (1) The degree to which a system, component, or process meets specified requirements.
> (2) The degree to which a system, component, or process meets customer or user needs or expectations.

[32]

The two parts of the definition are different as (1) is a fixed set of goals to reach quality based on the predefined requirements, whilst (2) is concerned with the perceived quality of the product from a user/customer perspective.

Both of these are important to remain competitive in software development, especially in business orientated software were the risks of delivering low quality have greater hazards.

Achieving high quality software is however a costly process. Osterweil estimated that at least 50-60% of the effort developing software is spent on software quality issues [29]. Slaughter et al. divided these issues into two groups; conformance and non-conformance. Conformance being controlling defects up front, whilst non-conformance being introduced defects and other failures (both internal and external) [31]. One of the reasons why software quality is expensive is due to the fact that is has to be completely integrated in the product, and not added after the main development has been completed. Quality with regards to security and reliability are global issues in the software and must be thoroughly thought through during the entire development process. Normal software defects are considered local and may be corrected later in the process, but not without the risk of introducing new undefined states which might result in new defects.

2.2 Open Source

Open source is the generic term for software released under licenses accepted by the free software foundation (FSF) and the Open Source Institute (OSI). The term open source was coined by Eric S. Raymond to give free software a more acceptable name for businesses in the early days of open source, as the word "free" was a hard sell for adoption in businesses since the word "free" is more commonly used as a term for a price tag of zero. Technically is "free software" determined by the acceptance from the FSF, and "open source" by acceptance from the OSI. The two terms are often mixed however, and rarely differentiated in the FLOSS-communities.

2.2.1 Conditions of Open Source Software

For software to be considered as open source its distribution must conform to the ten key conditions set by the Open Source Definition (OSD)[1] created by the Open Source Initiative (OSI). The conditions are as follows:

[1] `http://opensource.org/osd.html`

Free Redistribution

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

This condition prevents the authors demanding royalties for further distribution of the open source licensed software. The results of this is that external contributors to the software are guaranteed not to have to pay royalties for the product they are helping create, as well as a safe guard for the users of the software that they may not be forced to pay royalties or fees when they redistribute their own version of it, or bundled with their own software. OSI gives the following rationale for the condition:

> *By constraining the license to require free redistribution, we eliminate the temptation to throw away many long-term gains in order to make a few short-term sales dollars. If we didn't do this, there would be lots of pressure for cooperators to defect.*

Source Code

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

This conditions requires the authors of the software to make the source code available for the users of the software. Either by bundling in with the distribution, or as a separate package. It also requires the provided source code to be in its original form. As an example, if one was to create an open source application using Google Web Toolkit (GWT), the original java source code must be provided, and not just the produced JavaScript code from the compiler. The purpose of this is to ensure that the user has the possibility to modify the software with ease, and thus preventing vendor lock-in.

Derived Works

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

This condition ensures the users right to modify and redistribute the software, given that the user distributes his or her version under the same license as the original software. This is perhaps one of the key points of OSS, the possibility of modifying existing software and redistributing it. This provides the basis for rapid evolution and innovation to happen.

Integrity of The Author's Source Code

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

This condition requires the license to give the author of the software the right to restrict the usage of his software to protect the integrity of the product. One example of this is Donald Knuth's TeX project. It is an open source project and thus allows redistribution of modified version, but it's license requires the modified version not to use the name TeX.

OSI gives the following rationale for the condition:

> *Encouraging lots of improvement is a good thing, but users have a right to know who is responsible for the software they are using. Authors and maintainers have reciprocal right to know what they're being asked to support and protect their reputations.*
> *Accordingly, an open-source license must guarantee that source be readily available, but may require that it be distributed as pristine base sources plus patches. In this way, "unofficial" changes can be made available but readily distinguished from the base source.*

No Discrimination Against Persons or Groups

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

This condition requires that the license does not pose restrictions on whom might use the licensed software. This however is some cases in contradiction with the laws of some countries. On example of this is the United states which have export restriction on some types of software, such of software dealing with cryptography. But, even with this types of law, the OSD requires the software license not to restrict the usage in these matters.

No Discrimination Against Fields of Endeavor

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

This condition requires the license to pose no restriction on how the software should be used.

Distribution of License

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

This condition requires the license to not to restrict the software by requiring the agreement of additional licenses. The most common version of this is requiring the user to sign a non-disclosure agreement for either using the software or source code.

License Must Not Be Specific to a Product

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

This condition requires the license not to trap the user of some piece of software to any other. In other words, a license may not require that the license in only valid if the software is used in conjunction with another piece of software.

License Must Not Restrict Other Software

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

This condition requires the license not to impose restriction on how the software is distributed. In other words, the license may not require that all software distributed as a package is under the same license. There are some subtleties here with regards to the GNU Public License (GPL)-family of licenses as they require all linked software to have a compatible license. The distinction is in the difference between linked and bundled. The GPL clause is only in effect if the if the software forms a single piece of work.

License Must Be Technology-Neutral

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

This condition requires that the license and the software licensed under it, is not dependent on a specific way of distributions. This allows the software to be redistributed in any way that is reasonable for the software in hand.

The OSI gives the following rationale for the condition:

> *This provision is aimed specifically at licenses which require an explicit gesture of assent in order to establish a contract between licensor and licensee. Provisions mandating so-called "click-wrap" may conflict with important methods of software distribution such as FTP download, CD-ROM anthologies, and web mirroring; such provisions may also hinder code re-use. Conferment licenses must allow for the possibility that (a) redistribution of the software will take place over non-Web channels that do not support click-wrapping of the download, and that (b) the covered code (or re-used portions of covered code) may run in a non-GUI environment that cannot support pop-up dialogs.*

2.2.2 History of Open Source

In the very early days of the computer industry software was in fact open source as the software was freely distributed and could be modified by anyone. Money was not made on the software, but the hardware. As the adoption of computers increased, software gained value and the industry started to close their software and restrict usage of it. Richard Stallmann then created the first version of the GNU General Public License, commonly known as the GPL, in an attempt to protect the freedom of the software users to modify and share their work. Until around 1999 open source was mainly only seen as a part of a hacker culture, but as the software outcomes from this environment proved useful for business environments commercial applications of open source gained momentum. In late 1998 the OSI was founded by Eric Raymond and Bruce Perens [14] as an advocacy organization for OSS. It was different to the Free Software Foundation (FSF) in the methodology and attitude, as it's goals were not to be a high profile organization, but rather work quietly in the background with an pragmatic approach, as opposed to the FSF more philosophical hard liner approach. Magnus Sulland created a graphical representation [34] of some of the key points in the history of OSS shown in figure 2.3.

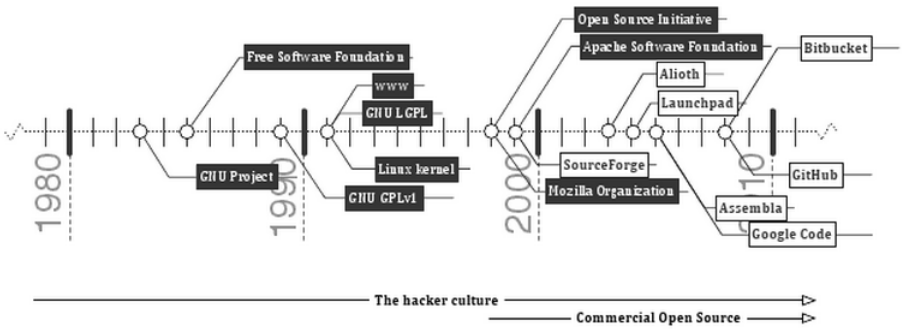

Figure 2.3: Timeline of key points in OSS history

2.2.3 Open Source Culture

Open source is, in its most basic form, a sharing community driven by personal motivation - either by solving a problem for one self, or solving problems for the technical challenge. Some open source supporters are involved due to the

philosophical aspects, whilst others see open source as a pragmatic methodology of creating software. Two main characters on each of these aspects are Richard Stallmann and Linus Torvalds. Stallmann, as the founder of the FSF is highly invested in the philosophical aspects, whilst Linus Torvalds (author of the Linux kernel) sees open source as a pragmatic way of programming. The latter is a prime example of an individual who is driven by technical challenges. This type of persons are the most common ones in the open source community.

In later years the large open source projects are increasingly influenced, and driven by companies donating developer time to them, and may thus reduce the level of personal commitment to open source that once was. The Linux kernel project is a good example of this. In their paper "Influence in the Linux Kernel Community" by Timo Aaltonen et al. [2] a study of which companies that had influence over the development was performed, and shows that a lot, if not most, of the development is done by commission in large companies that uses Linux in their products.

There is however still a lot of uninfluenced work being done by hobbyist and professionals alike in there own free time, purely for their own need or for the challenge.

2.2.4 Open Source Software Engineering

Most OSS is not created using strict engineering practices, and could be described as using incremental development cycles by the pure nature of how OSS is created. In the normal case OSS projects are started as either small experiments or tools to solve a single developer's problem. This is known as "scratching the itch". As a project grows and more developers are involved the software is structured as the developers see fit, and during the growing phases often redesigned for either increased quality or to easier fit new extensions. As numbers grow, developers with various interests joins, including architects and so forth. This adds expertise to the projects and helps drive quality of design, security, and code.

Component based software engineering

One key factor of most open source systems are that they depend heavily on external components. Key components have a high usage grade, thus enforcing quality by the components being tested in many different settings and environments. Open source makes it possible for users to submit bugs and patches.

The Parastoo et al. [23] study of reused software components showed that, in the given case, the defect density was 50% less in reused software components than in non-reused components. It also showed that reused components were given higher priority when amending defects. Component Based Software Engineering (CBSE) is, regardless of its strengths, not free of risk---especially in OSS. Due to the large quantities of available components, choosing the right one can be a challenge and due precautions must be made. The selection process is closely related to COTS selections. Jingyue Li et al. discovered two popular selection processes in a study of COTS in Norwegian IT companies [20]. The "familiarity-based" selection process and "process combining Internet searches with hands-on trials". The former process relies on past experiences by the project members or external experts, while the latter is a more manual trial and error approach.

2.3 Open Innovation

Open Innovation is the practice of innovating in the open, i.e. in a public manner. This is also called the Bazaar model, as coined in the book "The Cathedral and The Bazaar" by Eric S. Raymond. Open innovation and open source differs in some ways with regards to legal issues, but the innovation part is similar. The "problem" with open innovation and open source in combination is how to profit from your innovations without encumbering the innovations and technology with proprietary boundaries such as patents (and thus closing the innovation). In the general case of OSS and Open innovation, the technology created is in itself not valuable from a monetization point of view, but the knowledge and expertise of it is. Open innovation was first thoroughly examined in "Open innovation: the new imperative for creating and profiting from technology" [8] and described why closed innovation was staggering and what could be done about it by using the alternate Open Innovation model. The two innovation models are at the core two different

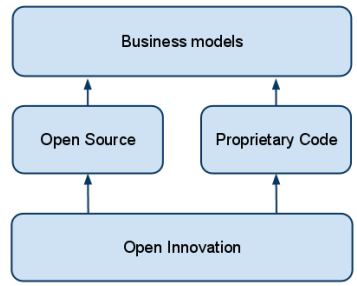

Figure 2.4: Relationship between open innovation, OSS and business models

philosophical models. Chesbrough highlighted this using table 2.1 of the principles of open and closed innovation. The relationship between OSS and open innovation is, despite its similarities, not a fixed one. As shown in figure 2.4 Open Innovation does not have to result in an open/free product, and therefor also have impacts on the business cases.

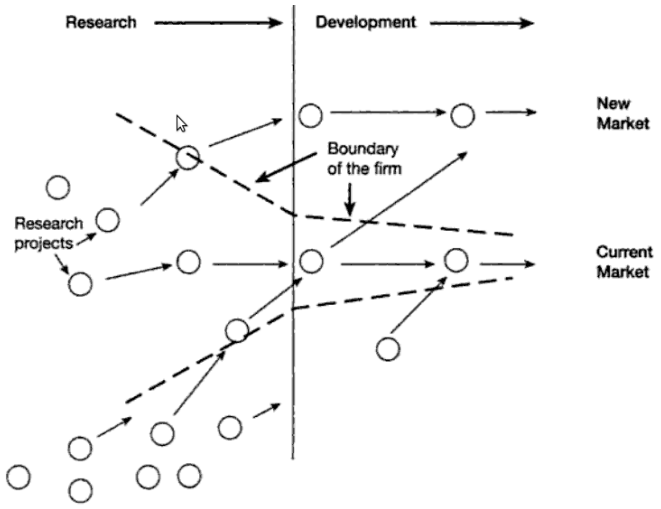

Figure 2.5: Open Innovation Paradigm for managing R&D

The notion of that the innovation in it self is not the value, but the expertise that the business had about it, is similar to how consultancies work in a day to day basis, as they sell expert services, and not a product in it self. There are several strategies for monetizing this expertise, as will be shown below. Munga and Fogwill divides these into two main sections, which have further subdivisions. The two main sections denote whether the strategies are product related, or service related.

Closed Innovation Principles	**Open Innovation Principles**
The smart people in our field works for us.	Not all smart people work for us. We need to work with smart people outside our company.
To profit from R&D, we must discover it, develop it and ship it our self.	External R&D can create significant value; Internal R&D is needed to claim some portion of that value.
If we discover it ourselves, we will get it to market first.	We don't have to originate the research to profit from it
The company that gets an innovation to the market first will win.	Building a better business model is better than getting to market first.
If we create the most and the best ideas in the industry, we will win.	If we make the best use of internal and external ideas, we will win.
We should control our IP[a], so that our competitors don't profit from our ideas.	We should profit from others' use of our IP, and we should buy others' IP whenever it advances our own business model.

Table 2.1: Innovation Principles

[a] Intellectual Property

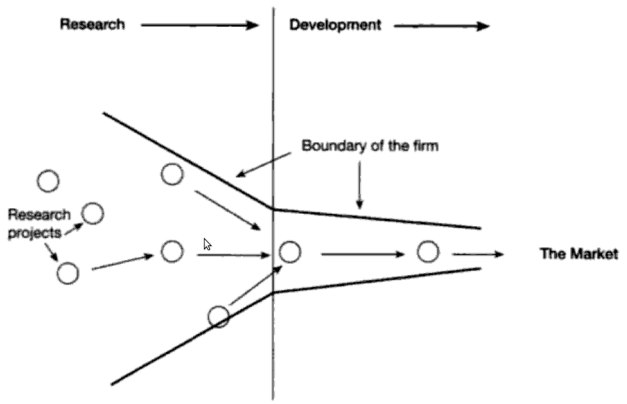

Figure 2.6: Closed Innovation Paradigm for managing R&D

2.4 Business Models in Open Innovation Environments

Osterwalder et al. defines business models as something that describes the rationale of how an organization creates, delivers, and captures value [28]. With open source and open innovation, the technology is not as valuable as the knowledge about it. This imposes some restrictions on how to model the business strategies. There are no restrictions on selling free software, but as the licenses allow redistribution of the software as the user sees fit the business must go beyond just selling the software to add the appropriate amount of value for the customer to select one as a business partner.

There are two main categories of means to appropriate returns on OSS: products and services, as classified by Dahlander(2004) [9]. Jan Fredrik Stoveland expanded the original table suggested by Dahlander in his thesis paper "Managing Firm-Sponsored Open Source Communities" [33] to from table 2.2

2.4.1 Product Based Business Models

Out of the four business models listed in table 2.2 under the products category, only three are relevant for this thesis, as we are dealing with pure software development, not hardware.

Category	Means	Description
Products	Packaging	Facilitating distribution and ease of use
	Proprietizing	Adding proprietary extensions
	Spin-off	Create proprietary products based on the software
	Black-box	Integration with hardware
Services	Education and training	Courses, certifications
	Consultancy	Consultancy work based on the expertise in the product
	Support	General support with or without a Service Level Agreement (SLA)

Table 2.2: Means to appropriate returns

Packaging

Packaging of software and adding value to the product as usability and ease of maintenance. Red Hat[2] is one example of businesses providing this service. In the case of Red Hat they offer this as a part of their support packages. Tested software packages, and tested installation and upgrade paths are worth a lot for companies either with a large environment where doing the testing them self would be very time consuming and costly, but also small environments where there might not even be a dedicated IT department, and user friendly installation and upgrading is crucial for the customer.

Proprietizing

Proprietizing open source products is the process of adding modules, or extensions, to an open source product as proprietary software. This requires some forethought as with licenses like the GPL this might be troublesome depending on the nature of the extensions. Despite of this, and the FSF's disliking of this way of dealing with free software, it is a quite popular business model. The most recent switch to this model by a major open source project is Oracle's purchase and new business models for the MySQL[3]-database, in which they

[2] http://www.redhat.com
[3] http://www.mysql.com

offer enterprise grade extensions in addition to the base OSS product.

Spin-off

Spin-off is the practice of creating proprietary products which is based on your OSS product. One could image creating an open source database system and a proprietary accounting system built upon and exclusive to this database. This allows for complete control over the software stack, and seemingly reduced cost for the customer as they only pay for the accounting software, and not the database. The providers however get a larger deployment of this database and is in a position to create more spin-offs based upon it.

2.4.2 Service Based Business Models

Service based business models are the models which does not include the product in any way, but provides services around a product or a set of products.

Education and training

Large computer systems are often complex and requires some skill to use and maintain. This allows for the business of education to be relevant for the customer. Whether it's for training the end-users of the program in usage and best practices, or the IT department for operations, this model is ever more popular as IT accesses every part of business operations. Under this model we can also place the creation and selling of books and other documentation that is not a part of the product itself. In very popular system one can create certification courses. Again is Red Hat a prime example of doing this. They provide an extensive array of training courses and certifications all which have become recognized by the industry as high quality, thus increasing its popularity and Red Hat's income.

Consultancy

Consultancy work is a wide area and might include just about anything. From development of integration software, add-ons, customizing and so forth, to providing hosting and operations for the product with service level agreements of various degrees (and various degrees of cost for the customer). This business model does require a high level of understanding of the software in

hand, and if hosting is provided, additional operational costs. In Norway the consultancy industry is the primary actor in the IT industry with an approximate 80% share of the IT sector.

Support

Support is by far the most common service business model in OSS. Most customer would like to have someone to call if something goes wrong, or they have any other need for help. From the customer's point of view they have the reassurance that help is just a phone call, or email, away. From the provider's point of view this can be a lucrative deal, as call centers (for large providers) can easily be outsourced at a low cost, or for small providers provide tight contact with their customers and build valuable business relationships, increasing the goodwill of the customer that might recommend your services to other businesses, thus expanding your customer base.

2.4.3 Implementation of Business Models

The mentioned business models are usually not exclusive to each other, and often sold as a package deal. Most large open source projects sponsored by a firm provides either alone or trough partners all of the above models.

In "Entry Strategies Under Competing Standards: Hybrid Business Models in the Open Source Software Industry" [5] the researcher's firm sample produced the results listed in table 2.3, and shows the multi-business model approach usage in the market.

Business model (service)	Companies supplying the service
Development of ad hoc solutions	87.7%
Consulting	84.9%
Support	82.9%
Installation	80.1%
Maintenance	76.0%
Training	64.6%
Distribution	63.0%
R&D	53.4%
Marketing of software produced by other companies	39%

Table 2.3: Business model usage data from Bonaccorsi et al. [5]

2.5 Licensing

OSS licenses are copyleft licenses which are in opposition of copyright licenses by enabling the users to modify the source code as they please, but under some rules. The first OSS license was the GNU General Public License which was released by Richard Stallmann in 1989 as mean to ensure the freedom of a piece of software.

OSS licenses are however difficult due to the restriction they impose as a part of ensuring the protection of the authors, as opposed to copyright licenses where you have to have the permission from the copyright holder to use and modifying the software. The choice of a license for a new product imposes restrictions on how the authors may use the software in the future, though this is in the general case only a problem once others have contributed to the project under the premises of the given license.

As there are a lot of different licenses to choose from, and it's not always clear which one is most suited for your project. Goals are to present some of the major licenses that have been tried in court (most OSS licenses have never been tried), and what the implications of choosing one are with regards to future development and legal protection.

A figure of the relationship and grouping of the most popular licenses can be seen in figure 2.7. In this figure the licenses are divided into four categories, ranging from weaker to stronger copyleft clauses. The arrows between the different licenses indicates that the licenses are are compatible.

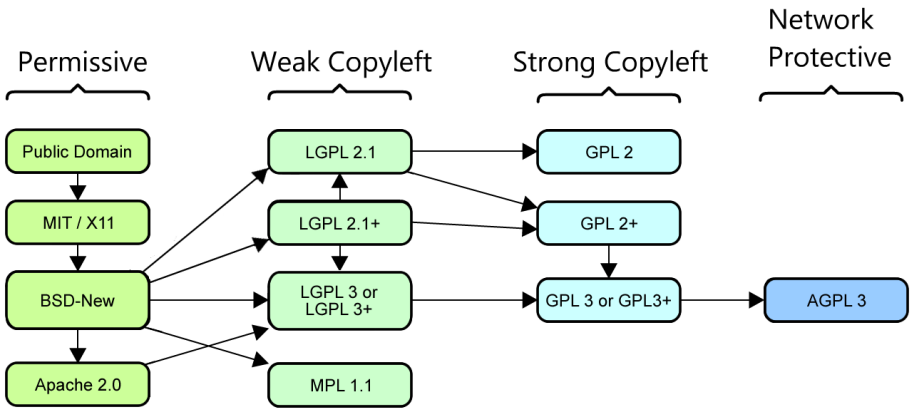

Figure 2.7: OSS license relationships by Fredrik Speakman adapted from David A. Wheeler

The main difference between OSS licenses are the level of protection for their own interpretation of freedom. The most discussed, and arguably the most common problem is the GPLs restrictions on being used by products with other licenses. As GPLv2 is by far the most used license for OSS choosing a license that is not compatible with the GPL (thus disallowing linking with GPL licensed software), can severely restrict both your choices in libraries and the adoption of your software. Choosing the GPLv2 as your license however does impose some restrictions on the future of the project. If the provider (person or organization) have not received any contribution from others, it can change the license for future version at will. If they have all contributors must agree on the change. In this case all the older version are still under the previous license however. This is in particular an interesting point if the providers have plans to sell proprietary "plug-ins" to the product. Proprietary "plug-ins" can be created for GPL licensed software, but requires a less integrated approach than one might like[4].

2.6 Release Strategies

Releasing new OSS products to the world is, according to best practices, quite different from the established norm in the proprietary software world. Eric

[4] "Plug-ins must be run as separate executable and not share data-structures with the original software"

S. Raymond coined the term "Release Early, Release Often. And listen to your customers" in his book "The Cathedral and the Bazaar" [30], which states that you should release as often as possible, even if the product is incomplete and buggy. This is a part of the OSS mantra of sharing and collaboration. It is difficult to include external developers in the decision making if development is done behind closed doors, thus discourage collaboration and involvement from the community. If the goal of releasing a product as open source is to involve others in its development, not only with regards to fixing flaws, but also innovation of new features, then, according to Raymond, the project needs a transparent development model. The key factor of the "Release Early, Release Often" mantra is the "And listen to your customers" part. Public mailing lists, Internet Relay Chat (IRC) channels and bug tracking software is the most common forms of communication. Another step needed to facilitate developer collaboration is public source code repositories. There are a lot of tools available for these purposes. From major projects like SourceForge[5] which is to date the largest repository of free software projects, to smaller projects like GitHub[6] and BitBucket[7]. The latter are projects that evolved around new source version control software that are optimized for distributed collaboration, which the older and larger projects did not support.

According to a study by Andrea Bonaccorsi and Cristina Rossi [6], one the biggest motivation for firms to provide OSS is to gain contributions and feedback from the free software community, thus stating the importance of doing community management right. In Stoveland paper about firm-sponsored open source communities [33] he describes the methods which Novell uses for handling the balance between controlling their products and the open development model. In that particular case Novell uses the transparent development model mentioned above extensively, and are so far successful in regards of maintaining a community whilst controlling the prouct by investing in its development.

[5] http://www.sourceforge.net
[6] http://www.github.com
[7] http://www.bitbucket.org

Chapter 3

Implementation Background

This chapter presents some of the needed background information needed for understanding the implementation part of this thesis. It includes a description of the existing solution which the implementation will try to improve on, as well as the technologies used in the implementation, such as XML, XSLT and SAX.

3.1 The OpenFEIDE Import Mechanism

The responsibility of OpenFEIDE is to convert data from internal system of the user, to a format that is compatible with FEIDE. To do this job it must have some sort of data which it can expose. In the usual deployment, this data comes from SAS-systems.

OpenFEIDE imports data from these systems using a scheduling system that retrieves XML documents from a given File transfer protocol (FTP)-server.

One of the problems with this approach is that the XML documents retrieved from these data sources may vary in their format, and the import logic would therefor be extremely complex if it was to support every thinkable format out there.

To solve this by using XSLT transformations on the imported data for converting the documents to a XML schema that OpenFEIDE understands. The process is illustrated in figure 3.1.

These XSLT stylesheets must be made for each data source since the data is likely to be different. This is also the basis for the implementation part of the contributions in this thesis, as a means to see if the process of creating these stylesheets can be done either automatically, or by some sort of user interaction from regular users.

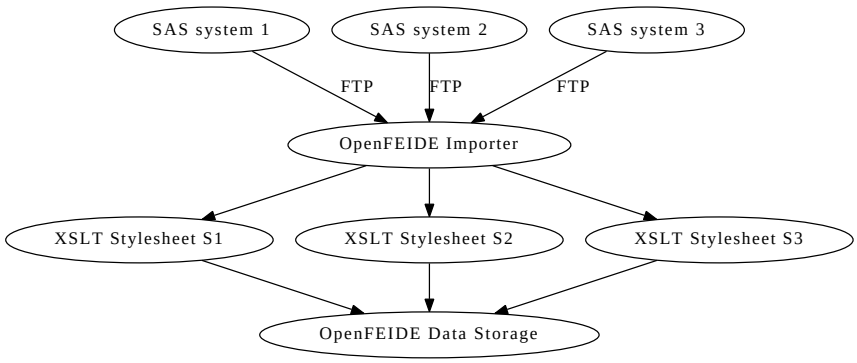

Figure 3.1: OpenFEIDE import process

3.2 XML Processing

XML is a markup language that was created as a simple machine readable document format, that also could be read by humans. The first release was done by the World Wide Web Consortium (W3C) in 1996[1]. The standard has been updated five times since then, and the latest and fifth edition was released in 2008. The specification includes ten design goals for the format:

1. XML shall be straightforwardly usable over the Internet.
2. XML shall support a wide variety of applications.
3. XML shall be compatible with SGML.
4. It shall be easy to write programs which process XML documents.
5. The number of optional features in XML is to be kept to the absolute minimum, ideally zero.
6. XML documents should be human-legible and reasonably clear.
7. The XML design should be prepared quickly.
8. The design of XML shall be formal and concise.
9. XML documents shall be easy to create.
10. Terseness in XML markup is of minimal importance.

[1] `http://www.w3.org/TR/xml/#sec-origin-goals`

The core of XML consist of `elements`, `attributes` and `character data`. `Elements` are a logical component. `Attributes` are descriptors fields that can be added to an element for giving the element additional info, while `character data` is used for the element names, defining the attributes as well as the content of elements. Attributes and character data are considered to be a child of an element, but element may also have other elements as children. This allows the format to represent hierarchical data thus creating way of logically structuring information inside a document. The hierarchical structure allows us to treat XML documents as a tree. The XML documented in figure 3.2 may thus be represented as the tree shown in figure 3.3. This is an important concept when we start thinking about extracting information from the XML document, and which approaches for doing so is suitable for the task at hand.

```
1  <root>
2    <persons>
3      <person>
4       <username>alice</user>
5       <email>alice@example.com</email>
6      </person>
7    </persons>
8    <groups>
9      <group>
10       <name>administrators</name>
11       <members>
12            <username>alice@example.com</username>
13       </members>
14     </group>
15   </group>
16 </root>
```

Figure 3.2: Example XML document for tree representation

3.2.1 XML Traversal

There are three main methods of dealing with XML documents in terms of programming interface: Simple API for XML (SAX), Pull parsing and Document Object Model (DOM).

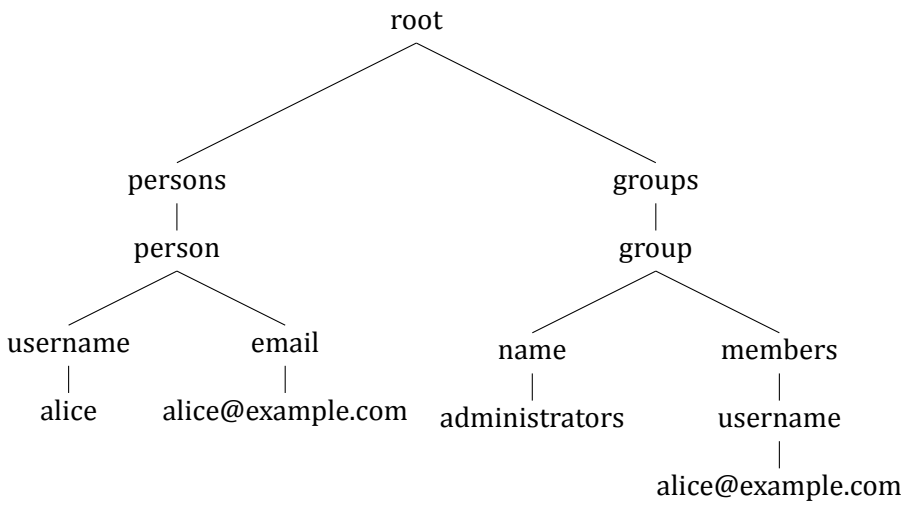

Figure 3.3: Tree representation of XML document

SAX Interface

The SAX interface is an event-driven method of reading XML documents. It works by reading the XML document serially and invoking callback methods when an event arises. These events occur when the state changes i.e. a new element is read. There are a number of different events that causes callbacks to be invoked. The standard Java SAX API provides 16 different events in their default SAX-handler. Some of the most frequently used ones are listed in table 3.1, along with the data that is available to the invoked callback-method. One important aspect of SAX is the fact that the callback methods only know of the data that is given to them. As a result of this and the serial way of reading, the application using SAX must keep track of the contextual information, if it need it, on its own. This lack of internal context has some drawbacks and some benefits. The drawbacks are that reverse traversal is not possible and many of the common operations needed for extracting information must be dealt with by the application. The primary benefits however is that its highly efficient in terms of CPU and memory usage. Memory usage is kept down by not keeping a history of previously parsed elements, and the CPU usage can be kept to a minimum since the application can choose which elements to process. One other benefit of this model comes from the fact that processing is done as it traverses the document. This makes SAX a very good method of dealing with streaming data sources since the data never have to end for the application to

Event	Available data	Description
`startDocument`	None	Start of document
`endDocument`	None	End of document
`startElement`	Address, QName, Local name, Attributes	Start of element
`endElement`	Address, QName, Local name	End of element
`characters`	List of characters read	Character data read

Table 3.1: Frequently used Java SAX API Callback Methods

be useful.

DOM Interface

The DOM interface represents the entire XML document as tree. It does do by traversing the entire XML document and building a tree structure of it in memory. When the initial traversal is done, the application may access and traverse any part of the document in memory. This provides the application with very fast lookups (since everything is in memory), but has the drawback of being both CPU and memory intensive.

The DOM specification [35] is created and maintained by the W3C, and specifies the interface compliant Application programming interface (API) implementations must provide.

Pull Parsing

Pull parsing is a iterator-based traversal interface. It is similar to SAX, but instead of the parser pushing data to the application via callbacks, the pull parser only traverses further if the application tells it to.

This allows the application to handle the elements in a iterative manner, which is often thought of as an easier programming model.

It shares most of the same benefits and drawbacks as SAX, except for the streaming XML support. Implementing streaming support on top of a pull parser is possible, but would diminish the advantages of having a simpler API that it seeks to provide.

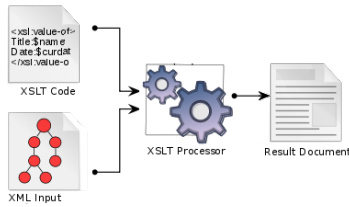

Figure 3.4: XSLT Processing pipeline from Wikipedia article about XSLT [38]

3.2.2 Extensible Stylesheet Language Transformations

XSLT is the process of transforming data from XML document(s) to some other structure or format using a set of rules called a stylesheet. It is not bound to any output format, but the input is always an XML compatible source. With its specification, XML Path Language (XPath) was specified as a way for navigating and doing operations within a XML document. The processing pipeline has one or more XML sources, one or more instances of XSLT code (stylesheet) and a XSLT Processor (see figure 3.4).

The first version of XSLT was developed by the W3C and 1999 [36]. The W3C recommendation was released in it second edition in in 2008 [37].

A very basic usage of XSLT processing of a document is shown in figure 3.5.

Source XML Document:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<persons>
    <person>
        <name>Alice</name>
        <email>alice@example.com</email>
    </person>
</persons>
```

XSLT Stylesheet:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="1.0">
    <xsl:output indent="yes"/>
    <xsl:template match="/persons">
        <names>
            <xsl:apply-templates/>
        </names>
    </xsl:template>
    <xsl:template match="person">
        <name>
            <xsl:value-of select="name" />
        </name>
    </xsl:template>
</xsl:stylesheet>
```

Resulting document:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<names>
   <name>Alice</name>
</names>
```

Figure 3.5: Example XSLT Processing

3.3 Information Retrieval

Information retrieval is the science of searching and retrieving data from a collection of data.

3.3.1 Measuring Correctness

Information retrieval theory provides a set of tools for measuring the performance and correctness of information retrieval systems. These measurements are based upon two main parts, precision and recall.

Precision is measured by calculating the fraction of the entries retrieved that are wanted. The formula for this is:

$$\text{precision} = \frac{|\{\text{relevant entries}\} \bigcap \{\text{retrieved entries}\}|}{|\{\text{retrieved entries}\}|}$$

E.g. If we have a collection of 10 elements and we known that the search we are performed should retrieve 4 relevant elements, but the search returned 5 entries where only 3 entries where relevant then we get:

$$precision = \frac{3}{5} = 0.6 = 60\%$$

Recall is the fraction of the number of entries that are retrieved out of all the wanted entries. The formula for this is:

$$\text{recall} = \frac{|\{\text{relevant entries}\} \bigcap \{\text{retrieved entries}\}|}{|\{\text{relevant entries}\}|}$$

Continuing the example from the precision calculation we get:

$$recall = \frac{3}{4} = 0.75 = 75\%$$

Precision and recall by them selfs are not very useful, so we use the F-measure formula for calculating a score between 0 and 1 which tells us how the retrieval performed (1 is the best, 0 is the worst). The general formula for this is:

$$F_\beta = \frac{(1 + \beta^2) \cdot (\text{precision} \cdot \text{recall})}{((\beta^2 \cdot \text{precision}) + \text{recall})}$$

The general F-measure formula allows us to weight the scoring based upon the value we assign to β. If we set $\beta = 0.5$ we weight precision twice as much as recall. We use this for adapting the score to the type of system we are testing.

Document ID	Content
1	Oranges are better than apples
2	Pears are the best and oranges are awful
3	Bananas rocks and are much better than oranges and pears

Table 3.2: Example Document Collection

Term	Documents	Term	Documents
and	2,3	apples	1
are	1,2,3	awful	2
bananas	3	best	2
better	1,3	much	3
oranges	1,2,3	pears	2,3
rocks	3	than	1,3

Table 3.3: Inverted Index for Example Document Collection

Systems that are sensitive of errors should weight precision much higher than recall.

E.g. Still continuing the same example, and with $\beta = 0.5$ we get the F-measure:

$$F_{0.5} = \frac{(1 + 0.5^2) \cdot (0.6 \cdot 0.75)}{(0.5^2 \cdot 0.6) + 0.75} = 0.625$$

3.3.2 Indexing Data Sources

The purpose of indexing data is to create data structures that allow us to do fast lookup into large collections. The most basic approach to this is to create an inverted index.

Inverted Index

The inverted index is a data structure that allows us to do lookup on a term and get which subset of a document collection that contains this term. E.g. The inverted index of the document collection in table 3.2 is represented in table 3.3. This structure makes it trivial to answer simple boolean queries efficiently.

Vector Space Model

Mapping the documents and their terms into a plane can be very beneficial, as it allows us to do regular trigonometric equations for analyzing the data. The most common way of doing this is representing the documents and terms in a matrix, and is called a Vector Space Model (VSM). The previous document collection may thus modelled as:

$$\begin{array}{cc} \text{ID} & \begin{array}{ccccc} \textit{apples} & \textit{awe ful} & \textit{better} & \dots & \textit{oranges} \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \end{array} & \begin{pmatrix} 1 & 0 & 1 & \dots & 1 \\ 0 & 1 & 0 & \dots & 1 \\ 0 & 0 & 1 & \dots & 1 \end{pmatrix} \end{array}$$

Similarity detection

The inverted index and a VSM gives us the ability to do fast evaluation of relevance of a document using the combination of the term frequency–inverse document frequency (tf-idf) and the *cosine similarity*.

The tf-idf calculates the importance of a term in a given document by scoring terms with a high frequency lower than terms that are less frequently used.

$$\text{tf}(t,d) = \frac{n_{t,d}}{\sum_k n_{k,d}}$$

$$\text{idf}(t) = \log \frac{|D|}{|\{d : t \in d\}|}$$

$$\text{tf} - \text{idf(t, d)} = tf(t,d) \times idf(t)$$

The *cosine similarity* measure calculates the cosine of the angle between two documents in vector space. Smaller angles indicates similar documents. The *cosine similarity* is calculated using the following formula:

$$\text{similarity} = \cos\theta = \frac{A \cdot B}{\|A\|B\|} = \frac{\sum_{i=1}^{n} A_i \times B_i}{\sqrt{\sum_{i=1}^{n}(A_i)^2} \times \sqrt{\sum_{i=1}^{n}(B_i)^2}}$$

Similarity detection in XML documents

The previous techniques are formulated for textual document databases, and does not directly apply for detecting similarities in XML documents.

There are however several approaches to this problem, such as *extended tf-idf* which models the XML structure as a multi-variant regression problem.

The *Path bag model* [17] which bases it's similarity model upon the labels and their paths in the tree, and the Keyword Common Ancestor Matrix (KCAM) [16] model which models the XML fragments as matrices and analyzing which XML fragments are most likely to be a representation of a object based on their structure.

I recommend reading the paper on KCAM [16] for a good introduction to the problem area.

Chapter 4

Research Design

This chapter contains the context and motivation behind the research question, information about how the research was designed and performed, and some of the challenges that were met in the research phase.

4.1 Context and Motivation

The motivation for the research undertaken in this thesis is to enable businesses that are interested in developing free software, to do so on well founded reasons. Free and open software is quite different from the traditional way of doing business in the software industry, and might to many seem counter-intuitive, and maybe even scary. As well as the as the goal for businesses to profit from their projects, there are both legal and technical issues that are somewhat different in the OSS-world.

As there is little research available on the practical aspects of developing and making money from OSS, the context of this thesis is to explore the business, technical and legal considerations concerning the creation of OSS in an reduced and practical manner aimed towards consultancy agencies.

The context as stated results in the research questions described in section 4.1.2 and listed in table 4.1.

ID	Research Question
RQ-1	Is providing OSS a viable business idea for consulting agencies?
RQ-2	How should software be released as OSS?

Table 4.1: Research Questions

4.1.1 Acando and the OpenFEIDE Project

Acando is a consultancy agency within Information technology consulting, IT management and Management Consulting. Acando's annual turnover exceeds EUR 135 million and the Group employs approximately 1,100 professionals in six European countries. In Sweden the company employs approximately 640 professionals organized in six business areas. Their business areas are separated as individual firms in Denmark, Finland, Norway, U.K and Germany.

The stakeholder in this thesis is the Norwegian section of Acando, Acando AS.

Acando AS is interested in OSS as the demand for such systems is increasing, and it's benefits such as freely available technologies and libraries could improve their cost efficiency. They have started working on their first fully open source product, OpenFEIDE.

OpenFEIDE is an open source initiative by Acando to create an open source Brukeradministrativt system (eng: user management system) (BAS) application. It's created on top of a open source Java stack with components such as Glassfish and the Spring framework. The application is targeted at Norwegian municipals and provinces that are starting to adopt FEIDE as a Single-Sign-On (SSO)-service for their schools and organizations. The functionality of BAS-systems is described in detail in section 5.4.3.

FEIDE[1] is an federated identity management system developed by Uninett A/S[2]. Its goal is to simplify the use of computer systems in the educational sector in Norway by providing users with one set of username and password, that works across every system in use in the sector. The solution is based upon Lightweight Directory Access Protocol (LDAP) using shared schemas and a central database for delegation of authentication and authorization requests to the different providers.

There are few alternatives on the market for BAS solutions. The most notable ones are Microsoft Identity and Integration Server combined with Active Directory, and Novell eDirectory. The only identified open source solution is Cerebrum developed primarily by Norges Tekniske og Naturvitenskapelige Universitet (NTNU).

[1] `http://www.feide.no`
[2] `http://www.uninett.no`

4.1.2 The Motivation Behind the Research Questions

The primary motivation behind the research questions are to get an understanding of open source software in regards to both business and development.

Open Source as a Viable Business Idea

The question targets the primary issue for companies that is considering to create OSS and releasing it to the world; How does one profit from giving the software away for free? And is the potential income enough to justify the investment required for creating a product that users want to use? To gain insight in whether it's a viable business idea for consultancy agencies to provide OSS, we need to look at the product itself, and the market of that particular product.

In most cases, there are no real distinction, from a customers point of view, between a proprietary product and a product based on OSS, so to gain this insight we need to perform market analysis', such as Political, Economic, Social and Technological factors (PEST) and Strengths, Weaknesses, Opportunities and Threats (SWOT) analysis, on the product and compare against other products in the same field. What does the competition offer their customers, can the product in hand offer more, what kind of services can we provide that will compel customers to choose our product. And what kind of services are suitable for an open source product?

Are there incentives for the potential customer pool for choosing OSS products over proprietary products? What are the values and risks for the different business models for the provider?

With business comes legal matters, and one of the primary concerns for a lot of companies are how open source licensing works and affects there ability to use the software at hand. We need to look at some of the most prominent licenses and see how they compare to each other and how they might affect the future of the software that is licensed under them.

Practical Open Source Development

The research questions states "How should software be released as OSS?", and revolves around the practical sides of the development phases of open source software. How is the project managed, how is development done and so on. To gain insight into known good ways of releasing and managing open source

software, we need to study other successful projects, and try to extract some knowledge about the processes and techniques these projects are using.

4.2 Applied Research Methods in this Study

There is little research available with regards to the practicality of developing OSS with he end goal of generating value based on the support and services that surround this type of development. These decision processes are often done by people already engaged in the OSS communities, and are based on the collective intelligence of these communities, and the practices they deploy. This is especially true in the technical parts of developing successful open source projects.

This state of affairs reflects on the chosen research methods in this paper.

4.2.1 Literature Search

- **Web-search**: Search engines such as Google Scholar was extensively used for procuring former research. Google Scholar is a tool created especially for this purpose, and contains most of the articles and papers published by ACM, IEEE, Springer and so on. It also contains results from Googles normal index that has been identified as papers.

- **Project and company web-sites** were used to gain information about specific projects and companies.

- **Bibliography tails** were key tools for finding relevant papers, as the quality and amount of papers found was higher than by any of the other methods.

4.2.2 Choice of Research Methods

The primary research method used in this thesis is the case study research strategy. Studies about Open source software in the context of business is a mixed field and need a broad approach for data gathering. Both qualitative and quantitative evidence is of interest.

There are three main cases in the studies presented, as well as data gathered from observing the greater open source community and attitudes.

Product	Company	Services offered
Varnish	Varnish Software AS	Support, Training, Consultancy
Gitorious	Gitorious AS	Hosting (SaaS), Consulting, Training

Table 4.2: Companies with a primary focus on Open Source software

4.2.3 Studied Companies

The subjects in the case study was selected based upon their similarity to the context of this thesis, companies providing single purpose end-user products. The companies selected are listed in table 4.2.

Varnish Software

Varnish Software is the company behind Varnish Cache, a web-cache application used by, amongst others, Facebook and VG. The company is based in Oslo and Stockholm and is a spin-off project from Redpill Linpro.

Varnish Cache is licensed under a BSD-license, and is available at their community site `http://www.varnish-cache.org`. This site includes access to their source code repository, bug tracker and mailing list archives.

Their business plan is based upon selling support and proprietary add-ons for the open source software.

The study is available in Appendix A.

Gitorious AS

Gitorious AS develops and maintains a product called Gitorious, which is a collaboration tool for the git version control system. The software is licensed under the AGPL-license and is available at their community site `http://gitorious.org/`, and is used by Nokia, OpenSUSE and QT.

Their commercial offerings include SaaS, consulting and support.

The study is available in Appendix B.

Other Companies

As there are already extensive case studies done on the large international open source based companies, such as Red Hat, I've not not repeated these. See for instance "The adoption of open source software in business models: a Red Hat and IBM case study" by Munga et al. [25], which is an extensive piece

Product	Purpose	Status
Linux Kernel	Operating system core	Success
Symbian	Mobile device operating system	Failed
Django	Web development framework	Success
Varnish Cache	Caching HTTP reverse proxy	Success
Gitorious	GIT Hosting solution	Succcess

Table 4.3: Open source projects

of work studying the business models with regard to open source at Red Hat and IBM.

4.2.4 Studied Projects

In addition to these companies a few open source projects where studied, both successful and failed ones. It is important to view both the successful and the failed ones for trying to determine common characteristics on why some projects are successful, while others fail. These projects are listed in table 4.3.

Linux Kernel

Probably the most famous open source project to date. It powers the GNU/Linux operating system, but also other projects such as the Android mobile operating system, and super computers.

The study is available in Appendix F.

Symbian

Symbian is an operating system for mobile devices created by Nokia and was the primary operating system for mobile devices for a long time. It was open sourced as an attempt to improve it's usage as other platforms such as iOS, Android and Windows Mobile started to take a larger share of the market.

The study is available in Appendix G.

Django Web framework

Django is a python based web framework created initially as a tool for newspapers to rapidly develop features for the web presence.

It is the most widely used and popular web framework in the python community.

The study is available in Appendix E.

4.3 Case Study Design

Case studies are a way of extracting information without influencing either subjects or the environment [4], and are well suited for answering "how" and "why" questions [1][4]. Because of these attributes we can, by designing the studies they way we want them, do all our research externally.

4.3.1 Company Studies

In the case studies targeting companies we want to know about how the go about working in an open source environment. To figure out how they do that, the following questions were selected and must be answered in each study:

(CQ1) What is the business model?
We need to understand how the business is set up in regards to offering services and/or products surrounding the open source software. We do this by identifying the means described in section 2.4, and analyze the model using the "Business Model Analysis Framework" suggested in "An Analysis of the Value that Open Source Contributes to Business Models" by Munga and Fogwill [24].

(CQ2) How do they interact with the project community?
What sort of role does the company have with regards to the project they construct they business upon?

(CQ3) Is the business profitable?
Control the viability of the business in terms of monetary profits.

(CQ4) How large is the company?
What is the size of the operation that works with the given open source project(s)?

4.3.2 Project Studies

The art of practical open source development can be extracted by looking at successful open source projects and see what they have in common, and see if there are any patterns emerging.

The following questions must be answered by each study:

(PQ1) How is the source code managed and controlled?
How does the project make its source code available, and which restrictions do they enforce with regards to publishing code?

(PQ2) What is the release process of the project?

Is there a formal process for releases? Who decides when something is ready for releasing?

(PQ3) How does the community communicate?

Does the community have specific tools and methods on which they choose to communicate through? If so, are they moderated or in any way managed by some entity?

(PQ4) In what way is documentation managed?

How is the project documentation distributed and managed? Does the project have dedicated web pages for documentation, if any documentation at all?

(PQ5) How does the project manage user feedback?

How are user feedback and support requests managed? Is there any single point of contact for these requests? If so, how are the response times for the questions?

(PQ6) How does the project manage user contributions?

How does the project accept bug fixes and new feature patches? How does it decide which ones to accept, and if they are good enough?

(PQ7) How does the project manage bug reports?

Is there a single point for reporting bugs? How are these managed after they have been submitted? Are there any restrictions on whom might triage these reports?

(PQ8) Which software license is the project licensed under?

What sort of licensing scheme is the project under? Why did they choose that license?

(PQ9) How easy is the software to deploy and use?

Does the project provide easy ways for it to be deployed and tested by potential user?

4.3.3 Data Collection and Challenges

Data collection was performed by inspecting each company and project web site, by performing web searches for further information, and in some cases watching recording of talks some of the creators have given about the history of his or her project. Business models are often kept secret, so the data collected is extracted and interpreted by the offering the companies publish.

Challenges in Subject Selection

One of the primary challenges of collecting data was to find unsuccessful projects that were over a certain size. There is a wide selection of projects that fail, but finding information about projects attempted by companies that failed is proven to be difficult. These projects seems to be removed or left dead, and by this being difficult to find. The primary concern is the validity of the findings as to the limited data set.

Finding suitable subjects for studying was also a big challenge because of the breadth of projects, and thus the results might not be applicable for whatever project that is being evaluation as an open source endeavor.

Depth of Gained Understanding

The research questions in this thesis are very broad in the sense that the touch almost every aspect of open source software and business in open environments. As a result of this each case study is equally broad and perhaps shallow. One could argue that each of the questions in the case studies could be suitable for further research on its own, and how the particular part has an

effects on the project or business decisions. The goal of this thesis is not, however, to get a complete understand of all the effects of different choices that are made by projects, but to get an overview of the complete picture.

Chapter 5

Contributions

This chapter presents the results of the studies and the implementation that makes up the contributions from this thesis. The contributions directed at the research questions are presented first, then the implementation.

5.1 Viable Open Source for Consultancy Agencies

Creating a positive return on investment in authoring OSS can be challenging, but not impossible. It requires a combination of technical solutions and business solutions, where both are connected to each other in various ways. There are multiple suitable business models for OSS projects, some are described on page 2.4 For consultancy companies the value lies in the expertise the company acquires by developing it, and thus enabling themselves to provide expert consulting on the software. Choosing a business model depends on the software that is being created, and what the customers are willing to pay for. They key point is to focus on services, not the product.

5.1.1 Legal Considerations and Licensing

Open source licenses are often an issue for companies as there is little knowledge about them and how they work. Most companies today prefers to buy proprietary solutions as a means to reduce the risks inherent in untested licenses. Ten years ago this might have been a problem, but now open source licenses are widespread and used by most of the major software companies in the world. This thesis explained the most common licenses in section 2.5. Most of the licenses are proved in court by now, so the choice of licensing depends now only on the way you want the software to be used, and to what degree you want to protect your software. For large software the far most common license is the Apache License (the 2.0 version). It's a well written legal document covering licensing issues many large companies run into, such

as patent rights. If you want a permissive license, which allows you to use the software in proprietary products, the Apache license is a safe bet. If you on the other hand want a restrictive license the GPLv2 is the most used and tested license.

Most companies are inclined to choose permissive licenses as they offer them greater flexibility. That being said, both of the companies (see appendix B and appendix A) studied in this thesis user restrictive licenses, and are still able to provide a wide array of services to their customers, and create a profit.

5.1.2 Estimating the Cost of Development

For estimating whether or not a project is going to pay off we first have to estimate the costs for producing the software. Cost estimation in software is hard problem due to the complexities of software development, but there are models that provide an approximate ball-park figure such as the COCOMO II model [3]. The COCOMO II model is based on the size of projects in terms of the number of source code lines. The basic formula is

$$Effort = a \times KSLOC^{b}$$

where the `a` coefficient has an initial value of 2.94 and `b` has an initial value of 1.0997, `effort` is measured in person-months. The COCOMO II model allows for several adjustment parameters for increasing the accuracy of the estimation based on the properties of the project being estimated, but for simplicity's sake only the basic model will be used in this thesis. More information about the COCOMO II model is available in their manual [3]. Once we have a baseline for how much effort a project is estimated to require, we can create a model for representing the flow of value between the consultancy agency, and the customers. A simplified model for calculating the result of a project can be represented as

$$result = (n * p) - (d * c)$$

where `n` = estimated consultancy hours needed in the market, `p` = price per hour of consultancy, `d` = estimated development time and `c` = cost pr hour of in-house development.

The model can be used to estimate how many hours of consultancy that must be sold for the project to break even. If we assume an exchange rate of 1NOK equals 6USD, the in-house cost of an employee is 600NOK, and the consultancy price per hour is 950NOK the equation for breaking even is

$$n = (2.94 * (KSLOC^{1.0977}) * ((600/6) * 7.5 * 5 * 4))/(950/6)$$

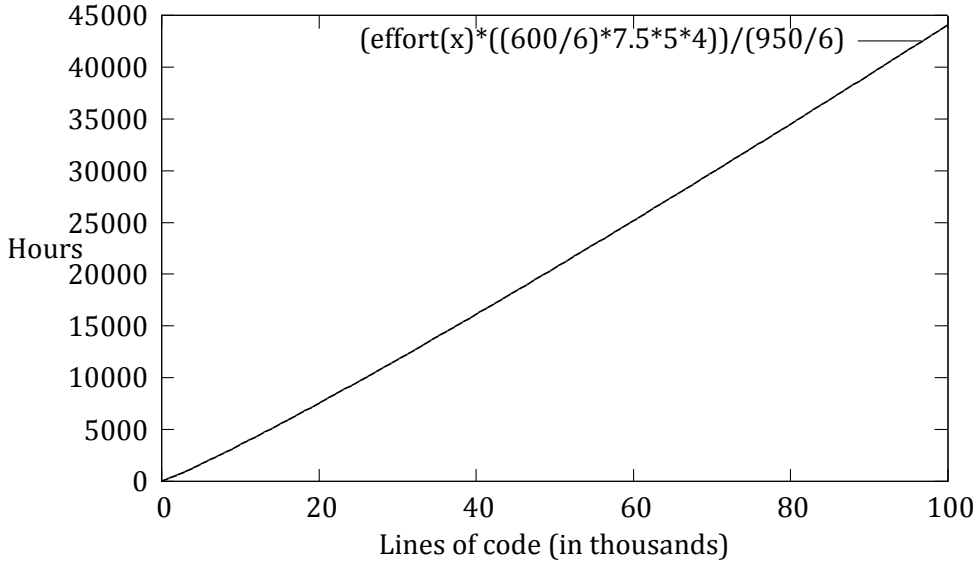

Figure 5.1: Needed sales of consultancy hours versus lines of code

The growth is shown in figure 5.1 tells us that even small projects require a fair number of consultancy hours. A 20,000 lines project clocks in at about 5000 hours, that just over three years of work[1]. Herein lies the difficulty of not selling the software, but only consultancy for it, and a large factor in why many companies, as mentioned in the previous parts of this thesis, often have several services connected to their products.

5.1.3 Business Models

Product Classification

Once the cost estimation is completed, one must see if the product profit potential is large enough to support the initial development costs and the long-term maintenance costs. Sandeep Kirshnamurthy designed a two dimensional classification scheme in his paper "An Analysis of Open Source Business Models" [18], with the dimensions being "customer applicability" and "relative product importance". This is represented in figure 5.2, and shows four different classifications. "Customer applicability" denotes the potential amount

[1] Norwegian full-time hour count per year is 1650 hours.

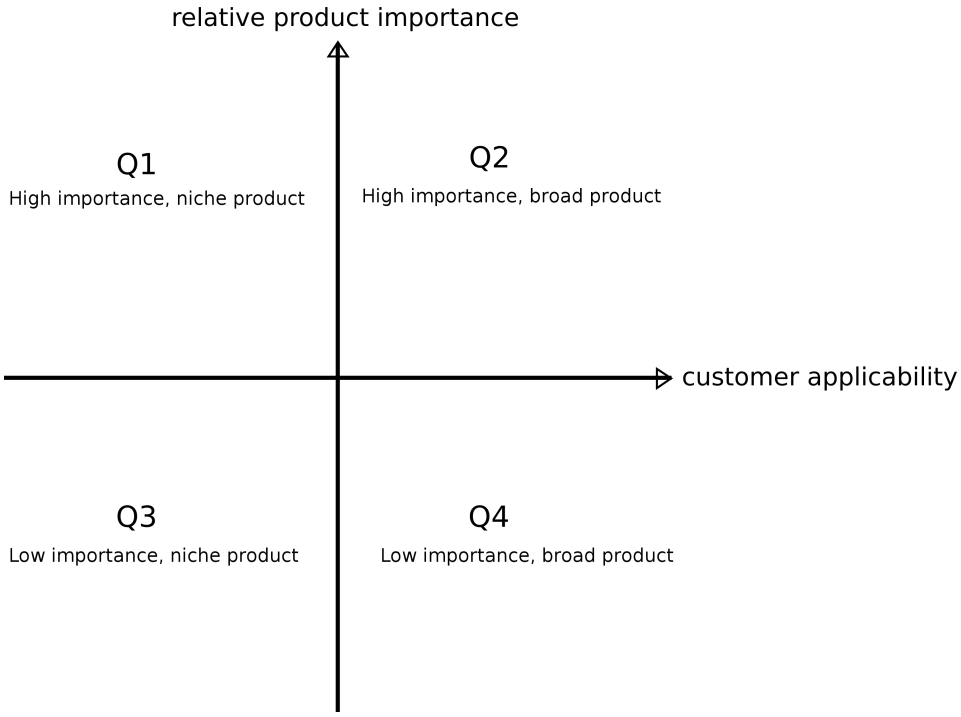

Figure 5.2: Product classification

of deployments. Products such as cross-platform web-browsers have a large customer applicability since it can easily be installed to the computers of a large customer base. "Relative product importance" is a measurement of how important a piece of product is for the potential customer base. Products such as an operating system has a high product importance, while a new screen saver has a low level of importance.

Q1 **High importance, niche product**
Products that are of high importance, but for a small market. One example of product in this area is data processing.
Profit potential: Medium

Q2 **High importance, broad product**
Products that are of high importance with a large potential market. Products such as web browsers, operating systems, office suites go into this category.
Profit potential: High

Q3 **Low importance, nice product**
Low importance nice products have the lowest potential market as the number of potential users is low.
Profit potential: Low

Q4 **Low importance, broad product** Products in this category may be called mainstream utilities. Typical examples of this is web-based tools such as online TODO-lists.
Profit potential: Medium

Similar tools such as PEST and SWOT analysis can improve the information at hand about the current market and the profit potential of the product, and which business models might be most suited for the product and the market.

Business Model Selection

Choosing the right business model is a difficult but necessary step, and must be approached on a project to project basis. As described in section 2.4, there are many different models available and suitable for open source projects.

As with any attempt to sell something, one must determine what the customer is willing to to pay you for. This varies from project type to project type. If the project at hand targets enterprise customers, support might be a very good start for an product offering, as this is a much sought after product for enterprises using open source software and is proven to be profitable e.g. Redhat Enterprise Linux.

The product classification scheme described above is a useful tool when evaluating the different business models. Products classified as "Low importance, broad product" (large customer base, low complexity), might not be suitable for consultancy offerings, but maybe proprietary addons providing extra functionality is a better fit. "High importance, niche product" might be a very good fit for consultancy offerings and adaptation offerings.

Each model must be reasoned against the product and the potential customers to work, and because of this most companies that do business solely on open source, uses several different business models, and often intertwine them to make each of them more attractive for the customers. One example of this is Varnish Software which bundles their proprietary tools with the support contracts.

5.2 Practical Open Source Development

This section explains a framework for developing OSS solutions from scratch in a well tested way. Each subsection explains one phase of the product life cycle.

5.2.1 Project Start Up

The initial development of an OSS project is not very different from a regular in-house proprietary project. There are however some considerations that must be made when the project is to be open sourced and released to the public.

Source code management (SCM) usage

When developing the initial versions it is important to remember that, in the ideal OSS environment, the complete source code history will be made available to the public at some point. This means that no sensitive information should be added to the version control, as these may be extracted and exploited by an external entity after the release. This includes data sets containing person information such as social security numbers, usernames and passwords for various systems such as the database system. This point is considered best-practice for in-house projects as well, but the importance increases dramatically when the information may become public.

The choices one makes with regards to when and how to release the source code may have a great impact on the contribution rate. If released early, and the project manages to attract some contributors, then those contributors are often bound to become regular faces, as they gain a greater sense of ownership of the project. The way it is released is also important. Very few developers today prefer to get "code dumps", that means a compressed archive of all the code. Most of todays developers prefer to fire up their SCM tools and clone the project. This has two effects: They can see all the history, and are likely to understand more of the project. The second effect is that any contribution they make, will most likely be easier to merge into the project, as the mechanical parts of joining the code bases is already done by the tool.

All of the studied projects have publicly available source code repositories, and are thriving.

Documentation

During the initial stages of any software project there will be made a lot of presumptions and choices for how to attack the problems at hand. Keep in mind that at the point of releasing the software external contributors will need this information and not just the internal development team. As with the previous point considering version control usage, this is also best practice as even the internal team members may be replaced and these new members will also need the complete understanding of why the project core is developed as it is. An additional effect is that the quality of the software tends to increase when the developers are forced to document it. This is due to the fact that this forces the developer to clarify the purpose of the implementation, which may trigger the developer to find a cleaner way of solving the problem to ease the documentation.

The most dramatic effect that good documentation can have, can be seen if we look at the history of the Django project. The Django project emerged amongst many other python projects as the Ruby-on-Rails project was taking off. It was not technical superiority, and it did not have any more features than a couple of other projects, but it had great documentation. The reason for this is that the original authors of Django were journalists, and hated that they had to search long and hard for documentation of what they were attempting to do. So when they created the Django project, they made sure that its documentation was top-notch. This made it the primary web framework for Python developers, and it still is to this date.

Deployment - ease of use

The goal of any project should be to have users. The more users a project have, the more contributors get interested in the project. As well as the markets surrounding the project increases, including the need for consult by experts. Therefor the software should be developed with deployment in mind from the very start. The most effective way for a project to enable users to use their software is to package the software for the users and for the different relevant platforms. Creating packages for the major Linux distributions is a good start. Debian[2] and Fedora[3] should be the primary targets here, as packages for these distributions also work on Ubuntu, RedHat, CentOS and a number of differ-

[2] `http://debian.org`

[3] `http://fedoraproject.org`

ent distributions. For MacOSX[4] one can package the software as an installer or create distributions for projects like Homebrew and Fink. In the case of Java based web projects, a pre-compiled WAR-file from the project homepage should be enough.

The purpose of packaging is ease of installation and maintenance. Using a versioning scheme such as Semantic Versioning[5] and providing detailed change logs on new releases is a big step towards this.

5.2.2 Developing in the Open

Having a publicly available website where externals entities may follow the development closely makes the development process transparent. This increases the chances of external contributors since they have the chance to follow and comment on progress. If they have patches for features they want to have included in the distribution they can easily keep it up to date with the source code. This helps the core developers to merge the provided patches as they could require patches to apply cleanly to the current version of the source code. Keeping the revision history public also helps bisecting bugs to the commit which introduced it, thus creating a fast-path for a new regression test and patch. The good part is that this entire process can then be done by external contributors.

One should also not neglect the slight pressure put upon developers when their changes are made public; this may improve not only code quality, but the quality of the metadata for the changes, such as the commit message. Any seasoned developer have seen large commits with commit messages like "fixes a bunch of stuff", which is not helpful for others who track the development.

However, it can be argued that the most important feature of this type of transparent development is the fast feedback loop from the community. The more people who are able to beta-test the software by running the latest version, the quicker bugs can be found and reported. This may reduce the wave of bugs that often are reported after a large release which can take a long time to fix for the developers.

There are several online services and open source software projects available for the purpose of sharing source code repositories. Most of these services and projects have features for managing the project, such as bug tracking and wiki hosting, and the online services often have additional features

[4] `http://apple.com`
[5] `http://semver.org`

available for paying customers. Some of these external providers are listed in table 5.1. In a study performed by Black Duck Software [27] from January to May 2011, GitHub was the most popular hosting service in terms of number of commits.

Version control software	Provider	Features
Bazaar	Launchpad	Code hosting, Bug tracking, Project management, Code reviews, Translations ++
Git	GitHub	Code hosting, Wiki, Bug tracking, Code review
Mercurial	Bitbucket	Code hosting, Private repositories, Bug tracking
Subversion, Mercurial	Google Code	Code hosting, Wiki, Bug tracking
Subversion, Mercurial	Microsoft CodePlex	Code hosting, Wiki, Bug tracking
Subversion, Mercurial, Git	SourceForge	Code hosting, Bug tracking, Wiki

Table 5.1: List of remote public code hosting providers

5.2.3 Managing Users and Contributors

Managing an open source community is no small matter, but the basics are pretty easy to get going for any new project. It's a matter of providing communication channels for both users and contributors, as well as for the core developers. Creating an active user base not only helps promote your project by word-of-mouth, but also helps you reduce costs by off-loading the development team by having the community do the basic support tasks.

User Feedback

One critical part of managing an open source project is the last part of the famous quote

> Release early, release often. And listen to your customers.

by Eric S. Raymond in the book "The Cathedral and The Bazaar". Providing some level of support for the users in the open are clear sign to any new users that the project still is alive. This is a critical part, not only for building a strong community, but the mere fact that there is publicly available signs of activity is crucial in the open source world, as a lot of open source projects are inactive.

This can be accomplished in a number of ways. The far most popular way is to have open mailing lists and IRC channels for communicating with the developers and other fellow users. This as all other forms of public exposure may create some difficulties with regard to moderation and spam control, but the cases where this poses a threat to the project are rare. The primary concern should be to have ways of communicating ideas and problems.

Managing Contributions

There are different approaches to handling external contributions, and the need for control varies with the size of the community of the given project. Most projects have a set of core developers that decide what goes into the "official" releases. These developers acts as gatekeepers for external contributions. Normally contributions are being made by an external developers creating a patch for the project then sending the given patch to the projects mailing list, whereas the core developers and the community can do code review and decide whether or not the patch should be applied to the official project. In recent years, with the rise of distributed version control, it's more common to create a "branch" for the contribution and sending the developers a link to the given changes for code review and merging. This allows the full version control history of the given patch to be merged into the official version control, thus creating a way of bisecting bugs in the patch at a later point.

5.2.4 Bug Tracking and Issue Management

Bug tracking and issue management is crucial to any project, but it may have a profound impact on open source projects whether this information is publicly available or not. There are two main benefits for open source projects for having publicly available bug and issue management systems as opposed to an closed system:

- Users have a single point of reporting bugs.
- New contributors have a source of tasks to work on.

The first point is important as it reduces the frustration of the user since they know where to report bugs and follow the progress for having them fixed. It also allows the developers to easily ask the reporting user for more information if the original report did not include enough information to reproduce the bug. The second point is useful for creating an environment where volunteers have a single place to look for tasks if they want to contribute to the project.

5.3 Relationship Between Business and Technical Aspects

One of the unfamiliar concepts of open source is the unique relationship between the technical aspects and the business aspects of the projects. Due to the openness of the open source software, the technical choices the project has made are open for anyone to see. This can cause both good and bad PR in the open source community. The technical drivers are part of the whole community aspect of the project, and thus also the business aspect.

Figure 5.3 illustrates the relationship between different choices and drivers, and how they might affect the project as a whole in a positive way. There are, as with any choice, trade-offs for each choice, such as control or the costs of actually managing a large community, but if the goal is to have more customers, and more feedback, positive effects it is what one wants.

Dahlander and Magnuson described two types of relationships between companies involved in open source software and the community surround the software in their paper "Relationships between open source software companies and communities: Observations from Nordic firms" [10], *symbiotic* and *parasitic*.

Parasitic relationships between a company and a open source community is often described as "leeching", since the company attempts to make a profit from the project with minimal involvement and contributions. The symbiotic relationship the company gains trust and position in the community by their efforts and values.

In the case of creating new open source projects and gather help from the open source community it is crucial to establish the latter relationship type. The amount and type of effort needed to become trusted enough as a company for volunteers to invest time and effort to the project varies from project to project. As in personal relationships the trust between parties relies on their trustworthiness, which is earned over time by being truthful and showing integrity.

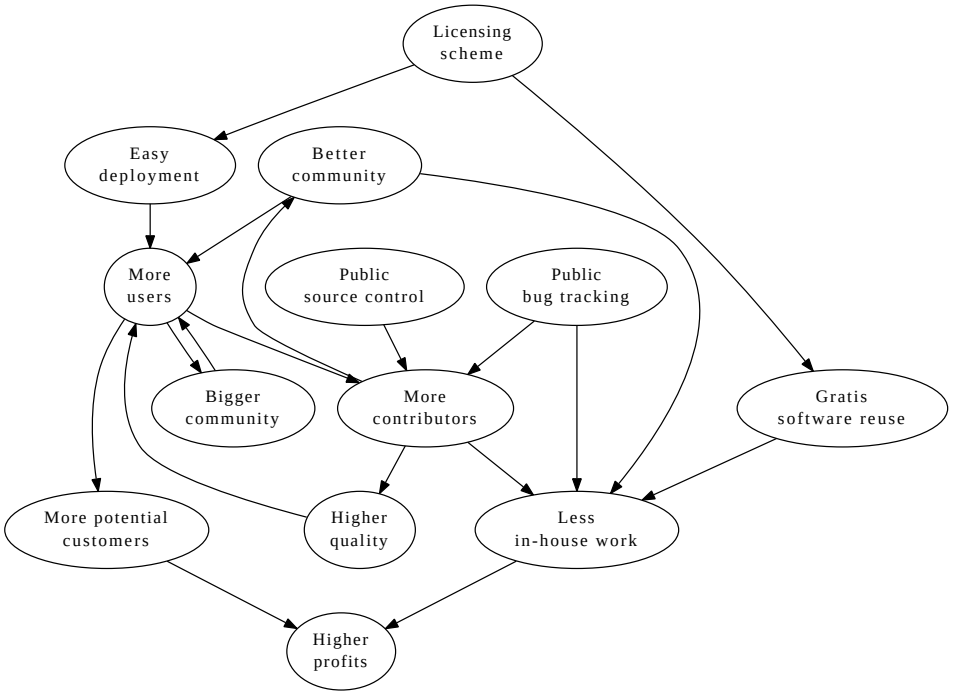

Figure 5.3: Relationship between technical aspect and community aspect with profits

5.4 Implementation of XSLTGenerator

The purpose of this chapter is to describe the premises for the software, as well as the software design and architecture. The section is based upon IEEE documentation standards IEEE-1016 for software designs, and IEEE-830 for software requirements.

5.4.1 Problem Description

OpenFEIDE requires data sources for retrieving the user base it handles. This importation procedure is based on the import of XML-files from internal SAS components. As these internal systems varies in type and version, the XML-files they are able to export varies. The current version of OpenFEIDE has a one-to-one mapping of an external XML source, and a local XSL-Transformation for converting the XML data to a format understood by the importation module of OpenFEIDE.

The problem with this approach is that the user have to write the XSLT for their data sources. While this is normally not a problem for a developer, it is a difficult process for potential customers that do not have developers at hand, and must do this on their own. The result of this is often that the customer gives up, and uses a competing product.

The goal of the implementation part of this thesis is to create a prototype module for helping non-developers do the task of creating appropriate XSLT templates for their data sources.

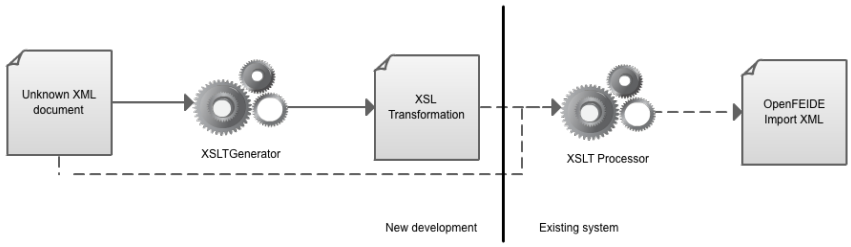

Figure 5.4: Visualization of implementation problem description

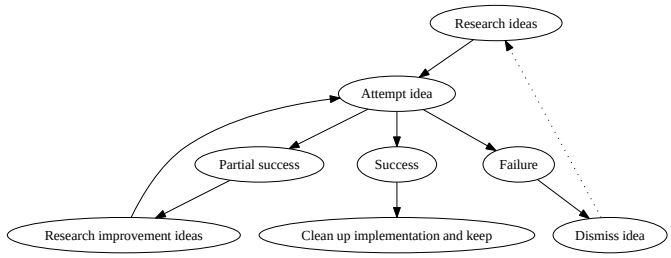

Figure 5.5: Development process

5.4.2 Development Methodology

As a one-man effort, the implementation did not follow any strict methodology, but the closest approximation is the Kanban methodology. The initial phases was done in spikes for testing different approaches to the problem. During these spikes methods like traditional IR-methods (tf-idf indexing and similar techniques), and DCAM was abandoned due to the lack of successful analysis and retrieval using these methods, to a custom scoring system developed especially for this problem area.

Parts of the application was done in a test-driven manner, mostly for shaping the internal APIs to something that would be comfortable to work with as a developer.

These spikes attempts and the dismissal of them can be viewed as an iterative development process as shown in figure 5.5.

5.4.3 Context of Implementation

Role of BAS Applications

An BAS application's role is to provide a digital system for management of user credentials and information in an organization. This central point for user management is an requirement for establishing a connection with the FEIDE system.

In the usual case the BAS imports data from existing systems such as SAS and making sure the data is organized and valid according to the specifications set by FEIDE[6], but there is no restrictions on how the BAS gains its data points.

[6] `http://www.feide.no/sites/feide.no/files/documents/temahefte_feidekrav.pdf` (Norwegian)

The information in the BAS is required to be exported to the external LDAP-catalog at least once every 24 hours. The external LDAP-catalog is required to follow the norEdu* schema[7] to be compatible with the FEIDE-system.

OpenFEIDE Technology Stack

As with any software product it's important to keep track of which external components are used, and which are linked with the product to ensure license compliance.
OpenFEIDEs external components are shown in table 5.2, and shows that all but one linked component is licensed under the Apache License, with the other one is under the LGPLv3. All these licenses are compatible and since JasperReports is released under the LGPL, OpenFEIDE is free to be released under any licensing scheme.

Component	Linked	License
Google Web Toolkit	True	Apache License, Version 2.0
Spring Web Services	True	Apache License, Version 2.0
Quartz Enterprise Job Scheduler	True	Apache License, Version 2.0
JasperReports	True	LGPL, Version 3
Log4J	True	Apache License, Version 2.0
Spring Security	True	Apache License, Version 2.0
Spring LDAP	True	Apache License, Version 2.0
Glassfish	False	CDDL, Version 1.1

Table 5.2: Identified external components in OpenFEIDE

OpenFEIDE Architecture

The OpenFEIDE architecture has tree critical internal components, the web server hosting the administration interface and managing the import from SAS systems, a SQL-server for logging and other audit data, and an internal LDAP server for storing the FEIDE compatible data. The internal LDAP-server replicates it's data to an publicly available LDAP-server which is the entry point for the central FEIDE LDAP to communicate with. A rough representation is shown in figure 5.6.

[7] `http://www.feide.no/sites/feide.no/files/documents/norEdu_spec.pdf`

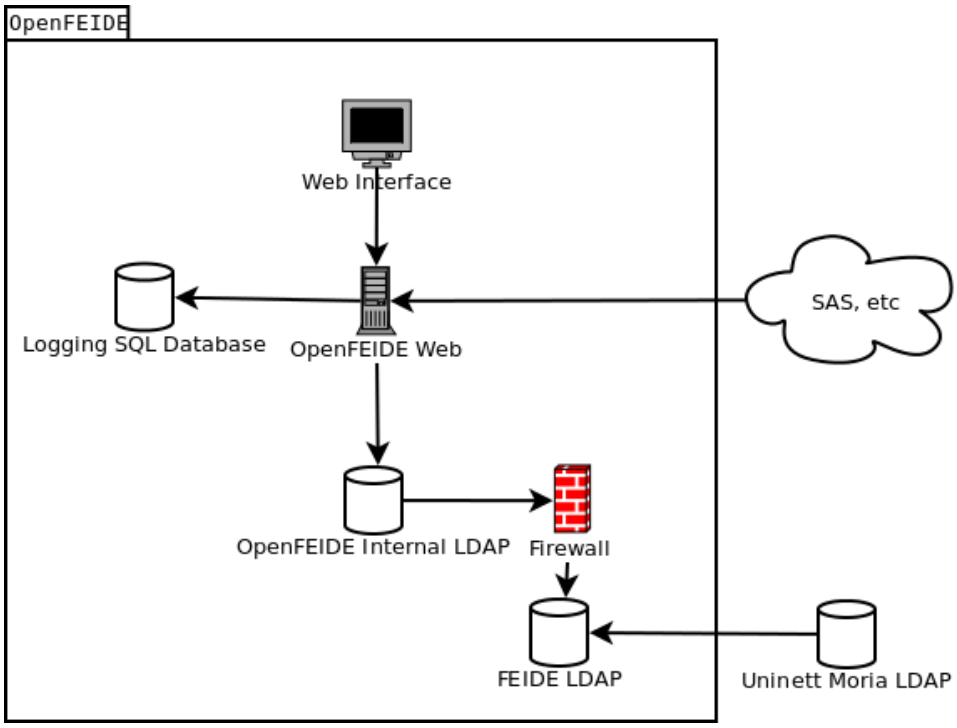

Figure 5.6: OpenFEIDE Architecture Schema

The system is designed to never expose any of the core systems to the public internet, reducing the risk of cascading intrusions from the outside. If the external LDAP is tampered with, the data will get fixed in the next update from the internal systems. The external LDAP has no way of communicating with the internal systems, so it is impossible for an attacker to gain entry to the internal servers from this server.

The internal data source providers (such as SAS), communicate with the webserver through web services provided by the Spring WS library, or by storing the data for importation on an FTP-server available to OpenFEIDE.

The web services are run on the Glassfish Application Server software, and are responsible for logging all activity to audit logs, generating reports and exporting data to the LDAP-server.

5.4.4 Problem Analysis

The main issue stated in the problem description can be summarized into the following:

> How can we enable the average user to successfully import data into OpenFEIDE from an unknown source?

Even though OpenFEIDE is specialist software, the training effort should focus on the day-to-day use of the software, not the adding of data sources. And considering the well known mantra for creating user friendly software: "Don't make me think" (from the book with the same name by Steve Krug [19]), the process should, as much as possible, be completed without any decision making or interaction from the user.

This however creates new problems. The format of the XML files OpenFEIDE are given from the data sources is unknown, this makes the process of of creating correct and deterministic importation procedures hard.

The solution for this problem is to improve the level of knowledge the system has about the source. Currently all we know is that the input is represented as XML, and that the data, most likely, has information that is of usefulness to OpenFEIDE, such as information about persons, groups and so on. We can use the information about what kind of data we are interested in, and use this information to search the data in the given file for the appropriate content.

Searching for data fields in this manner is however error-prone, so there is a need for some control on how and when fields are identified as appropriate content. Two simple ways of achieving this control is to employ either one of, of multiples of the following ways:

1. Identification confidence level

2. Manual interaction from user

The first way is a parameter introduced to the automatic process of detecting data, and the other is asking the the user for help to identify a certain piece of data. The second could thus be used as a control mechanism for the detection phase, with or without, the confidence level contract needed for successful identification of data, or serve the purpose of identifying data entirely on its own.

5.4.5 Analysis of Current Import Procedure

The existing import procedure of OpenFEIDE is done manually by adding a new data source in the web interface. This source must be located on an FTP-server. The transformation into the OpenFEIDE import format is done by a hard coded XSLT-Transformation.

The existing activity chain for adding a new data source into OpenFEIDE is illustrated in figure 5.7.

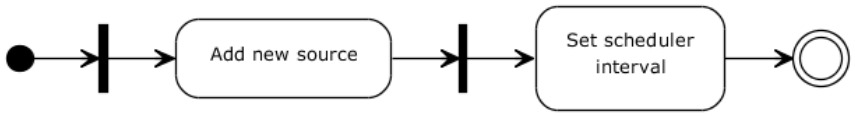

Figure 5.7: Activity chain for adding new source to OpenFEIDE before XSLT-Generator

5.4.6 Analysis Results

The software under development is a module for the OpenFEIDE importer adding support for creating XSLT transformations for the users data sources without requiring the user to know XSLT. The software will be developed as an pluggable module, as illustrated by figure 5.9, which the import may call upon the instantiation of a new data source if the user chooses to use it, as opposed to creating the XSLT transformation by hand.

Implementation Placement

XSLTGenerator is enabled it would added in the initial stages of the import creating the activity chain shown in figure 5.8.

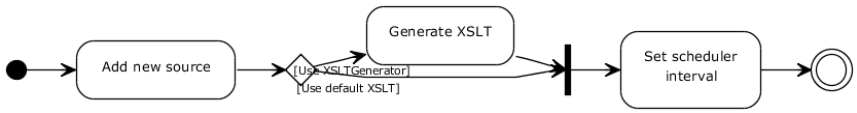

Figure 5.8: Activity chain for adding new source to OpenFEIDE after XSLTGenerator

Implementation Usage

As OpenFEIDE is a fairly young project, and development is still on-going, and without regards to the development of this thesis, the path of least resistance is to create XSLTGenerator as a separate project that is easily added to OpenFEIDE at a later stage. From a developer perspective the software should be easily added to the product by enabling it as a dependency in the Maven `pom.XML` file, and calling the documented API for initiating and using it.

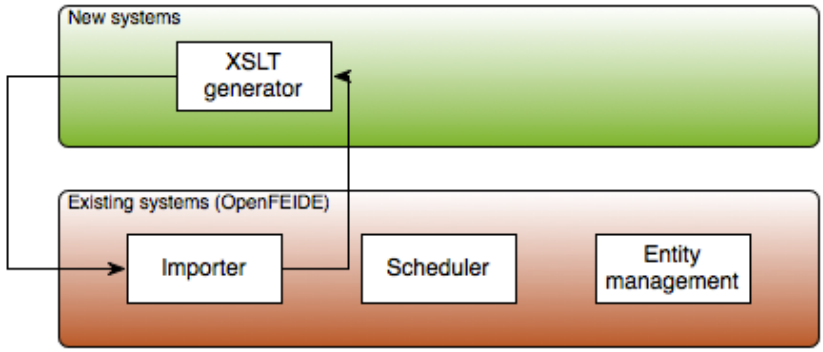

Figure 5.9: Contextual placement of the new system

Soft System Requirements

This results in the following soft system requirements:

- The software should be able to create XSLT transformation for XML sources without requiring the user to know XSLT.
- The software should be easily integrated by developers by adding it as an Maven dependency.
- The software should expose a well documented API for integration into software as OpenFEIDE.

5.4.7 Requirements

The purpose of this section is to define the requirements for the software being developed, and defining the guidelines for the development process.

Functional View

The generation of the transformations can be done in two separate ways as shown in figure 5.10; one it fully automatic generation by analyzing a sample XML output from the new data source. The other is a manual process where the user may add labels to the different XML-fragments that appear in the XML output from the data source as shown in figure 5.11. The software will use this data to generate an XLST-document for the specific source.

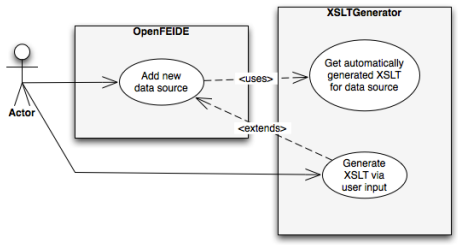

Figure 5.10: Main user interaction use case

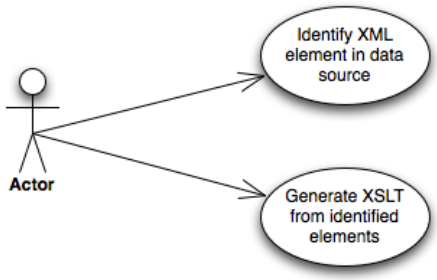

Figure 5.11: Use case for manual generation

The two different approaches requires different architectural needs and implementations.

Combined Functional Requirements

Solving the issues raised in the functional view of the two subsystems requires the functional requirements listed in table 5.3 to be fulfilled.

ID	Role	Description	Priority
F-1	Developer using the API	I want the software should accept any valid URI supported by the Java 1.6 platform for retrieving the XML sample.	High
F-2	Developer using the API	I want the software to be able to analyze any valid XML document.	High
F-3	Developer using the API	I want the software should produce a valid XSLT transformation as output.	Medium
F-4	End user	I want the software to expose an user interface for manually mapping XML fragments to their meaning.	Medium
F-5	Internal	Elements must be considered distinct based upon the element name and the elements attributes.	Medium

Table 5.3: Functional requirements

Non-functional View

In addition to the functional requirements several non-functional requirements are present due to the problem description of this theses, as well as the existing technological platform of OpenFEIDE.

The software should strive to fulfill the following non-functional requirements:

Constraints

The software should adhere to the following constraints:

- The software must run on the Java 1.6 platform
- The software must be able to inter operate with the OpenFEIDE application.
- The performance of the system should be able to display any page in less than 1 (one) second per request.
- Core parts of the application must be accompanied by a suitable test suite.

ID	Genre	Description
NF-1	Performance	No page should take more than 2 seconds to load.
NF-2	Performance	The XML analyzer should, as OpenFEIDE, focus on performance without regards to memory usage.
NF-3	Security	The software should not include any security mechanism.
NF-4	Extensibility and flexibility	The XML analyzer should be easily modifiable and modular.
NF-5	Interoperability	The software should be easily integrated into any GWT-project.
NF-6	Interoperability	The software should be easily themed by adding custom CSS.
NF-7	Usability	The software should be usable by non-technical persons.

Table 5.4: Non-functional requirements

- The software has no requirements for memory usage, but should work in low-memory environments with lowered performance.
- The software must run on all major browsers.
- The software must be able to integrated into an deployed GWT application.
- The module is aimed at non-technical users and must be designed to be used by this user group.
- The parsing infrastructure for the automatic generation perspective must be easily modifiable as this is probably the most brittle part of the module.

5.4.8 Goals and Guidelines

The software development process should adhere to the following guidelines:

- Follow the KISS principle ("Keep it simple stupid!") as much as possible.

- Follow the standard Java naming conventions.
- The software should be as fast for the user as possible without impairing code readability.
- The software should allow to be themed into the look-and-feel of another application.

5.4.9 Architecture

Logical view

The chosen architecture is based upon the standard GWT application architecture, which resulted in the main Java packages listen in table 5.5. The different parts of the system is designed to be easily be expandable by following the singe responsibility patterns for business logic, and well defined connection points for the business logic classes. The fact that this is an OSS-component also influenced the architectural design for easily extracting parts of the application for usage in other products. One example of this is the parsing utilities where the logic is separated into three layers. Each layer knows only about the layer beneath it, thus creating an easily manageable dependency graph as seen in the package dependency diagram illustrated in figure 5.12.

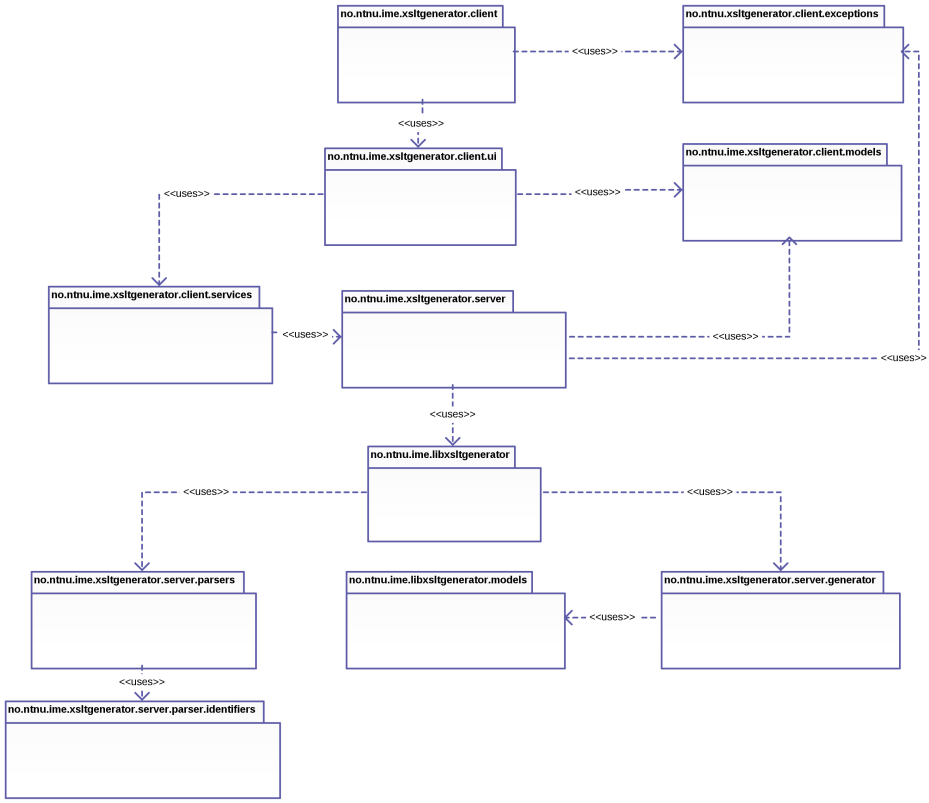

Figure 5.12: Packages and their dependencies

Package Layout

GWT enforces a layered software layout as mentioned in section 5.4.9. This layout resembles the much used three layered MVC-structure. The packages that contains the keywork `client` are designed to be translated into JavaScript, and run in the browser. All data that is sent between the client and the server must be implemented as such a class.

The source code is split into several Java packages. The package layout is an best effort attempt for keeping each part of the project separate and to allow clean reusable internal APIs. The package structure is listen in table 5.5.

Package name	Description
no.ntnu.ime.xsltgenerator.client	Base package for the code that will be run on the client.
no.ntnu.ime.xsltgenerator.client.ui	UI-classes for specific parts of the application.
no.ntnu.ime.xsltgenerator.client.exceptions	Exception either raised from the server side or the client side that reaches the user.
no.ntnu.ime.xsltgenerator.client.models	Shared data structures for serialization.
no.ntnu.ime.xsltgenerator.server	Base package for the server side code.
no.ntnu.ime.xsltgenerator.server.utils	Common utilities.
no.ntnu.ime.libxsltgenerator.generator	Functionality for generating XSLT.
no.ntnu.ime.libxsltgenerator.analyzer	Analyzing and statistics for element identifying.
no.ntnu.ime.libxsltgenerator.parsers	XML-element parser implementations.

Table 5.5: Java package overview for XSLTGenerator

5.4.10 Functionality Types

Automatic generation

The automatic generation is done by analyzing the contents of an XML-document exported from any given data-source. The main problem with this approach is the fact that the software has no knowledge about the structure of the XML-document it's analyzing, thus making the process error prone. For combating this issue, we use the knowledge of FEIDE to help the process. This is done by creating a set of rules to match the data that is of interest of FEIDE. The four primary data types in FEIDE are:

1. Persons
2. Entitlements
3. Groups
4. Organizations
5. Memberships (persons that are part of a group, or an organization).

Each of these data types have a given set of attributes for describing the entity they are representing. This information can be used to create a set of rules for detecting each data entity. A person has to have a name, a social security number, an address etc. If enough data is available the software can interpret the common structures for location each data entity. The success criteria for this type of analysis is highly dependent on a well-formed and logically structured XML-document. An example of a structured and logical XML-document is available in figure 5.13, and an unstructured example is available in figure 5.14.

Manual generation

The manual generation is done with the help of user input. The idea is that the user assigns labels to the different XML-fragments in the document. The XML document can easily be displayed as a tree-structure providing the user with an easy logical view of the document. This is important since the user have to make informed and correct decision on what an XML-fragments is in the context of importing.

There are some pitfalls creating this type of representation that must be addressed, such as multiple elements of the same type in the same XML-fragment.

```
<document>
  <persons>
   <person>
     <name>John Doe</name>
     <username>johndoe</username>
     <email>john.doe@example.com</email>
    </person>
  </persons>
</document>
```

Figure 5.13: Well formed XML example

```
<document>
  <persons>
   <person name="John Doe" email="john.doe@example.com">
   <person-usrname-index>1</person-usrname-index>
    </person>
  </persons>
  <usernames>
    <username>johndoe</username>
  </usernames>
</document>
```

Figure 5.14: Non-logical XML structure

In XSLTGenerator elements are separated by not only their element name and position in the tree, but also the attributes of the given element. This enables the user to attach different labels on

```
<contactinfo type="email" />
```

and

```
<contactinfo type="mobile"/>
```

The XSLT generator must take this information into account when creating the XSLT templates for ensuring a high information quality.

5.4.11 XML Analyzer System Design

The XML parsing system is designed to be very modular and easily modifiable as required by the non-functional requirement NF-4. This is done by using Java best practices with the generics facility introduced in Java 1.5 and

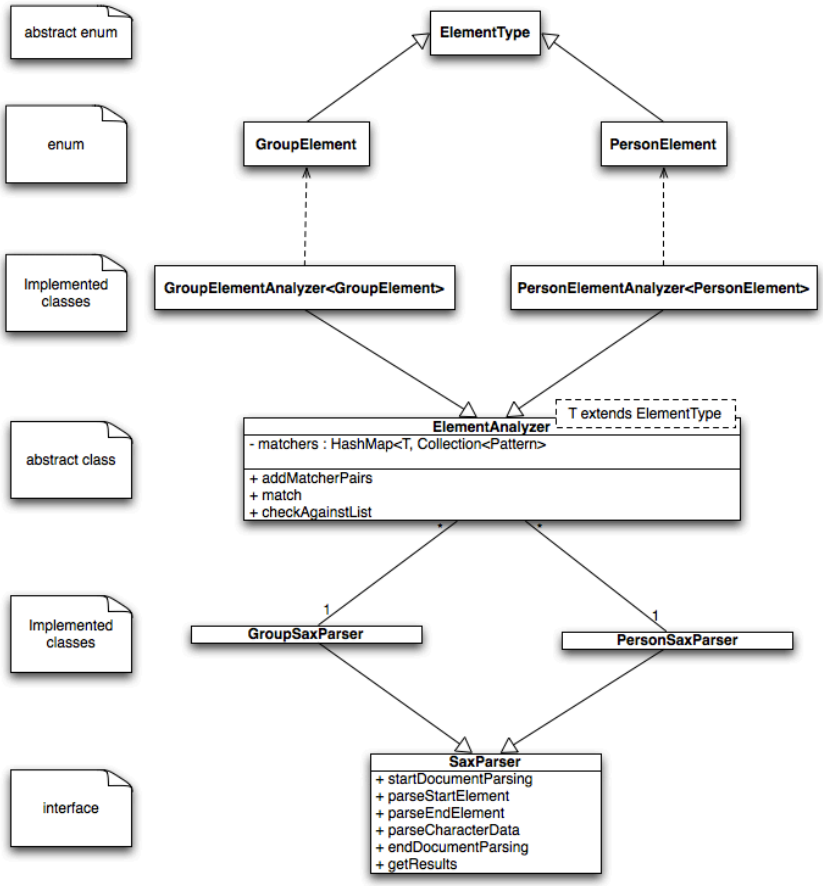

Figure 5.15: XML Analyzer design

standardized interfaces. The analyzer design is split into different layers as shown in figure 5.15. The `SaxParser`-interface is the starting point for detecting any singular type of XML-fragment in the XML-document, such as the `PersonSaxParser` and `GroupSaxParser` shown in figure 5.15. Each `SaxParser`-instance may deploy any type of analysis it wants, but the ones that were created as a part of this thesis are based on analyzing each XML-element that occurs in the given document by its name, content and placement in the document tree. The abstract class `ElementAnalyzer` is the result of the common needs that occurred during the development of the software. Each implementation of `ElementAnalyzer` is bound to an implemented enum with `ElementType` as the super class by type inference. The implemented enum is responsible for clarifying what kind of field the analyzer thinks the current element is. An example of such an enum can be represented is shown in figure 5.16 and it partial parser implementation in figure 5.17.

```
1  public enum PersonElementTypes implements ElementTypes {
2    FIRST_NAME("Fornavn"),
3    LAST_NAME("Etternavn"),
4    MIDDLE_NAME("Mellomnavn"),
5    GIVEN_NAME("Fornavn"),
6    SSN("Personnummer"),
7    GENERIC_NAME_NODE("Navn"),
8    GENDER("Kjonn"),
9    BIRTH_NODE("Fodtselsdato"),
10   ZIP_CODE("Postnummer"),
11   GENERIC_ADDRESS_NODE("Adresse"),
12   COUNTRY("Land"),
13   CITY("By"),
14   STREET("Gate"),
15   GENERIC_CONTACT_NODE("Kontaktpunkt");
16
17   public final String description;
18
19   PersonElementTypes(String description){
20     this.description = description;
21   }
22
23   public static PersonElementTypes descriptionToEnumValue(String
         description){
24     for (PersonElementTypes e : PersonElementTypes.values()){
25       if (e.description.equals(description)){
26         return e;
27       }
28     }
29     return null;
30   }
31   public String toString(){
32     return this.description;
33   }
34 }
```

Figure 5.16: ElementType enum implementation

```
1
2  public class PersonSAXParser extends
       SaxParserDefaults<PersonElementTypes> implements SAXParser {
3
4      ArrayList<String> stack;
5      HashMap<String, ArrayList<PersonElementTypes>> count;
6      PersonElementAnalyzer elementAnalyzer;
7      private PersonElementTypes currentType = null;
8
9      @Override
10     public void startDocumentParsing() {}
11
12     @Override
13     public void parseCharacterData(String data) {
14        if (data.trim().isEmpty()) { return; }
15        if (PersonNodeContentAnalyzer.isPersonId(data)){
16           currentType = PersonElementTypes.SSN;
17        }
18     }
19
20     @Override
21     public void parseStartElement( String uri, String localName,
22                                    String qName,
23                                    Attributes attributes) {
24             stack.add(localName);
25             currentType = elementAnalyzer.match(localName);
26     }
27
28     @Override
29     public void parseEndElement(String uri,
30                                 String localName,
31                                 String qName) {
32             stack.remove(localName);
33             count.get(localName).add(currentType);
34             currentType = null;
35     }
36 }
```

Figure 5.17: ElementType Parser Implementation

Path	Detected type	Aggregated score
/	null	0
/root	null	0
/root/persons	null	3
/root/persons/person/name	PersonType.NAME	1
/root/persons/person/username	PersonType.USERNAME	1
/root/persons/person/email	PersonType.EMAIL	1

Table 5.6: XML Fragment analyzer scoreboard

In the provided analyzers these types are then mapped to their location in the XML-document. In the example XML-document in figure 5.13 there are the follow distinct XPath-addresses, their match value, and score. The score is calculated by summarizing all the child nodes that are matched as a type, in this example a person-type. The resulting scoreboard for the example XML-document is shown in table 5.6. As seen in this table the highest scoring distinct XPath-address for the document is `/document/persons`. This path is not identified as any specific type, thus the analyzer predicts that this is the common root path for the fragments containing information about the element, in this example the `PersonType`. This information is needed for creating the appropriate XSLT-templates.

Performance considerations

Since the analyzer architecture and design relies on each enabled parser (XML-element to ElementType detection) checking every entity in the XML document performance might become an issue. Earlier experimental spikes of the software stored the XML document and traversed it as a in-memory DOM-structure. This made the analyzer extremely slow. The finalized architecture only traverses the XML-document once as the different parsers are plugged into a global SAX-parser. This reduced the analysis time of the software from 10 seconds to 3 seconds. Since 3 seconds still is a long time, the architecture was refactored to have static compile-time evaluated regular expression matchers inside each parser. By doing this the regular expression are expanded into the decision trees once, and not per element check. After this refactoring the time to analyze the same data set was reduced to 300ms using the same hardware.

The down side of this is that the new architecture for the parser is more rigid and does not allow run time modifications. Aside from this it had an positive effect on the architecture as well, as the code flow was greatly simplified. The current flow of operations can be represented as this pseudo code:

```
1 for(Entity entity : XMLDocument.getEntities()){
2   for (Parser parser : parsers){
3       parser.handle(entity);
4   }
5 }
6 for (Parser parser : parsers){
7   parser.analyzeCollectedData();
8 }
```

5.4.12 System Sequences

The system sequences are shown in figure 5.18 and represents the control flow during a single run of the implementation.

The `User` represents the actual human user, `Client` represents the Javascript application running in the user's web browser, and the `Server` represents the Java application running on the server.

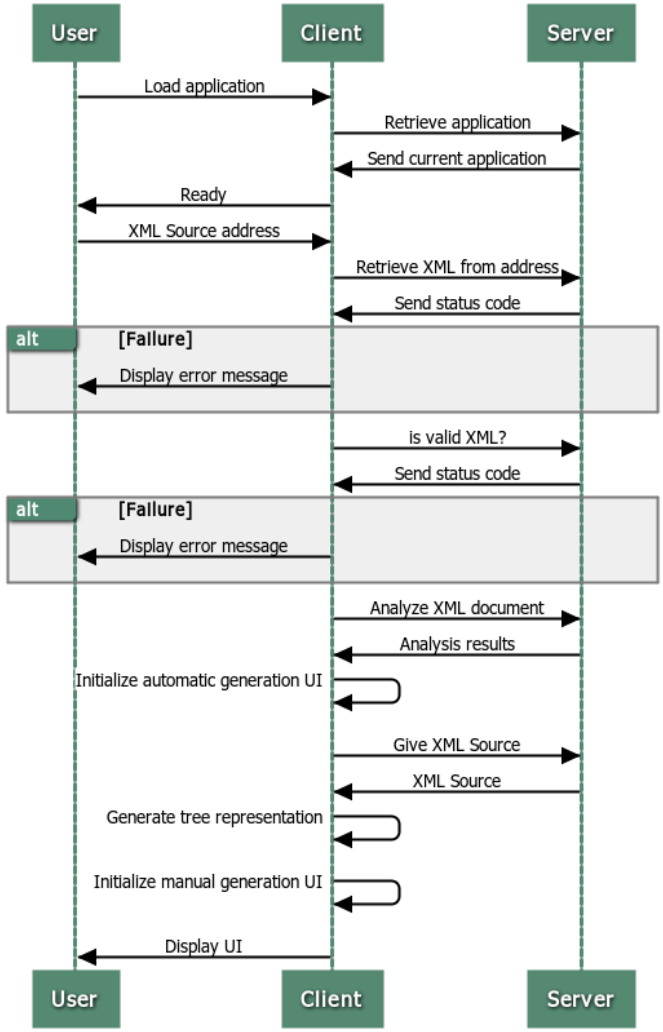

Figure 5.18: Client server communication during user interaction

5.4.13 Technology Selection

As OpenFEIDE is built using Java 1.6 with the Google Web Toolkit, Maven and Spring, the creation of the XSLT-generating module was built using the same technologies, except for Spring which was not needed. The XML-processing was done using the included XML tool set in the JDK, so no new dependencies were created.

Google Web Toolkit

GWT is a Java development framework for creating Javascript based web applications. It allows the developer to use Java for both client side and server side code, thus allowing a shared code base. The client side Javascript is then compiled into optimized Javascript and run on any A-grade browser, in addition to Internet Explorer 6.

Communication to the server is done using RPC and a built-in serializer. Since the server side code shares the same data structures, the data sent over the network to the server can be reduced to a minimum as the information about the structure of the data already is available.

Maven

Maven is a tool for building and managing dependencies of Java projects. It has become the de-facto build-tool in the Java communities as it is seen as having a less complex structure and syntax than it's predecessor Apache Ant.

It works by defining your project in an XML-file in the root at your project, and using either the maven command-line tool or a plugin for your Integrated Development Environment (IDE) of choice.

The purpose of the tool is to have a repeatable set of steps for getting a working piece of software from the code base.

Plain Java

It was decided early in the project that a bare minimum of libraries should be used, as a means to reduce the complexity of the project, and for reducing the number of sources of uncertainty. On of the positive effects of this is the fact that there are often problems in larger Java projects that the different libraries depend on different versions of third-party libraries. This causes problems as the Java class loader is non-deterministic if multiple version of a class is available. There are solution to this, such as OSGi, but these are highly complex and

should not be used unless absolutely necessary. By not requiring any libraries that are not already in use, the risk of this happening is minimized.

Chapter 6

Evaluation & Discussion

In this chapter the results presented in chapter 5 and 6 will be used to answer the research questions. The chapter also discusses the results as whole.

6.1 Answering the Research Questions

The context of this thesis is a consultancy agency considering investing in OSS and thus have some questions regarding its profitability and how OSS authoring and releasing is done.

These questions are split into two broad research questions which the next sections attempt to answer.

6.1.1 Viable Open Source for Consultancy Agencies

The research question

Is authoring OSS a viable business idea for consultancy agencies?

The core of the research question is whether or not consultancy companies can create open source software and see a profitable return-on-investment by offering services for the created product.

There are several steps needed for estimating whether or not an OSS project will yield a profit. These steps are very similar to traditional COTS development, and the software being open source affects mostly the choice of business models that are suitable.

Profit estimation

A basic model for estimating the profits of a project is presented in section 5.1.2, and consists of a method for estimating the cost of software development (COCOMO II), and some tools for analyzing the market (SWOT and PEST) along

with a classification scheme. For a practical introduction to SWOT I recommend the slides from Sanjay S. Mehtas' course on marketing strategy [22].

Business models

The other part of estimating if the project is going to be successful is choosing the appropriate business model(s) for the project. Section 5.1.3 contains some guidelines for choosing the business model according to the classification scheme presented in section 5.1.3.

The far most popular business model is to offer support contracts. They are often split into several levels of available support, and are often bundled with proprietary products for adding value to the service. Redhat provides "verified" upgrades as a part of their support contracts, while Varnish Software adds additional tools.

All of the studied companies have diversified their business model by offering multiple types of services. Gitorious AS (study in Appendix B)is a good example of this. They offer both the product as Software as a service (SaaS) and as managed service on the customers hardware. This is a typical example of attempting to meet the needs of as many potential customers as possible, which seems to be typical for companies dealing with open source.

Licensing

One of the big hurdles and seemingly difficult aspects of OSS is the issue of licensing, both how to select the correct license and how the licenses affect the choices one have for leveraging other pieces of OSS. As there are currently 69 licenses[1] that are considered open source licenses by the OSI, it might seem like a impenetrable jungle, but in reality only a handful of these licenses are widespread. The licenses ranges from permissive to restrictive as illustrated in figure 2.7 and described in section 2.5, where the restrictive have strict rules on how the code is used and might be cumbersome if one wants to use the code in relation with proprietary products. Restrictive licenses do have the advantage of forcing other version of it to be published with full source code, so that one can always leverage the work of others. Permissive licenses however do not impose such restrictions, but that also might have the effect of the contributions of others not being available for you.

When choosing a license its important, at least as a company, to choose well known and tested licenses. Most companies choose permissive licenses

[1] `http://www.opensource.org/licenses/alphabetical`

such as the Apache license. The Apache license is especially suitable for products that are being made available in the US and other countries with strong patent laws. Opscode, the creators of the configuration tool Chef[2], wrote an excellent blog post on why they chose the Apache license over other licenses. It's available at their blog[3] (last checked 2012-01-01), and is a recommended read.

Summary

Creating profit from Open source software is a complex business as all the parts of the project are connected to the value chain, and each part has its own set of auxiliary effects.

The chances of an consultancy agency, measured purely in the direct monetary aspect, succeeding with the creation of a brand new OSS project are quite small, because of the critical mass of users needed to sustain development purely on consultancy is quite high. However, if one focuses on other aspects of OSS projects, it might make sense for this kind of company to invest resources in its creation. These aspects might include PR, research, training, customer relations, recruiting and so on. This might only be suitable for large agencies, since the main issue of these aspects are that they are hard to measure in any meaningful way, which results in a less controllable environment.

The key factors for creating a successful consultancy business around an Open Source project is a sufficient customer mass, which makes creating niche open source products hard. If we look at projects such as Varnish Cache, which is an application for front end caching of web servers, the market is not limited to a small set of customers, but potentially a world wide customer base. This again sparked the need for Linpro A/S to create a daughter company called Varnish Software to only manage that project alone. Varnish Software was studied as a part of this thesis (see Appendix A), and shows that providing a simple array of services they manage to operate a company, and doing so successfully. Herein lies one of the common characteristic for companies working with open source - they almost always offer multiple services. This requires the company to be very agile and aware of the market, but if one are successful in doing so, the company will have more legs to stand on when the market changes. Some of the most used business models are presented in section 2.4, and gives a brief introduction to each of them.

[2] `http://www.opscode.com/chef/`

[3] `http://www.opscode.com/blog/2009/08/11/why-we-chose-the-apache-license/`

In the context of OpenFEIDE it depends on whether the investment needed for the project to become a true competitor to COTS in that particular market matches the possible income from the available customer mass for that particular type of product. It is possible, but for it to work the OpenFEIDE-project must become more open and focused on building a community, thus making the product build a customer base on its own, without the company having to contribute all of the investment needed, as illustrated in figure 5.3. Building this type of community is a demanding tasks however, and too requires resources to become successful. A set of specific action points for kick-starting this process is written in Appendix H.

6.1.2 Practical Open Source Development

The research question

How should software be released as open source?

Successful OSS projects are collaborative in nature, so the gist of the answer to this question is to develop and release free software in an open fashion for encouraging this collaborative nature.

This however is easier said than done, especially if the developers are new to the open source scene. To answer this question in a meaningful matter, we have to look at how other successful projects are structured, and how they approach development and releases to establish a community and a collaborative force. By studying several successful projects a set of "best practices" was created and presented in section 5.2. These practices may be seen as a framework for creating open source software in a community friendly manner.

Following this framework will however only provide the project with a set of tools that have been successfully deployed by other projects, and are by no means a bulletproof set of rules that guarantee success.

The points addressed in section 5.2 can be classified into two groups; infrastructure and policies (see table 6.1).

Infrastructure

The points classified as infrastructure are tools that should be setup and managed in such a matter that is useful for the overall goal of having an open development model. Source code management for instance should be publicly available, and not behind internal firewalls. Using external services such as

Classification	Points
Policies	User feedback
	Managing contributions
	Deployment
	Publicly available source code
	Issue management
Infrastructure	Source code management
	Documentation
	Bug tracking

Table 6.1: Classification of practices

GitHub is probably a good idea. It provides the necessary tools for getting any open source project started: source code hosting, wiki for documentation and a simple bug tracking tool. Custom tools can be setup when the need arises.

Policies

An open and distributed development model is greatly improved if there are some official policies set in place by the main developers. These may span from technical decisions to community efforts. One example is the Django project (see Appendix E) which enforces that every contributed new feature must include not only code documentation, but documentation for the end user. Policies might also be put in place on how to handle bug tracking and the triage process of them. The point of these policies are to streamline the project into an effective project.

Summary

If the goal of a project is to be successful in the sense of an active community, then it is important to make the project open and welcoming. Creating a strong community has many effects on the project, some of which are presented in section 5.3 and figure 5.3. Building these communities are not a small feat, but the chances of success increases with the amount of effort and openness that is put into the project. Following the advices given in chapter 5 is a good starting point for any new OSS project. One of the hard parts for teams that are unfamiliar with open development and innovation may be the fact that the code is no longer tailored for one customer, and must be made to fit a broader

audience, even if that results in a extra software being created for the customer that needs the software.

6.2 Summarized Recommendations

To summarize the recommendations the following list indicated the most important aspects.

Develop in the open

Make your code repositories publicly available as soon as possible. This is a part of developing in an open fashion which is more inviting for contributors than development done behind closed doors. Development and discussions done in private are often seen as a sign of alienation for potential contributors, and diminishes their sense of ownership and worth for the project.

Make your project easy to find for both users and developers

This is essential if the project is to attract both users and developers; if no one can find your project, then nobody will use it. It might be a good idea to both have a project page on its own domain, as well as to publish the code on well known source code hosting services.

Provide ways for both users and developers to communicate

The chances of getting feedback from users, and to assist aspiring contributors, it is very useful if there are well defined contact points where they may reach the existing community and developers. Tools like IRC and mailing lists are popular choices in the open source scene.

Make the software easy to use and deploy

Providing an easy way of installing the software is really good idea to increase the market share of the software. Not only does it make it easier for people to try it, but if done correctly it may provide you with solid upgrade paths. This helps you to provide fixes for the software quickly, and thus increasing the quality of the product. Packaging the software for for all major platforms is therefor a recommended strategy.

Provide a single point for reporting bugs and other issues

As with the communication point this is about having well defined places for reporting and searching for bugs. This is not only for users reporting

bugs, but also for developers that want to contribute to find something they can work on.

6.2.1 Validity and Limitations

The validity of the guidelines can be questioned, as there is no true answer on how to organize open source software projects, and any procedure must be adapted to fit the project at hand. However it is in the opinion of the author that the guidelines provide a solid base for any open source project to grow, and have a firm presence in the open source software market.

6.3 Implementing XSLT Generator for OpenFEIDE

XSLTGenerator is the result of implementing a system that allows a non-technical person to create XSLT-files for converting XML-sources into a format supported by OpenFEIDE. The result was a two separate ways of solving the problem; the first is a fully automated system that uses analysis and statistics to determine the correct way of translating the document. The other is a manual process involving the user.

The key question can be constructed as follow: "How can we create an XSLT-transformation for an unknown input source to fit a know format without technical interaction from a human?" and can be simplified to the process shown in figure 6.1.

This is a difficult problem to solve in a general way since the parser must account for any type of XML-structure. In reality there are only a handful of formats that are candidates for being imported into OpenFEIDE, such as documents following the ABCEnterprise schema, which is the only freely available schema that has public documentation, and thus the only set of example data that XLSTGenerator was built after.

Given a properly structured document the built analyzer should be able to extract most of the context needed for importing data, but there is one area

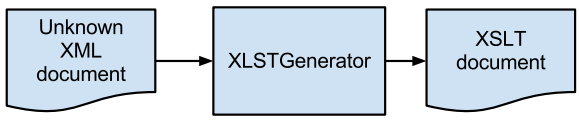

Figure 6.1: Purpose of XSLTGenerator

that is not solved. XLSTGenerator has no way of knowing the meaning of a membership to a group. Given the group name of "1.klasse" and derivatives of this name, it is impossible to know the meaning of that name in terms of access privileges. One could hard-code support for every example that one may find in the wild, and hope that the solution will converge against something that will work 100% of the time in the future, but this is unlikely to be a good solution as the consequences of given person the wrong access might be grave (giving students access to the grading software for instance).

Neither implementation will solve the problem 100%, due to the loose specifications and large varieties of XML-structures that might enter the system.

6.3.1 XML Analyzer results

During development only one document from a production system was available for testing. This XML-document was an export from a SAS-system structured according to the ABCEnterprise XML-schema. This example data is not included as an appendix as it contains confidential information about students in data set.

Detection results

The analyzers included in this thesis achieves the results listed in table 6.2 tested against the sample document. The results are calculated using the techniques described in section 3.3.1 and with $\beta = 0.5$. The good news from these results are that we have no false positives, however we do lack a lot on the case of recall on the results. As a result it is not yet production ready, especially when it comes to the case of entitlements. We have not identified any functional generic solution for detecting and understanding entitlements from arbitrary XML documents. This is a show stopper for using the process without any human interaction.

6.3.2 User Interface

XSLTGenerator includes a basic user interface built upon the provided API. Running the sample application the first page the user sees it the entry point shown in figure 6.2. This page accepts any URL support by Java 1.6 as required by functional requirement F-1. Once the user presses the "Parse XML Source"

Type	Wanted	Detected	Precision	Recall	F-measure
Organizations	8	2	1.00	0.25	0.62
Org. Units	14	0	0.00	0.00	0.00
Persons	12	12	1.00	1.00	1.00
Groups	2	2	1.00	1.00	1.00
Group membership	3	2	1.00	0.66	0.90
Entitlements	2	0	0.00	0.00	0.00
Average			0.66	0.48	0.58

Table 6.2: XML Fragment analyzer test results

button a progress bar appears while the software analyzes the XML and created the needed UI-elements.

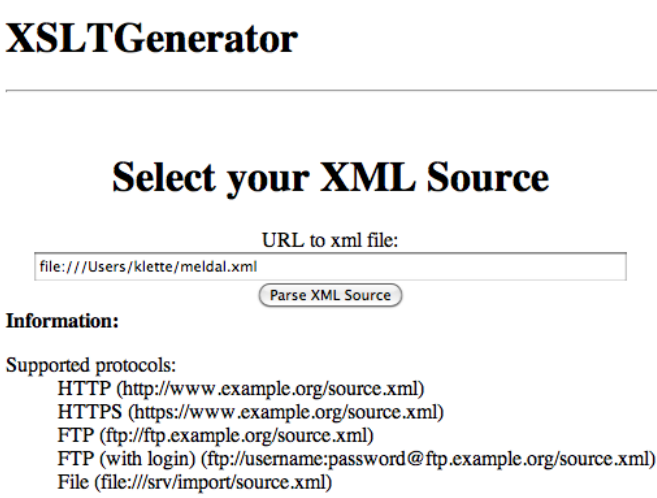

Figure 6.2: Screenshot of sample entry point for XSLTGenerator

Once the this process is done the user is shown a page with a menu bar at the top that allows the user to switch between the automatic generation mode and the manual mode. The former shows example data output from all the types and fields it was able to discover in the XML document as shown in

figure 6.3.

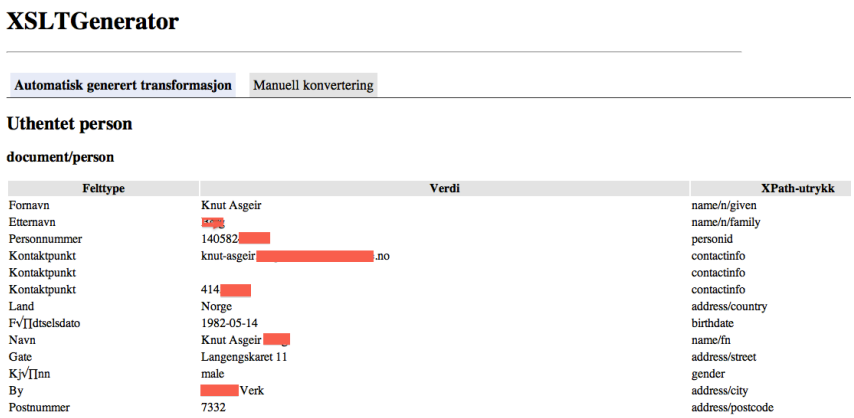

Figure 6.3: Screenshot of sample entry point for XSLTGenerator

At the bottom of the page the generated XSLT is shown (see figure 6.4. This information is not useful for the target user, but at this sample application is created as an example for the developer that wants to use the module it includes some technical data.

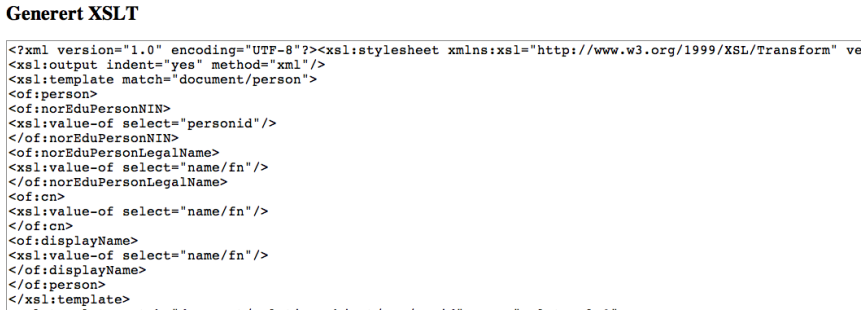

Figure 6.4: Screenshot of sample entry point for XSLTGenerator

If the user clicks the "Manual generation"-tab, he is taken to the manual mode example application page. This page displays all the distinct XML-fragments in a tree-structure. The user then goes through all the elements he can interpret and adds the appropriate label from the menu at the right. When an element has been assigned a label it's marked as green in the tree. Once the application has enough data to generate any XSLT, the result is displayed at the bottom of the page. This page is shown in figure 6.5.

The manual generation tab allows the user to select not only the direct meaning of the current XML-element, but also add side-effects. The use-case for these side-effects are group memberships that modify the standard FEIDE affiliation groups. One example of this is a group membership, or relation element that has an `type`-attribute called "has-teacher". This would imply that the person connected to the group in the relation is a teacher, and thus should in addition to the regular groups be included in the `FACULTY`-group defined by FEIDE.

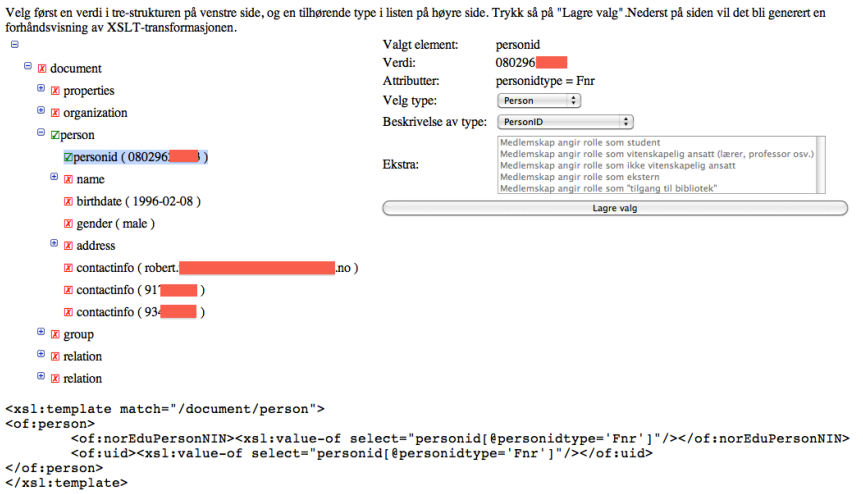

Figure 6.5: Screenshot of sample entry point for XSLTGenerator

6.3.3 Requirement Fulfillment

In section 5.4.7 and section 5.4.7, there were five functional requirements, and 7 non-functional requirements specified. The status of the delivered code in terms of these requirements are listed in table 6.3.

ID	Priority	Status	Comment
F-1	High	100%	
F-2	High	100%	Any valid XML can be analyzed, but results may vary
F-3	Medium	50%	The manual generation does not produce valid XSLT as of delivery
F-4	Medium	90%	The interface is shown, but is missing some functionality
F-5	Medium	100%	
NF-1	N/A	90%	Depends on client machine
NF-2	N/A	100%	
NF-3	N/A	100%	
NF-4	N/A	70%	Scoreboard system is still complex, and might be hard to replace
NF-5	N/A	90%	GWT server-side code is required to create client objects for communication
NF-6	N/A	100%	
NF-7	N/A	20%	Deep domain knowledge is needed for all cases to be covered

Table 6.3: Requirement Fulfillment

Chapter 7

Conclusion

7.1 Contributions

7.1.1 Viable Open Source for Consultancy Agencies

> ***RQ1**: Is authoring OSS a viable business idea for consultancy agencies?*

There are no clear answers to RQ1, and as with many other things in both computer science and business the answer is "It depends", but through exploring this thesis it is shown that it is possible. It does not come free however, and the consultancy agency must be flexible enough to not only rely on concultancy jobs, but might have to offer differenct services for the investments to pay off.

7.1.2 Practical Open Source Development

> ***RQ2**: When and how should free software be released?*

It is not hard to develop open source software in a practical manner, it might just be a bit different from internal development. The main issues are that you share your code, and your project with the world, and must have some methods for handling this. By reading and understanding the issues discussed in section 5.2, and following the recommendations made in section 6.2 takes you a long way for creating open source projects that follows known good standards.

7.1.3 Implementation of XSLTGenerator

The automatic processing and analysis of XML-documents, and the generation of XLST-transformations for converting the data into compatible formats seems promising, but has some major pain points that must be addressed before being considered production ready. The main problem lies within the

loose standards which govern the types of documents that are suitable for importing, such as ABC Enterprise, with much weight on the relation types. It's hard to figure out the appropriate access levels that should be granted out of non-specified strings. As a result, the process must always include some sort of human interaction.

The manual process does however provide the appropriate abstraction for a regular user to create a set of rules that might be converted into an XSLT-document. The problem is, as with all non-technical human interaction, is that the consequences of the actions are often not understood and might thus result in unwanted behavior. In the case of OpenFEIDE, this might result in the students getting access to every system that the teachers have, and thus gain access to setting their own grades and similar unwanted behavior.

Even though the delivered implementation is not in a state which makes it ready for usage, it have exposed some of the problems with an automatic/semi-automatic approach for generating the XSLT stylesheets for OpenFEIDE.

Chapter 8

Further Work

This chapter represents the authors view on how to do further work and research to better understand the issues addressed by this thesis.

8.1 Open Source For Consultancy Companies

There is currently a lack of scientific data on the creation of successful open source software in an business oriented manner. For a better understanding it would be very interesting with research following a project from its birth to monetizing by a consultancy agency. This is often hard to do since these processes often span over multiple years of development and iterations of the both the business model and product.

8.2 XSLTGenerator

XSLTGenerator in its delivered form, is not yet integrated into OpenFEIDE. If the project wants to incorporate the code into their own, even with the defects and risks discussed earlier, then it would have to be integrated using the provided APIs.

The two approaches discussed in the implementation has their own set of needed further work. The automatic approach lacks support for handling multiple memberships, while the manual generations approach needs further work in translating the input from the user to a workable XSLT template. As previously discussed, neither approach is, most likely, never going to provide a bulletproof way of integrating the software with external systems without any technical expertise. The automatic generator could however be used as a tool for integrators to give them a starting point for generating XSLT templates for the sources they work with. Creating this sort of tool, given the exposed API, is a trivial task for any programmer. An example of this is shown in figure I.2.

References

[1] *Case Studies for Software Engineers*, 2005.

[2] Timo Aaltonen, Jyke Jokinen, and Jyke Jokinen. Influence in the linux kernel community. In *OSS*, pages 203--208, 2007.

[3] Chris Abts, Ellis Horowitsh, A. Winsor Brown, Ray Madachy, Sunita Chulani, Don Reifer, Brad Clark, and Bert Steece. COCOMO II - model definition manual. `http://csse.usc.edu/csse/research/COCOMOII/cocomo2000.0/CII_modelman2000.0.pdf`, 2000.

[4] Izak Benbasat, David K. Goldstein, and Melissa Mead. The Case Research Strategy in Studies of Information Systems. *MIS Quarterly*, 11(3):369--386, 1987.

[5] Andrea Bonaccorsi, Silvia Giannangeli, and Cristina Rossi. Entry strategies under competing standards: Hybrid business models in the open source software industry. *Management Science*, 52(7):1085--1098, 2006.

[6] Andrea Bonaccorsi and Cristina Rossi. Altruistic individuals, selfish firms? the structure of motivation in open source software. *First Monday*, 9(1), 2004.

[7] Eric Braude. *Software Engineering: An Object-Oriented Perspective*. John Wiley and Sons,, United Kingdom, 2001.

[8] Henry William Chesbrough. *Open innovation: the new imperative for creating and profiting from technology*. Harvard Business School Press, 2003. ISBN 1-57851-837-7.

[9] Linus Dahlander. Appropriating the commons: Firms in open source software, 2004.

[10] Linus Dahlander and Mats G. Magnusson. Relationships between open source software companies and communities: Observations from nordic firms. *Research Policy*, 34(4):481 -- 493, 2005.

[11] E. A. Edmond. A process for the development of software for nontechnical users as an adaptive system. *N/A*, 1974.

[12] Martin Fowler and Jim Highsmith. The agile manifesto. *SOFTWARE DEVELOPMENT -SAN FRANCISCO-*, 2001.

[13] Øyvind Hauge, Carl-Fredrik Sørensen, and Reidar Conradi. Adoption of open source in the software industry. In Barbara Russo, Ernesto Damiani, Scott A. Hissam, Björn Lundell, and Giancarlo Succi, editors, *OSS*, volume 275 of *IFIP*, pages 211--221. Springer, 2008.

[14] Open Source Initiative. History of the OSI. `http://www.opensource.org/history`.

[15] ISO. ISO/IEC 9126-1:2001, Software engineering -- Product quality -- Part 1: Quality model. Technical report, International Organization for Standardization, 2001.

[16] Lingbo Kong, Shiwei Tang, Dongqing Yang, Tengjiao Wang, Jun Gao, and Jun Gao. Kcam: Concentrating on structural similarity for xml fragments. In *WAIM*, pages 36--48, 2006.

[17] Evangelos Kotsakis. Structured information retrieval in xml documents. In *Proceedings of the 2002 ACM symposium on Applied computing*, SAC '02, pages 663--667, New York, NY, USA, 2002. ACM.

[18] Sandeep Krishnamurthy. An analysis of open source business models. In *Eds.) Perspectives on Free and Open Source Software*. The MIT Press, 2005.

[19] Steve Krug. *Don't Make Me Think: A Common Sense Approach to the Web (2nd Edition)*. New Riders Publishing, Thousand Oaks, CA, USA, 2005.

[20] Jingyue Li, Finn Olav Bjørnson, Reidar Conradi, and Vigdis By Kampenes. An empirical study of variations in cots-based software development processes in the norwegian it industry. In *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*, 2004.

[21] Michael R. Lyu. Software reliability engineering: A roadmap. In *2007 Future of Software Engineering*, FOSE '07, pages 153--170, Washington, DC, USA, 2007. IEEE Computer Society.

[22] Sanjay S. Mehta. Marketing strategy - chapter 4 - SWOT: The analysis of strengths, weaknesses, opportunities, and threats, 2000.

[23] Parastoo Mohagheghi, Reidar Conradi, Ole M. Killi, and Henrik Schwarz. An empirical study of software reuse vs. defect-density and stability. In *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*, 2004.

[24] Neeshal Munga and Thomas Fogwill. Analysis of the value that open source contributes to business models, 2009.

[25] Neeshal Munga, Thomas Fogwill, and Quentin Williams. The adoption of open source software in business models: a red hat and ibm case study. In Barry Dwolatzky, Jason Cohen, and Scott Hazelhurst, editors, *SAICSIT Conf.*, ACM International Conference Proceeding Series, pages 112--121. ACM, 2009.

[26] Naur and Randell. Software engineering. *NATO*, 1969.

[27] Stephen O'Grady. Survival of the forges. `http://www.readwriteweb.com/hack/2011/06/github-has-passed-sourceforge.php`.

[28] A. Osterwalder, Y. Pigneur, and T. Clark. *Business Model Generation: A Handbook for Visionaries, Game Changers, and Challengers*. Wiley Desktop Editions Series. John Wiley & Sons, 2010.

[29] Leon J. Osterweil. Strategic directions in software quality, 1996. ISSN 0360-0300. doi: http://doi.acm.org/10.1145/242223.242288.

[30] Eric S. Raymond. *The Cathedral and The Bazaar*. O'Reilly Media, 1999. ISBN-13: 978-0596001087.

[31] Sandra A. Slaughter, Donald E. Harter, and Mayuram S. Krishnan. Evaluating the cost of software quality., 1998. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/280324.280335.

[32] IEEE Computer Society. IEEE computer society. IEEE standard glossary of software engineering terminology: IEEE standard 610.12-1990. number 610.12-1990 in IEEE standard., 1990. ISBN 1-55937-067-X. doi: http://dx.doi.org/10.1109/IEEESTD.1990.

[33] Jan Fredrik Stoveland. Managing sponsored open source communities. Master's thesis, Universitet i Oslo, 2008.

[34] Magnus Sulland. A study of requirement negotiation in open source communities. Master's thesis, Norges Tekniske og Naturvitenskapelige Universitet, 2010.

[35] W3C. Document object model core specification. `http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113/core.html`.

[36] W3C. XSL transformations (XSLT) version 1.0. `http://w3c.org/XSLT1`.

[37] W3C. XSL transformations (XSLT) version 2.0. `http://w3c.org/xslt2`.

[38] Wikipedia. XSLT processing pipeline. *Wikipedia*, 2011.

Appendix A

Company study: Varnish Software

Varnish Software is a company that offers services for the Varnish Cache software. They are currently 8 people, and are located primarily in Oslo, Norway.

(CQ1) Business models

Varnish Software offers multiple services surrounding Varnish Cache, some of which are package deals, such as a high level of support includes some of the proprietary tools. Varnish Software therefor uses a polyglot business model. Each of the different offerings are analyzed using the business model analysis framework create by Munga et al. [24].

Support

Multiple levels of support are offered ranging from €3300 to €33500 a year.

The Value Offering

Since the company is the core development contributors to the product they are supporting, the cost of training and maintaining the competency needed for keeping this offer attractive. Given the level of competence inherit in the company, it can offer support to a wide range of customer, both at the enterprise level and the small end of the scale.

By offering this service the company also gains knowledge about common pain points for the different customer groups, and may thus choose whether to solve these problems in the core open source product, and by doing so improving the quality of the product and increasing their market share. Or they can choose to write additional proprietary software for solving these problems, and sell them as standalone product, or as a value-add for support contracts.

The Market

The customers range from small businesses to large enterprises. All of which can benefit from the software. Large enterprises often require their IT-departments to only purchase and use software that have support contracts available for them, while small technical oriented companies wants the benefit of having support directly from the creators of the software.

As far as competition goes, the fact that it is the creators of the software that is providing the support gives a huge advantage in the market.

The Revenue Logic

The support contracts are based upon response times and the number of support requests that are handled. The higher levels are value-added by bundling additional tools and support features, providing the customers with good incentives for selecting a higher level of support.

Proprietary addons

Varnish Software offers addons and tools that interact with Varnish cache, such as control panels, monitoring software and more.

The Value Offering

Providing complementary products on top of the base open source software, the company reduces the costs of by not having to create individual products for each case. Creating these tools might also increase the sales of the other types of offerings, as the product might become a more attractive fit for potential customers if the complementary product is available, and making it tangible for them to use the software.

The Market

The customer of this ranges from the medium to large scale, as most small business only need the basic functionality that is provided with the base product. Since the company also provides support, and therefor

knows where the pain points for there customers are, are in a unique advantage as they know what type of complementary products that might sell.

The Revenue Logic

The complementary products are only sold as bundles with their support packages. This simplifies their offerings, while adding value to the support offers.

Training

Varnish Software offers three different training courses, in both online and classroom versions. These courses have a starting price at €1000 pr participant.

The Value Offering

They often say that the best way of learning something, is by teaching it. This reinforces the competency in the company, and by interaction with "students" they might come up with new ideas. As with the support offering, the training offers have a low cost, since the competency need for a good service is already inherit in the company.

The Market

Training is a versatile offer as the customer base spans all ranges and needs. Similar to the support offers, the company has a unique market position because of their direct involvement with the development of the software. This also means that they can underbid a lot of the competition since the internal training process is done in parallel with the other offerings.

The Revenue Logic

Because of the breadth of the potential customer base, and the different needs of the customers when it comes to training, the offer is very often interesting for the user of the software. As long as the deployment rate of the software is high, there will always be a customer base.

Consulting

Varnish Software offers multiple consulting packages in addition to custom consultancy offers. The price range varies from service to service, but the most packages starts at around €1000.

The Value Offering

Consultancy is a good way of learning more about the problem space of which the product lives, but again with a low cost since the consultants are also developers. If done correctly, the consultancy jobs might lead to up sale of some of the other offerings.

The Market

Wether the consultancy jobs are modifying the product, or integration with other systems, the company is still in a unique position to market and sell this offer, especially on jobs that require changes to be made on the product itself. The market however might be quite small, and mostly reserved for the large customers that have complex setups or needs.

The Revenue Logic

Consulting for this type of project is often integration projects that might be complex and require a lot of work. The size of the problems this offer attempts to solve makes this a profitable service.

(CQ2) How does the company interact with the community

As the both the project and the company is quite small, and the fact that they are the core contributors to the project results in them having a very active involvement in the community.

(CQ3) Is the business profitable

As only the public available information is at hand, it is too early to determine the profitability of the company. They have however announced that they are

hiring, and might be an indicator for them being profitable, or on their way of getting there.

(CQ4) How large is the company?

At the time of writing, Varnish Software consists of 8 full-time employees.

Appendix B

Company study: Gitorious AS

Gitorious AS is a company that offers services for the Gitorious software. They have currently two full-time employees, and are located primarily in Oslo, Norway.

Business models

The purpose of this section is to gain some insight into how Gitorious AS operates from a business perspective. Gitorious AS offers multiple services surrounding the Gitorious software, including SaaS, consulting and software management.

Software-as-a-Service

Gitorious AS offers hosted versions of the software starting at $99/month. The SaaS offerings include higher levels that allows for customization of the software.

Gitorious AS also offers a SaaS model of the product hosted on the customer's hardware. This is useful for installations within protected corporate networks, and thus expands the client base to include large corporations.

The Value Offering

> Providing hosting and support for your own product makes the product even more important, and quality arises from the fact that is reduces the amount of work you have to to per customer. This is a positive spiral, as the increased quality may attract more customers, and it also allows you to service more installations using fewer resources.

The Market

Since the company is the primary driver behind the project, they have a unique market position on which they operate, as they should, and probably do, know the software and how to operate it in a safe and stable way best.

There are however steep competition when it comes to the problem area with other companies such as GitHub and Atlassian offering their own versions. Gitorious AS does however underbid them both when it comes to on-site installations and support.

The Revenue Logic

The product area is huge, as every serious company that software development needs this type of software. That combined with the fairly low price tag, and the option for on-site installations, makes the potential customer base quite large, spanning from the smallest of teams, to enterprise installations.

If done correctly the operating costs of this service can be quite low, and thus it may provide a very profitable service for the company.

Consulting and Customization

Gitorious AS offers consultancy services, such as integration and customizing. They operate with a fixed hourly rate of $180.

The Value Offering

Because of the company's position regarding the software, the cost of doing this sort of work is quite low. There is little training needed, as the team already knows the software, since the wrote it.

The Market

The market spans from small teams who just want a custom stylesheet on their installation, to large enterprises who wants to integrate with other systems. This span, and the deep knowledge of the software makes Gitorious AS quite unique in the market.

The Revenue Logic

Consulting and customization are sought after features for most companies that want the software to meet there specific demands. With the price tag Gitorious AS sets on its consultancy services, most companies can afford to have these jobs done.

(CQ2) How does the company interact with the community

As the both the project and the company is quite small, and the fact that they are the core contributors to the project results in them having a very active involvement in the community.

(CQ3) Is the business profitable

As only the public available information is at hand, it is too early to determine the profitability of the company. They have however announced that they are hiring, and might be an indicator for them being profitable, or on their way of getting there.

(CQ4) How large is the company?

At the time of writing, Gitorious AS consists of 2 full-time employees.

APPENDIX C

PROJECT STUDY: GITORIOUS

The purpose of this section is to highlight some of the ways the Gitorious project is managed.

Case Study Answers

(PQ1) Source code management

Being software for hosting git source code repositories it's only natural that Gitorious is self-hosted using their own software.

The source code is publicly available at `http://gitorious.org/gitorious/mainline`. Commit access is restricted to a core developer team.

(PQ2) Release Process

Gitorious uses a rolling release process which means that there is no versioning or release processes.

(PQ3) Community

There does not seem to be a large community surrounding the software, but they do use mailing lists and IRC for communication.

The project includes a `HACKING`-file which gives the reader an introduction on how to contribute to the project.

(PQ4) Documentation

All the project documentation is distributed as a part of the source code. Except from overview and installation documentation, there is not much dedicated documentation.

(PQ5) Managing User Feedback and Support

There are no dedicated places for this except the regular mailing lists and IRC channels mentioned above.

(PQ6) Managing User Contributions

User contributions are expected to be posted to the bug tracking software, or as a pull-request in the version control software.

(PQ7) Bug tracking

Bug tracking is done using a self-hosted ChiliProject[1]-instance, and is publicly available at `https://issues.gitorious.org`.

(PQ8) License

Gitorious is licensed under the AGPLv3 license. It very similar to GPLv3, but has separate clauses for web sites, indicating that if modifications are made and the site is publicly available, then the source code must be available.

(PQ9) Deployment

Deployment on different operating systems are documented as separate files, and some include ready-to-go installation scripts.

Summary

Gitorious is a quite small and niche project, but seems to open and under constant development.

The key project points are listed in table C.1 for brevity.

[1] `https://www.chiliproject.org/`

Question ID	Part	Description	Public
PQ1	SCM	Self-hosted git	Yes
PQ2	Release Process	None	
PQ3	Communication	Mailing lists	Yes
PQ4	Documentation	OK documentation	Yes
PQ5	User Feedback and Support	Email and IRC	Yes
PQ6	User Contributions	Bug tracker or Pull Requests	Yes
PQ7	Bug tracking	Self-hosted ChiliProject	Yes
PQ8	License	AGPLv3	
PQ9	Deployment	Packaged	No

Table C.1: Key project management techniques used in Gitorious

Appendix D

Project Study: Varnish Cache

Varnish Cache is a very fast reverse proxy developed for serving web pages out of a cache, thus reducing the workload on application servers and give the user the ability to serve more customers.

Users of Varnish includes `vg.no`, `facebook.com` and `wikipedia.org`.

Case Study Answers

(PQ1) Source code management

Varnish uses git as their SCM-tool, and the repositories are publicly available at `git://git.varnish-cache.org/varnish-cache`.

(PQ2) Release Process

Releases are done by a release manager when a new version seems to be stable enough. There are no real processes behind the decision beyond what the core team of developers think.

There are three levels of releases, and the version is denoted by a three digit, dot separated identifier, e.g. 3.2.1 which denotes that it's major version 3, minor version 2 and bug-fix release 1.

(PQ3) Community

The community interaction is based upon mailing lists and forums available at the project homepage, as well as an IRC channel.

The mailing lists are split into 6 different lists, each for its own purpose.

varnish-announce Project announcements.

varnish-bugs Notifications from the bug tracking system.

varnish-commit Commit logs and discussions about specific commits.

varnish-dev Varnish developer discussions.

varnish-dist Discussion about packaging and distribution.

varnish-misc Free-for-all discussions about Varnish.

(PQ4) Documentation

Varnish is documented using Sphinx (a Python based documentation tool) and is publicly available at `https://www.varnish-cache.org/docs/`.

In addition to the official documentation collection, there is also a wiki available with more information.

(PQ5) Managing User Feedback and Support Requests

The project offers, and specifies which mailing list that is the appropriate for user feedback and questions. It also includes some guidelines how to ask questions, and how to reply.

(PQ6) Managing User Provided Contributions

Contributions such as bug-fixes and patches for new features are expected to be added to the bug tracker with the appropriate labels and descriptions.

(PQ7) Bug tracking

Varnish Cache uses a self-hosted Trac instance for their tracking bugs and feature requests. The tracker is publicly available at `https://www.varnish-cache.org/trac`.

(PQ8) License

The project is licensed under a 2-clause BSD license.

(PQ9) Deployment

Varnish is packaged by all the major Linux server distributions. They also have extensive documentation on installation and configuration available at their homepage.

Summary

The key project points are listed in table D.1 for brevity.

Question ID	Part	Description	Public
PQ1	SCM	Self-hosted git	Yes
PQ2	Release Process	Release manager	No
PQ3	Communication	Mailing lists	Yes
PQ4	Documentation	Excellent documentation	Yes
PQ5	User feedback	Mailing lists	Yes
PQ6	User Contribution	Bug tracker	Yes
PQ7	Bug tracking	Self-hosted Trac	Yes
PQ9	Deployment	Packaged	Yes
PQ9	License	BSD (2-clause)	

Table D.1: Key project management techniques used in Varnish Cache

Appendix E

Project study: Django

Django is a framework for web-development written in the Python programming language, and coins itself as "a web-framework for perfectionists with deadlines".

Released as open sourced in 2005 and has seen an incredible growth over the last couple of years.

Case Study Answers

The purpose of this section is to highlight some of the ways the Django project is managed.

(PQ1) Source code management

Django uses Subversion as its SCM-tool hosted by the project itself, but is publicly available at `https://code.djangoproject.com/`. Commit access is restricted to a team of core developers.

(PQ2) Release Process

The Django release process is documented in detail at `https://docs.djangoproject.com/en/1.3/internals/release-process/`. The releases are made by the core developer team.

(PQ3) Community

The community efforts of Django is enabled by heavy usage of mailing lists and IRC channels. There is very limited moderation of the communication channels, but the core developers do have the ability to do so if needed.

Django provides guidelines for community efforts in their documentation as well as reading material for people wanting to contribute to the project.

Mailing lists

There are two primary mailing lists used by the Django community. django-users, and django-dev. django-users has almost 22000 subscribers and is targeted for users of Django. django-developers has 6800 subscribers and is targeted for developers working on Django itself.

IRC-channels

As with the mailing lists there are two main channels for communication. Both are using the Freenode IRC-network, and are named #django and #django-dev. #django is targeted at the users of Django, whilst #django-dev is for the developers of Django itself. #django has around 400 users and #django-dev has about 85.

Conferences

As the Django community has grown the project has gotten dedicated conferences, both in the US and in Europe, DjangoCon and DjangoCon Europe.

Both these conferences are run annually and are always sold out with several hundred people attending each time.

(PQ4) Documentation

Django has a strict documentation policy, meaning that no feature is committed to the source code without proper documentation being available.

All the documentation is available at `https://docs.djangoproject.com/`.

(PQ5) Managing User Feedback

User feedback and support are handled through the commity channels. See above.

(PQ6) Managing User Contributions

Django handles its contributions by using a team of core developers as gate keepers to the source code, and the bug tracking software as central point for sending patches. This process is illustrated in figure E.1.

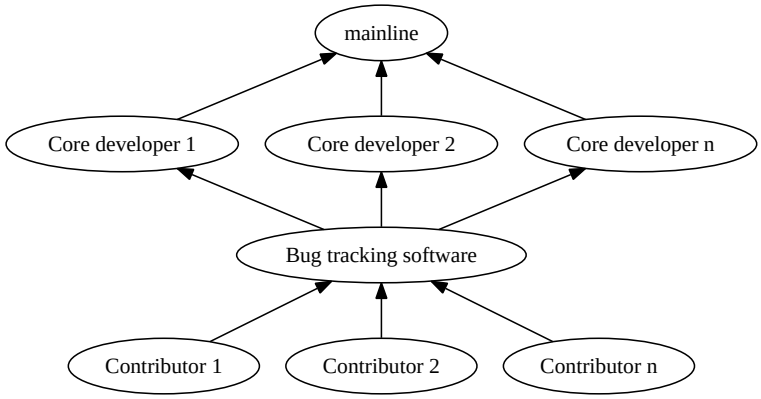

Figure E.1: Django contribution management

(PQ7) Bug tracking

Bug tracking is done using the Trac bug tracking software[1], and is publicly available at `https://code.djangoproject.com/`. The database has all the historical records dating back to the release of Django (bug #1 was opened 2005-07-13).

There are no restrictions on the triage process of bugs, so it is completely open, with the exception of security bugs which are only available to the core developers until a fix is released.

The bug tracking software is also used for new features and acts as such as a tool for managing user provided contributions.

(PQ8) License

Django is licensed under the permissive license BSD.

(PQ9) Deployment

Django includes documentation for deployment, and is also packaged for all major operating systems and flavours including Debian, Ubuntu, FreeBSD, and RHEL. Django is also installable using python packaging tools such as pip and easy_install with a single command.

[1] `http://trac.edgewall.org/`

Question ID	Part	Description	Public
PQ1	SCM	Self-hosted Subversion	Yes
PQ2	Release Process	Core developers	Yes
PQ3	Communication	Mailing lists and IRC-channels	Yes
PQ4	Documentation	Excellent documentation	Yes
PQ5	User Feedback	IRC and Email	Yes
PQ6	User Contributions	Bug tracker	Yes
PQ7	Bug tracking	Self-hosted Trac	Yes
PQ8	License	BSD (3-clause)	
PQ9	Deployment	Packaged, one-click-installs	Yes

Table E.1: Key project management techniques used in Django

Summary

The Django project is a well run open source project consisting of a core group of developers and a large community. The development process is very open, albeit strict due to the impact of bugs introduced in a project of its size and usage.

The key project points are listed in table E.1 for brevity.

Appendix F

Project study: Linux

Linux is the kernel which powers the GNU/Linux operating systems and was first released using a custom license in 1991. Linux 0.99 was the first version of Linux to be licensed under the GPL, and was released in December 1992.

Linux is estimated to be running on 60% of all servers, and can be described as one of the most successful open source projects to date.

Case Study Answers

(PQ1) Source code management

Source code management is done using the git version control software and is publicly available at `http://git.kernel.org/`. Linux uses a purely distributed development model as far as version control goes, but the de-facto mainline kernel is managed by Linus Torvalds, the original author of Linux.

(PQ2) Release Process

Mainline releases are done by Linus Torvalds, while older major versions are maintained and get bug fix releases controlled by different people.

(PQ3) Community

As the Linux developer community is so large the communication means varies greatly, but mailing lists are heavily used by almost everyone. The largest of these is the Linux Kernel Mailing List (LKML) which is the central point for development that is applied to the mainline kernel and averages between 350 and 450 emails each day. It has about 6500 subscribers which is quite high considering the high volume of emails.

In addition to announcements and patch notification, the LKML also acts as a review tool for new features and bug fixes. This however more often takes

place in the mailing lists for the particular sub-system of Linux that is being affected by the proposed change.

(PQ4) Documentation

Linux includes detailed documentation with its distributed source code, as well as man pages for system calls.

(PQ5) Managing User Feedback and Support

While user feedback and support request are often not asked directly to the kernel developers, but the the distributions that use the kernel, the same mailing lists that the developers user are accessable and public, and the only official point on where to get in contact with kernel developers.

(PQ6) Managing User Contributions

As mentioned the mainline kernel which makes its way to most distribution is controlled by Linus Torvalds. As the sheer volume of changes made to the kernel each day is so high (A publication by The Linux Foundation in August 2009 stated that each day 10,923 lines of code is added to the kernel, 5,547 lines removed, and 2,243 lines change), the merging of patches into the kernel are filtered by a web of trust around Linus. Linus get merge requests from people he trust to do a good job[1], these few people have people they trust and so forth as illustrated in figure F.1.

(PQ7) Bug tracking

The Linux kernel uses both the LKML and bug tracking software, with a preference for the LKML as the primary tool.

Their bug tracking software of choice is Bugzilla[2], and it's publicly available[3] at `https://bugzilla.kernel.org/`.

(PQ8) License

Linux is licenced under the GPLv2 licence.

[1] Often referred to as his lieutenants

[2] `http://www.bugzilla.org/`

[3] Currently not available 2011-12-01

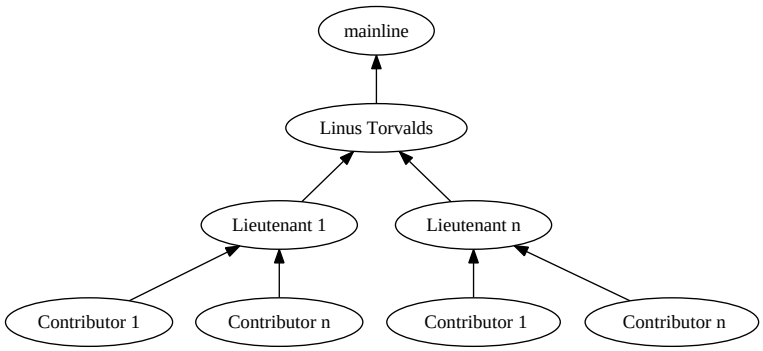

Figure F.1: Linux development contribution management

(PQ9) Deployment

Linux is mostly provided by the operating systems that use it and requires little to no effort to get installed as it is installed automatically with the operating system.

For custom installations Linux provides detailed instructions on how to build and package the kernel, but the installation may vary upon other software determined by the operating system that uses it.

Summary

The Linux kernel project is one of largest and the most famous open source projects, and has a high level of success. The development is very open, but also, due to the size of the project, fragmented. This fragmentation is handled by a hierarchical structure of trusted maintainers which produce the mainline Linux kernel that is widely used.

The key project points are listed in table F.1 for brevity.

Question ID	Part	Description	Public
PQ1	SCM	Self-hosted git	Yes
PQ2	Release Process	Managed by Linus	
PQ3	Communication	Mailing lists	Yes
PQ4	Documentation	Excellent documentation	Yes
PQ5	User Feedback	Mailing lists	Yes
PQ6	User Contributions	Mailing lists, Web of Trust	Yes
PQ7	Bug tracking	Self-hosted Bugzilla and Email	Yes
PQ8	License	GPLv2	Yes
PQ9	Deployment	Packaged	Yes (indirectly)

Table F.1: Key project management techniques used in Linux

Appendix G

Project study: Symbian OS

Symbian OS was once the most popular operating system for mobile devices, but with the rise of Apples iPhone it could not keep up. As an attempt to revitalize it, Nokia initiated the work for having it open sourced under the belief that the would automatically leverage the open source communities and get free/low-cost contributions to the operating system, which would once again make it a desired platform. The source code was to be released under the Eclipse Public License (EPL), a fairly well known and respected license.

However, after the initial publicity created by this, and the initial surge of interest from various developer communities, the source code and the needed infrastructure were never really made accessible in a meaningful matter.

As of today, the released source code lives only in a long abandoned code dump on SourceForge[1].

It is as such an interesting project to see as a worst case scenario for an open source release (no users, zero contributions, etc). It should be mentioned that Nokia a couple of years later shutdown the project of creating a living open source community around Symbian OS, relicensed the software under a proprietary license and created the "Symbian Platform" product based upon it.

Note: This study applies to the open source release of Symbian OS, and not the platform itself.

Case Study Answers

(PQ1) Source code management

Symbian released the source as complete packages and as Mercurial repositories, and were publicly available at `http://developer.symbian.org`.

[1] `http://sourceforge.net/projects/symbiandump/`

(PQ2) Release Management

No releases were made after open sourcing.

(PQ3) Community

The community was supposed to gather around the project website, with mailing lists as the primary communication channel for developers, and the wiki for documentation.

As with the Linux project, Symbian had dedicated mailing lists for each subsystem and project.

(PQ4) Documentation

Some of the documentation was provided to the general public using articles and a centralized wiki. Most of the documentation required the reader to register and login to gain access.

As little of this documentation exists today it's difficult to predict the quality of the documentation, other than the few that are reachable using the Internet Archive[2].

(PQ5) Managing User Feedback and Support Requests

Both mailing lists and bugtrackers were set up, but never active.

(PQ7) Bug tracking

The project used a self-hosted Bugzilla instance publicly available at `http://developer.symbian.org/bugs/`.

(PQ8) License

The source code was released under the EPL (Eclipse Public License).

(PQ9) Deployment

Actual deployment is not relevant to the general public, as specialized knowledge and hardware is needed. The distribution did include a deployment simulator for testing however.

[2] `http://web.archive.org/web/20091012040700/http://developer.symbian.org/main/documentation/`

Part	Description	Public
License	EPL	
SCM	Self-hosted Mercurial	Yes
Bug tracking	Self-hosted Bugzilla	Yes
Documentation	Good documentation	No
Communication	Mailing lists	Yes
Deployment	Packaged	No

Table G.1: Key project management techniques used in Symbian OS

Summary

The Symbian OS open source project was the largest conversion of proprietary source code to open source at its inception. This might have been its downfall, as a lot of the same tools and techniques used for managing the project as successful projects , such as Linux, was used.

But there was never any real community built around the product, and thus it never turned into the successful open source project Nokia had imagined.

The key project points are listed in table G.1 for brevity.

Appendix H

The Future of OpenFEIDE

Based upon the previous chapter these are some recommendations on how to migrate OpenFEIDE to a possible viable open source project, both in terms of business aspects and technical practicalities.

Practical Open Source Development

The purpose of this section is to give some recommendations for the OpenFEIDE project based upon the results in this thesis.

Source Code Management

As one of the core features of open source development is the availability of the code and a transparent development process, the source code should be available publicly and preferably as source code repository.

As mentioned in section 5.2.2 there are many ways of doing this, but the first choice is whether or not to use a source code repository hosting providers such as GitHub, or host the code on your own servers.

My recommendation is to use a remote service as this enables you to concentrate on creating a product, and not on hosting issues and the like. At the current time I would recommend the usage of GitHub as the provider. They provide all the basics that are needed for a open source project of OpenFEIDE's size. If Acando is to pay for the service they would also get private repositories. As a result of this they could have the public source code in one repository, and customized versions for customers in private branches. This way the code can easily be shared across repositories and reduce the complexities of maintaining different versions of the software.

GitHub has extensive documentation on how to start using the service, and how to migrate existing data from almost every source code management tool.

Documentation

Documentation is the key for users to feel confident that it is a serious and maintained project, and thus willing to try it.

The first piece of documentation that should be written in detail is installation instructions, both for production environments and for developer environments.

There is no need for any advanced documentation scheme. Simple textfiles, or wiki pages are good enough, at least for user documentation. Developer documentation such as Javadoc and design documents might require some other formats. If the source code is hosted at some remote hosting provider, it probably has some sort of wiki-functionality that can be used for this.

Keeping the documentation easy and accessible is crucial for it to be read.

Deployment and Ease of Use

One of the big hurdles for new users is often the installation of the product. If they don't manage to install it, they will never use it. Because of this, any application should be packaged and distributed in a way that makes it dead simple for the user to install. As mentioned in section 5.2.1, if this is done correctly, the project will have well defined upgrade paths for all installations. This adds operational quaility the product, and may result in operation teams to prefer your software over other projects that are harder for them to keep updated.

As OpenFEIDE is software that is run on servers, the first two platforms to package the software for is Debian[1] and Red Hat Enterprise Linux[2]. Both projects have excellent documentation on how to create packages, `http://pkg-java.alioth.debian.org/docs/tutorial.html` for Debian and `http://docs.fedoraproject.org/en-US/Fedora_Draft_Documentation/0.1/html/RPM_Guide/index.html` for Red Hat, CentOS and Fedora (they all use the same package manager, and package format). The Debian packages are compatible with Ubuntu as well.

Managing Contributions

Contribution management should be handled in a way that makes the people that tries to send contributions feel like they get a response from the develop-

[1] `http://www.debian.org`

[2] `http://www.redhat.com`

ers. There are many ways to do this, and as most projects uses bug tracking software for solving these issues, it is often a good place to start for handling these contributions, as it requires little effort and knowledge from either side for managing this.

If modern distributed version control systems are used, such as git or mercurial, the contributions could be taken as "pull requests". "Pull requests" are requests for the developers to "pull" (fetch and merge) either a branch of the contributors code, or a patch set. There are some differences on how this is handled from one version control system to the next, but almost all the hosting services for these have built-in functionality for handling such requests. See GitHub's help page[3] for an example on how this is done using these tools.

Community

The probability for a product like OpenFEIDE to gain a huge community is fairly small. The number of people would most likely be something like $n * 2$ where n is the number of user organizations. This is similar to what Uninett's NAV project has. That being said, even though the community is small, it does not mean that is does not need some infrastructure for communication. IRC channels are a popular choice for instant messaging, and mailing lists for more structured discussions. There are a lot of free services providing these features. The Freenode[4] IRC network is a popular choice for open source projects, while Google Groups is favoured for mailing lists. Using these services, the total amount of time setting the infrastructure up is roughly five minutes, and is thus trivial.

Migration

OpenFEIDE is at the time being, only available internally for Acando employees. The source code repository should be migrated to be publicly available. However before this can be done the current repository must be restarted or modified as it includes example data with confidential data. This can be done by importing the subversion repository into git and then following `http://help.github.com/remove-sensitive-data/` to remove the offending file from the history, and then create a new subversion repository based upon the git repository (or keep using git).

[3] `http://help.github.com/send-pull-requests/`

[4] `www.freenode.net`

Once the code is public OpenFEIDE should establish a common place on the Internet for finding information about it, downloading it and reporting bugs. This is also important from a business perspective as it makes the project more visible to the wider community.

Viable Open Source

License Selection

Choosing the license is, as mentioned, not always easy, and requires some thought. My recommendation for OpenFEIDE is the Apache Software License v2. There are couple of reasons for this. First of all, being a permissive license, it does little to restrict Acando's future use of the code base, and they might reuse parts of the project for other efforts without having to worry about licensing issues (given that the other efforts are of a proprietary nature). The second reason is given the technology selection (see table 5.2), using the Apache license is a safe choice, as most of the other software components that is used by OpenFEIDE are also licensed under it. This reduces the chances of mixing incompatible licenses.

The drawbacks of selecting the Apache license, is as with any projects released under permissive licenses, is that you are not guaranteed to get access to other versions of the software.

Business Models

OpenFEIDE is a product with special integration needs. This is due to the fact that most organizations have different data sources for it to use, and different access control requirements.

As the integration situations as so varied and complex, there should be a market for doing deeper integration and customizations. These changes can be kept by Acando if they so choose, and reused to reduce cost in other deployments without releasing these changes to the public.

A more radical step for a consultancy agency like Acando is to provide OpenFEIDE as a service. If done right, and the infrastructure to support it can be obtained at a low cost, this could be quite profitable. It does, however, require the company to provide the necessary resources for maintenance and running costs of the solution.

In this section some of the possible business models are analyzed in context of OpenFEIDE.

Software-as-a-Service

SaaS is providing the customer with the software as a service, and not as set of executables that they run on their own. The business model has had rapid growth the last couple of years as the internet gets increased adoption and better available infrastructure, such as fast connections.

Value for the customer

SaaS is an attractive business model for many companies because of its stable economic nature. Almost all the unknown factors introduced by IT-systems are removed, due to the fact that the customer does not have to provide the infrastructure, nor services for the product.

- Reduce operational costs
- Fixed pricing (no hidden costs)
- Small number of unknown factors

Value for Acando AS

The main value for the hosting company is the possibility of increasing revenue by being able to parallelize work across multiple installations, and may thus handle more customers with fewer resources. Another value is gained from having complete control over the environment that the software runs in, which may reduce the complexity of operations and support greatly.

- Self-controlled environment reduces risk
- The cost of serving another customer after the initial one is low.

Risks for Acando AS

There are risks in any business model, and SaaS business models definitely have them. Managing and hosting system critical software is a science in its own regard, and is easy to get wrong. In order to handle such requirements, investments must be made, and are not easily reverted if the business model fails. The investments add more expenses to the list, and thus requires a higher amount of customers to become profitable. One additional risk with OpenFEIDE in mind, is the customizations that must be made for each customer. These might add up to being so differentiated from other setups, that it cancels out the value of having multiple instances.

- Lack of needed skills and resources for operating the environment?
- Differences between setups might be large, and might add a lot of complexity.
- Needs critical mass for hosting and resource allocating to pay off.

Support

Support is always a pretty low risk service to offer, as it is very flexible. In the start-up phase of monetization of the project, the supported levels might be quite relaxed and only handled by the developers as a "side job" to programming. This gives the customers some security at a very low cost for Acando. If the number of customers increases to the point were the developers do not cope anymore, then it could be assumed that the market is present for expanding the support offerings, and dedicate resources for that purpose.

Value for the customer

- Support directly from developers
- The availability of support contracts (often a requirement)

Value for Acando AS

- The cost of serving another customer after the initial one is low.
- Low initial costs if developers are used for support calls
- Low risk as there are few investments
- Potentially high profit if software is stable
- Knowledge of where the pain points for the customers are, can result in either better software or proprietary addons if solved.

Risks for Acando AS

- Too much pressure on support could slow development down

Support is often the first line of attack for commercializing open source software, and not without reason. The low risks and costs of setting this business model up, and the potential high profits, makes it a prime candidate for one of the offerings the company can provide.

Customization / Integration

Consultancy services are well suited for projects as OpenFEIDE because of its need for integration into other systems. One could argue that almost every installation of OpenFEIDE would need these services, either from Acando, the customers own IT department or a competitor.

Value for the customer

- Custom tailored integration for their needs
- Developers that are guaranteed to know both the project and the problem domain.

Value for Acando AS

- Consultancy jobs are highly profitable due to high prices
- Possibilities of up-selling other products by close customer relations

Risks for Acando AS

- Given a limited set of customers, one time integration jobs will run out.

Business Model Recommendations

Offering only one type of services is probably not enough to cover enough customer needs to make a profit. This seems to be the case for many open source projects, and Acando should attempt to offer multiple of these services, and iterate and adapt them as one gets customer feedback.

The model selected by Varnish Software seems to be a good fit for the type of speciality software like OpenFEIDE, meaning offering support and consultancy services. The SaaS model has too much risk at the startup phase to validate the investments needed for it to bring enough value to the customers for being attractive.

Appendix I

XSLTGenerator Documentation

Library module

`libxsltgenerator` is the library that powers the analysis and actual generation of XSLT-files. It is included as an appendix to the thesis and is located in the libxsltgenerator folder.

Installation of libxsltgenerator

The library is packages as a Maven module, so all you have to do to make it available for your applications is to run:

```
cd libxsltgenerator && mvn clean install
```

Usage in Other Maven Projects

If the installation process succeeded without errors, you can include the library in other maven projects by adding the following to the project pom-file's dependencies section:

```
1  <dependency>
2    <groupId>no.ntnu.ime</groupId>
3    <artifactId>libxsltgenerator</artifactId>
4    <version>0.0.1-SNAPSHOT</version>
5  </dependency>
```

Example Library Usage

The exposed functionality of the provided API is quite small, and done so by design to keep the usage as simple as possible.

The following is a complete example on how to retrieve, analyze and generate XSLT from a XML document on the local hard drive.

```
1
2  public String fetchXSLT(){
3    String FIXTURE_XML = "file:///home/developer/fixture.xml";
4
5    XSLTGenerator generator = new XSLTGenerator();
6
7    // Analyze the given document
8    Map<String, List<SaxParserResult>> parsingResults =
         generator.parseXMLSource(FIXTURE_XML);
9
10   // Generate XSLT for the entire document
11   String generatedXSLT = generator.getXlstForDataset("/",
         parsingResults);
12
13   // Get the original XML document as a File-instace
14   File xmlSourceFile = generator.retrieveXMLSource(FIXTURE_XML);
15
16   // Invoke the XSLT Pipeline on the generated XSLT on the source
         xml document
17   String translatedXML =
         generator.attemptConversion(generatedXSLT, xmlSourceFile);
18
19   return generatedXSLT;
20 }
```

Figure I.1: Example Library Usage

Console Applications

The library can easily be used as a command line tool for creating starting points for new XSLT templates. A basic example is shown in figure I.2.

```
1  package no.ntnu.ime.libxsltgenerator;
2
3  import java.io.File;
4  import java.io.FileWriter;
5  import java.util.List;
6  import java.util.Map;
7
8  import no.ntnu.ime.libxsltgenerator.models.SaxParserResult;
9
10 public class Tool {
11
12   public static void main(String[] args) throws Exception {
13     if (args.length < 3){
14       System.err.println("Usage: java -jar xslttool.jar input.xml
             output.xsl");
15       System.exit(1);
16     }
17     String source = args[1];
18     String sourceURL = new File(source).toURI().toString();
19     XSLTGenerator g = new XSLTGenerator();
20     Map<String, List<SaxParserResult>> result =
           g.parseXMLSource(sourceURL);
21     String xslt = g.getXlstForDataset("/", result);
22     File output = new File(args[2]);
23     FileWriter fileWriter = new FileWriter(output);
24     fileWriter.write(xslt);
25     fileWriter.close();
26     System.out.println("Done");
27   }
28 }
```

Figure I.2: Example Commandline Tool

GWT Web Application

The example web application that uses the library is located in the XSLTGenerator-folder in the included source code.

In addition to the automatic parser example, the application also includes the manual selection prototype.

Compile and Run

The source code includes a maven configuration that allows you to test the application in an production like environment.

To compile and run the code in such an environment, use the following command:

```
mvn clean package jetty:run-war
```

If successful, the application is available at:

```
http://localhost:8080/XSLTGenerator.html
```

Note:
GWT only works on 32bit JVM.
Make sure that you use a 32bit JDK and JVM (i586), if you are on a 64bit system (x86_64)