Journal of the Association for Information Systems

JAIS

Special Issue

Adopting Free/Libre/Open Source Software Practices, Techniques and Methods for Industrial Use*

Richard Torkar
Blekinge Institute of Technology
richard.torkar@bth.se

Pau Minoves
i2cat Foundation
pau.minoves@i2cat.net

Janina Garrigós
i2cat Foundation
janina.garrigos@i2cat.net

Abstract

Today's software companies face the challenges of highly distributed development projects and constantly changing requirements. This paper proposes the adoption of relevant Free/Libre/Open Source Software (FLOSS) practices in order to improve software development projects in industry. Many FLOSS projects have proven to be very successful, producing high quality products with steady and frequent releases. This study aims to identify FLOSS practices that can be adapted for the corporate environment. To achieve this goal, a framework to compare FLOSS and industrial development methodologies was created. Three successful FLOSS projects were selected as study targets (the Linux Kernel, the FreeBSD operating system, and the JBoss application server), as well as two projects from Ericsson, a large telecommunications company. Based on an analysis of these projects, FLOSS best practices were tailored to fit industrial development environments. The final results consisted of a set of key adoption opportunities that aimed to improve software quality and overall development productivity by importing best practices from the FLOSS environment. The adoption opportunities were then validated at three large corporations.

* Michael Wade and Kevin Crowston were the accepting guest editors. This is one of the accepted papers for the Special Issue on Empirical Research on Free/Libre Open Source Software in 2010 Issue 11 and 12. This article was submitted on 12th October 2009 and went through two revisions.

Volume 12, Issue 1, pp.88-122, January 2011

1. Introduction

To structure the software development process, organizations generally adopt a software development methodology. Avison and Fitzgerald (2003) define a software development methodology as "a recommended collection of phases, procedures, rules, techniques, tools, documentation, management and training used to develop a system." There are many available software development methodologies that an organization can follow to drive its projects, ranging from the traditional waterfall model to more modern ones that adopt an agile approach. For example, Ericsson AB, one of the major software producers of telecommunication systems and the site of the present study, uses a software development methodology called Streamline. Created by and for Ericsson, this methodology is widely implemented in-house with some project-specific variations.

In parallel with the development of formal methodologies, over the past twenty-five years Free/Libre/Open Source Software (FLOSS) communities have evolved a distinctive way of producing software. Noticing the success of several FLOSS projects, industry has shown particular interest in understanding how the massively distributed development teams commonly found in FLOSS communities manage to deliver high quality software. Distributed development poses significant challenges for software developers, but FLOSS teams often seem to be able to overcome these challenges. Based on the hypothesis that industry can benefit from adopting some practices from FLOSS development, the aim of this study is to collect, identify and analyse relevant FLOSS software development practices and then transform them to general adoption opportunities.

Free/Libre/Open Source Software has been an object of research for some years. FLOSS projects have been analysed mainly from two perspectives: as a product or as a development methodology. Studies investigating FLOSS as a product focus on measurable characteristics of the software or projects, such as defect density, software packaging statistics, software growth or number and type of contributors. Target projects of such studies have included the Apache httpd server (Paulson et al., 2004), the Linux kernel (Paulson et al, 2004), the GCC compiler (Paulson et al., 2004), the Debian Linux distribution (Amor et al., 2005; González-Barahona et al., 2001), OpenBSD (Li et al., 2005), and the Eclipse development environment (Herraiz et al., 2007). In the second group, we find studies investigating FLOSS as a way of producing software. These studies can be broadly categorised by their focus on community culture (e.g., Glass, 2003), organisational models (e.g., Gacek and Arief, 2004; Lattermann and Stieglitz, 2005) or processes and methods (e.g., Scacchi, 2002; Warsta and Abrahamsson, 2003).

While this previous research is relevant for understanding and extracting practices from FLOSS communities, how companies might benefit from adopting FLOSS processes and methods is still a largely untouched topic. One of the few examples of an initiative in this direction is HP's Progressive Open Source (Dinkelacker et al., 2002). Dinkelacker et al. focused on the FLOSS communities' openness, trying to adapt it for an industrial environment. Specifically, they created an infrastructure for HP's projects that increases code visibility within the organisation, trying to achieve the benefits of code visibility in FLOSS projects.

To achieve the main goal of this study, i.e. to detect and extract additional FLOSS practice adoption opportunities, five software projects were compared. Two belonged to Ericsson AB, while the other three were the following FLOSS projects: the Linux Kernel, the FreeBSD operating system and the JBoss application server. Thus, this study has a broad scope, setting up a case study with an emphasis on development practices, methods and techniques, spanning five projects. The study was performed between January 2009 and May 2010.

The most important result from this study is a list of suggested "adoption opportunities" for industry, that is, distinctive FLOSS development practices that might have benefit for development in a company setting. Additionally, as an intermediate step, we provide an analysis of the differences between FLOSS methodologies and the Streamline methodology as used by Ericsson. As a basis for the analysis, we created a framework to compare development methodologies, a framework

especially tailored for comparing FLOSS methodologies with other methodologies. The final step in the study was a validation of the results (Gorschek et al., 2006) through a series of workshops performed at Ericsson and two other large companies. Like Ericsson, these two companies make use of agile methodologies when producing software-intensive systems, but in other domains, thus contributing to the generalizability of our recommendations.

The remainder of the paper is structured as follows: In Section 2, we explain the framework developed to structure the comparison of Streamline and the FLOSS development methodologies (further detailed in Appendix A). In Section 3, we describe the research methodology adopted. The results from the comparison, with an accompanying analysis, are found in Section 4 and Appendix B, while the outcome of the study, i.e. the recommended adoption opportunities, can be found in Section 5. Section 6 presents a validation of the study results, while Section 7 covers validity threats. Finally, Section 8 concludes the paper and presents future and on-going work.

2. Initial Study Framework

In this section, we discuss the development of an initial framework to guide data collection for the Streamline and the FLOSS development methodologies. We first discuss the goal of having a framework. We then present possible comparison frameworks found in the literature, the reasons why they were or were not selected for our study and finally, the framework that was chosen as a base for creating the final comparison method.

Due to the exploratory aim of this research, a framework that prompted consideration of all the methodology viewpoints was desired. The desired outcome of a comparison would be a complete list of differences between the methodologies used in the Ericsson and FLOSS projects. One of our requirements was a comparison framework that would help to elicit non-obvious issues, as well as the capacity to elicit issues at different levels (activities, principles, roles, etc.). We considered several comparison frameworks to find one that would be most suitable for our purpose. We next briefly review the frameworks we considered.

We first considered the Normative Information Model-based System Analysis and Design (NIMSAD) presented by Avison and Fitzgerald (1995) and Avison and Taylor's (1997) comparison framework. However, we found that these frameworks assumed the existence of a problem the methodology is trying to address and consequently focused on suggesting the most appropriate methodology depending on the encountered problem. We did not want this problem-oriented approach, as we wanted to find differences at different levels, so these frameworks were not adopted. Similarly, Davis's (1982) framework was discarded for being too oriented towards requirements engineering.

We also considered the Compare Design Methodologies (CDM) originally proposed by Song and Osterweil (1994) and as applied by Guimarães and Souza Vilela (2005). CDM was not adopted for two main reasons. First, it is used to compare methodologies of the same family, assuming a low level of difference, and is therefore focused on spotting small differences. For this reason, we were concerned that it might be unsuitable for comparing two potentially very divergent ways of working. Second, its purpose is to find the best methodology among some candidates, based on some subjective criteria. We were not interested in finding whether FLOSS was better than Streamline or vice versa, but rather to identify the concrete differences between the methodologies.

We next turned to Avison and Fitzgerald's (1995) frameworks. Avison and Fitzgerald (1995) identified two reasons for comparing methodologies: an academic reason, that is, to understand the nature of methodologies (our goal), and a practical reason, that is, to guide the selection of a methodology for a given development. Taking these premises as starting point, they proposed two frameworks. The first framework provided a list of twenty-six 'ideal-type' criteria that a methodology should fulfil, as shown in Table 1. A problem with this framework is that it may be out of date. For instance, the known preference of agile methodologies for having little documentation is considered a negative trait by this framework.

Table 1. Avison and Fitzgerald Assessment Criteria		
Rules	Information system boundary	Total coverage
Designing for change	Understanding the information resource	Teach-ability
Documentation standards	Effective communication	Validity
Simplicity	Early change	On-going relevance
Inter-stage communication	Automated development aids	Effective problem analysis
Consideration of user goals and objectives	Planning and control	Participation
Performance evaluation	Relevance to practitioner	Internal satisfaction
Relevance to application	External satisfaction	Scan for opportunity
Visibility	Separation of analysis and design	

Avison and Fitzgerald's (1995) second framework includes seven basic points of view from which a methodology could be described (see Table 2). The application of this framework provides an academic description of a methodology that allows for a structured comparison, ranging from its philosophical background to the final form that practitioners consume. We felt that this framework provided a good overall framework for describing a development methodology, but as stated was too abstract for our needs. The framework ignored some details that could uncover attributes that would be very valuable for the comparison phase, such as communication and coordination issues or life cycle details.

Table 2. Avison and Fitzgerald Points of View for Describing a Methodology	
• Philosophy o Paradigm o Objectives o Domain o Target • Scope	• Output • Practice o Background o User base o Participants • Product

Our final approach was to combine and customise the two Avison and Fitzgerald (1995) frameworks to reinforce their strengths and alleviate some weaknesses. As Avison and Fitzgerald (1995) themselves stated "the above criteria (Table 1) form a useful checklist but clearly need to be tailored for a particular purpose." Specifically, we used both frameworks as a prompt for our initial data collection, but then created a tailored comparison framework that included aspects such as software development life cycle, degree of participation and an increased focus on what might be considered as relevant activities. The use and tailoring of these frameworks will be discussed in the following section.

3. Research Methodology

In this section we describe the research approach adopted to identify features of the software development methodologies followed in the three FLOSS projects and in Ericsson's Streamline methodology. Overall, the study was conducted as follows: i) develop a comparison framework (as described in Section 2); ii) perform a case study of the development methodologies of the five projects; iii) derive a set of adoption opportunities from comparison across the results of the case study; iv) perform a validation (Gorschek et al., 2006) of the results through a series of workshops at three different companies in three disparate domains. More specifically, we followed a four-step process to produce the possible adoption opportunities, explicitly modelling our methodology on the grounded theory approach (Martin and Turner, 1986). The first two steps were inductive (Martin and Turner, 1986), where data collection preceded theory formulation. The concrete steps were as follows:

1. Collect information about the development methodology applied at Ericsson and in FLOSS projects.
   (a) Grounded theory equivalent: Data-collection, note taking and open coding.
2. Analyse, synthesise and categorise the information in order to allow a side-by-side comparison of FLOSS and Ericsson methodologies.
   (a) Grounded theory equivalent: Memo building, axial and selective coding, sorting and writing.
3. Compare the methodologies and identify the differences that are significant (differences in which FLOSS can provide an improvement for Ericsson).
   (a) Side-by-side comparison, discussion and industry feedback.
4. Suggest FLOSS methodology adoption opportunities to Ericsson and other companies.
   (a) Comparison analysis and industry feedback.

Section 3.1. presents the selected projects and their fundamental characteristics. Section 3.2. describes how the main information gathering was conducted, and Section 3.3. outlines how the comparison was carried out and the tailored comparison framework created.

3.1. Case Study Setting

In this subsection, we discuss the choice of projects for our study. As a qualitative research approach was used, only a small number of projects could be examined. For the industry projects, two projects inside the development unit were available for the case study. Due to confidentiality reasons, we will refer to them as Projects A and B. Both Projects A and B used the Streamline methodology.

To achieve a list of adoption opportunities from FLOSS development that might be applied at Ericsson for project/process improvement, three FLOSS projects were identified. There are many successful FLOSS projects, with very diverse configurations. A small sample could never be representative of the whole FLOSS universe, and this was never the intention. Instead, a decision was made to favour a sample that resembled the targeted Ericsson projects, since such a sample would be more likely to reveal suitable adoption opportunities. To make the appropriate choice, some initial Ericsson project data were collected. This data served as an initial contact with the Streamline methodology and more importantly, as a way to characterise Ericsson projects in terms of size, number of developers, domain, and so on. These gathered characteristics were used to guide the selection of FLOSS projects that resembled the Ericsson ones. The following sections describe the attributes of Projects A and B and the selected FLOSS projects.

3.1.1. Industry Project Characteristics

Several attributes were considered to characterise Ericsson's software development processes and methods. However, the final list was synthesised down to four. These were the characteristics used when making the FLOSS project selection:

- Large source code size. Project A contained around 1,250 KLOC while Project B contained 110 KLOC. Hence, we were interested in FLOSS projects with a fairly large code base.

- High number of developers. 90 developers worked full time at Project A (around 14,400 man-hours per month). 40 developers work at Project B (around 6,400 man-hours per month) with this number doubling in less than a year. Considering the inequality of dedication between industry and FLOSS developers (on average FLOSS developers invests less than 10 hours a week on development activities (Robles et al., 2001), we were looking for FLOSS projects with more than 100 active contributors.

- Legacy. Project A started seven years ago and, thus, carried a large amount of legacy code. Project B, while it was started three years ago, reused and maintained the code of a nine-year-old project. Therefore, legacy was a significant property of both projects.

- Similar domain. As all software projects, Projects A and B design decisions were driven by

many non-functional requirements. Some of them, like robustness and performance, could be elicited, but many more subtle ones remained hidden. To have projects with similar non-functional requirements, FLOSS projects from a similar domain were selected. This domain was generally stated as enterprise server-side software with an emphasis on high availability and performance.

3.1.2. FLOSS Project Characteristics

Three FLOSS projects were chosen as study targets because they were similar to the industry project characteristics and were especially interesting from a development methodology point of view. The selected FLOSS projects were the Linux kernel[1], the JBoss Application Server[2] and the FreeBSD operating system[3]. All three projects are supported by a large group of contributors, over many years, and have managed to create large and established source code repositories. Also, all three communities have relevant industry participation. This factor is particularly important as it ensures that the FLOSS projects are valuable enough to become commercially interesting. Additionally, that paid and volunteer contributors coexist in the same development community is a new relevant trend in FLOSS projects. Table 3 presents descriptive statistics for all projects as of April 2009. However, other attributes of interest were also considered when selecting the projects:

- The Linux kernel. With an impressive community of over 1,000 developers, 70% of which serve the interests of over 100 companies (Kroah-Hartman et al., 2008), the Linux kernel project is probably the largest example of FLOSS engineering at work. Of special interest to our study is the issue of balancing interest diversification in functionality requests, while maintaining robustness and performance.

- JBoss Application Server. JBoss community development is strongly tied to a commercial firm (JBoss, a division of Red Hat[4]). This sponsoring produces an interesting mix of FLOSS and industry processes, especially in the software configuration management and quality assurance areas. Additionally we find pressure to respect release schedules and promised features as commonly found in traditional industry projects.

- FreeBSD. The FreeBSD development philosophy has similar principles concerning industry's continuous integration practices as applied by Projects A and B (Jørgensen, 2001). This feature was a decisive attribute to select FreeBSD as a project to study.

Table 3. Descriptive Statistics of Selected Projects

	Project *A*	Project *B*	Linux kernel	JBoss	FreeBSD
Lines of code (thousands)	1,250	110	8,500	1,200	6,000
Developers	90	40	1,000	50	200
Legacy (years)	7	9	15	10	13

3.2. Information Gathering

An important part of this study was the information-gathering phase. All the information found from the studied projects was collected using field notes that were then analysed to compare the methodologies. We aggregated the methodology information about the three FLOSS projects in addition to the information concerning the two Ericsson projects. The information gathering was performed with the main objective in mind of defining the development methodology of the five studied projects to allow for structured comparisons. The Avison and Fitzgerald matrix was used to

[1] http://www.linuxfoundation.org
[2] http://www.jboss.org
[3] http://www.freebsd.org
[4] http://www.redhat.com

ensure a broad coverage of the methodology characteristics. Additionally, in order to target concrete and real world development challenges, practitioners' subjective opinions about the development methodology was collected in order to focus the research on interesting fields that could lead to adoption opportunities.

The amount of information available in written form is overwhelming both at Ericsson and, to some extent, in a typical FLOSS project. However, a completely different issue is how close this documentation is to reality once theory is put into practice and time passes. We wanted to study the real practice of the methodologies, with its problems, issues and workarounds as perceived by the practitioners. For this reason, whenever possible, we used the practitioners' knowledge to confirm and extend the information found. Sources of information for the Ericsson projects included:

- **Documentation**, including descriptions of processes, roles and activities as well as project specific methodology implementation documentation. Also, the development wikis proved to be a useful source of technical and teamwork data.

- **Interviews** were carried out with developers in six different key roles in each project, offering different points of view over the complete life cycle of the project. Responsible engineers in the following areas were interviewed: requirements, design, testing, and project management. Additionally, one engineer responsible for development methodology Ericsson-wide provided insight on the Streamline principles. The interviews were conducted in a semi-structured manner as described by Hove and Anda (2005). This study was executed at an Ericsson development site, allowing for high interaction between researchers and practitioners.

- **A survey** was conducted. Once in the comparison phase it became necessary to gather the point of view of the developers and testers about several key topics. To gather data from a representative sample of the developers of both projects, a survey was designed and executed. The questions covered task assignment, tools, ways of working, documentation, communication and decision-making. The survey also provided a practitioner's opinion overview, very much as in Torkar and Mankefors (2003). The survey targeted 130 developers and had 92 respondents during the 22 days it was open (leading to a response rate of more than 70%).

When studying FLOSS projects, we relied on three information sources.

- **Documentation**, including projects' homepages, wikis, guidelines, handbooks and mailing list archives. The open nature of FLOSS projects in general meant that a lot of data was publicly available. The data was later used to extract information regarding processes, roles and activities.

- **Overview of research literature**. As we selected three well-known FLOSS projects, there was a certain amount of previous research available covering the three projects. Although as stated before, previous research on these projects was usually quantitative in nature, it was still very useful in supporting our qualitative analysis.

- **Survey**. Due to time constrains, an adequate round of interviews in the FLOSS communities could not be performed. Instead, this part of the research relied on previous research literature, an already conducted survey (Robles et al., 2001) and first-hand documentation from the FLOSS communities.

3.3. Analysis Approach

Figure 1 offers an overview of the comparison method adopted. Readers familiar with the Osborn-Parnes Creative Problem Solving (Hurson, 2007) method will recognise the divergent-convergent thinking pattern. The first steps, up to the field memo, were used to generate as many ideas about the

methodologies as possible. The Avison and Fitzgerald (1995) frameworks discussed above were used as a prompt for this part of the study, ensuring that we had examined each methodology from multiple viewpoints. The last steps were designed to synthesise all information and reveal both important issues and the general information structure. The described technique produced two results: first, a more appropriate 'table of contents' for the framework used for analysing and presenting the differences, leading to our final comparison framework (presented in Appendix A); and second, a list of knowledge gaps with respect to the methodologies. Filling these knowledge gaps brought the comparison to its final state.

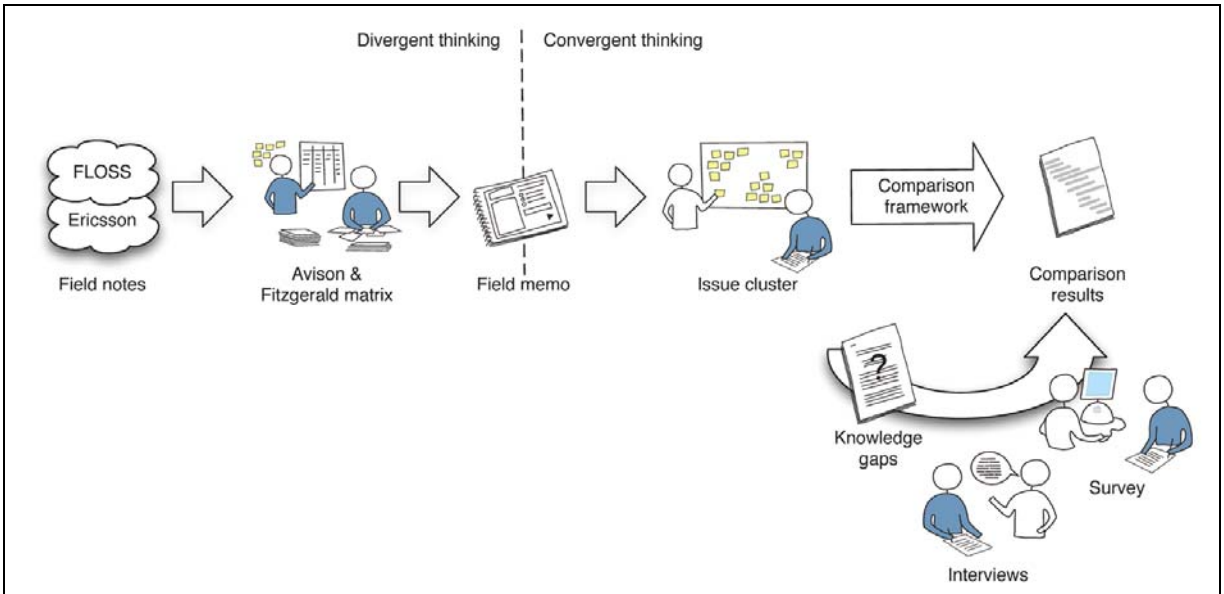

Figure 1. Comparison Method

Divergent thinking. As stated before, the purpose of the first steps of the comparison method was to elicit as much information as possible, rather than focusing on a particular predefined set of issues. This approach was chosen to encourage broader thinking and discussion about the methodologies. In the initial phase, many pages of notes were taken summarising interviews and documentation analysis. Topics covered were as broad as software engineering itself.

To extend the scope of these notes, two matrices were created: one containing information from Ericsson and the other containing information extracted from the FLOSS projects. The dimensions for the matrices were created by applying each assessment criteria (see Table 1) to each description point (see Table 2). For each intersection, a discussion among researchers was initiated. This discussion was based on initial field notes as well as further information gathering.

This procedure helped to generate a lot of discussions regarding the investigated methodologies from multiple viewpoints. It also helped in spotting several knowledge gaps previously overlooked during the initial study phase. Both these ideas and the knowledge gaps were captured in a field memo, a structured document that served as an instrument to articulate our findings.

Convergent thinking. In the next phase of the research, findings in the field memos were clustered. As noted above, we used the two Avison and Fitzgerald (1995) frameworks to generate ideas, but wanted a more organized and parsimonious presentation of the methodologies to organize the presentation of our results. To do the clustering, the findings were written on a number of sticky notes and all sticky notes were placed on a whiteboard without any special order. Collaboratively among all researchers, the sticky notes were then grouped by topic, generating issue clusters. From this process, a topic structure emerged, which helped us to spot both the relative size and relations

between issues. This topic structure became the foundation of the comparison framework we used when comparing the Streamline and FLOSS methodologies (see Appendix A). Given its origin, the final framework has some resemblance to the original Avison and Fitzgerald (1995), but with a structure that is more fitted to the two sets of methodologies being compared.

Once we had the clusters, we examined what we had learned about each methodology in each cluster. For clusters where we detected a lack of information, a set of questions was written to address knowledge gaps or points that could be clarified. These knowledge gaps were used to start another round of interviews, documentation analysis, and a survey. Unlike previous phases, this time, information gathering was highly directed, seeking answers to concrete questions instead of being open-ended. This process was continued until a final set of findings for each aspect of the comparison framework was obtained.

4. Selected Comparison Results and Analysis

In this section, we present an analysis of the differences between FLOSS methodologies and the Streamline methodology. Differences are analysed from several perspectives, according to the comparison method described in Section 3, using interviews, document analysis and a survey as input. The structure of this section follows the topics of the comparison framework that was developed, which is described in detail in Appendix A. In the main body of the paper, we present only a subset of the analysis—the different activities in the compared methodologies—that gives a flavour of the kinds of comparisons made. The full detailed comparison, covering background (history, objectives, principles, paradigm, usage), model, life cycle, rules and tools, and participatory aspects of the methodologies, is given in Appendix B. In this section, we cover design making, coordination and communication, requirements, planning and control, information availability and verification and validation. For each of these, we discuss how the activity is addressed in Streamline and in the studied FLOSS projects, as a basis for identifying possible adoption opportunities.

4.1. Decision Making

Decision-making is done across several levels in Ericsson Streamline projects. As stated before, Streamline has a principle concerning self-organising teams that is reflected in the decision-making processes. Implementation-related issues are fully handled by development teams, including the design of the solutions. However, a System Architecture Team (SAT) also participates in the decision-making to ensure that the overall project architecture is respected. Teams are also responsible for generating their own planning estimates, as discussed in Section 4.4. Even with the considerable number of decisions that are left to development teams, our survey results show that 62% of the respondents think that their participation in more decision-making would improve the decisions taken. One respondent puts it this way: “It’s important that the right people influence each decision. The trend is, all employees or no one, which is equally bad.”

In contrast, decision-making in FLOSS projects enables as much participation as possible. Concretely, decision-making in FLOSS projects revolves around the maintainers, and perhaps a “benevolent dictator” guiding the project. Fogel (2005) describes the benevolent dictator as the “final decision-making authority… who, by virtue of personality and experience, is expected to use it wisely.” The maintainer shares some of the benevolent dictator responsibilities but on a local scale, usually on one (or a few) software module(s) where he has the most experience. The maintainer is responsible for maintaining the module integrity and quality and for assuring the coherence of the module towards the whole system. There are several module maintainers but just one benevolent dictator for the FLOSS projects studied. How these roles exercise their decision-making power varies from project to project. However, there are two common denominators in all FLOSS projects:

- Decision-making power must be used with discretion, usually only when it is clear that the community will not reach an agreement by consensus.
- Decision-making must be public, transparent and explicit, and all members of the community should be able to participate if desired.

Discussions being public has several beneficial effects for the community. First, while the decision-making process can seem slow and cumbersome compared to Streamline, in the long run, decisions are usually more elaborated and incorporate more points of view. As a result, the community embraces decisions taken to a higher degree. Also, newcomers can learn about the development by just reading the discussion threads (both current and archived), which provide the rationale for the decisions. To avoid discussions on recurrent topics that have already been decided, FLOSS communities make extensive use of FAQ pages and mailing list archives. Additionally, since discussions are public and later archived, the participants tend to think twice before stating an opinion, as their community recognition is at stake. As an extreme case, in the Linux kernel project, there is a common agreed rule to avoid discussions about new features or possible bug fixes without attaching source code as proof of concept, and usually benchmarking data. This way pointless discussion is kept to a minimum. This reflects one fundamental difference between FLOSS and industry: "the fact that decision-making in Linux does occur but it happens after development, not before it" (Iannacci, 2003).

4.2. Coordination and Communication

In Streamline, small teams perform small (less than six weeks) implementation iterations on the project's baseline. We have already seen the considerable degree of decision-making power that these teams have to achieve their implementation goals. However, the specific goals to be achieved have to be prepared beforehand to avoid conflicts and/or overlapping work. By the use of anatomy plans on requirements, planning and a clear roadmap, implementation tasks are allocated so that teams can work concurrently in a coordinated way. A traversal SAT provides assistance to avoid issues from an architectural point of view. Progress is monitored by the means of burn-down charts. A set of procedures have been established to handle cases where a development team needs resources that cross project boundaries (such as approval to acquire some third-party product).

While the FLOSS structure looks simpler at first sight, the underlying communication interactions can be more complex. Iannacci (2005) describes the coordination and communication structures in the Linux kernel project as heterarchical. If we analyse FLOSS projects from a hierarchical point of view we will soon notice that there are rarely more than three levels: usually a heterogeneous contributor base, a group of maintainers or core team and the project leader or benevolent dictator. However, the project leader is the only one who stays at the top (with the focus on delegating tasks and responsibility), while the position of the rest may change depending on the specific interaction. If we analyse the typical evolution of developers in a FLOSS project, we will see that they sometimes act as a maintainer, while other times as a developer, reviewer or simply a user. As developers shift personal interests, the roles of maintainer, developer and reviewer are dynamically and temporarily assigned to the more credited and interested individuals. Thus, when observing the network of interactions as a whole, we can see a loosely coupled network where interactions are dynamically established and deprecated.

While Streamline's predictable and quite static organisational structure can appear more ordered and efficient, FLOSS communities extract several benefits from heterarchies. Firstly, by the means of the maintainer's system, tricky tasks are usually taken up by the best available developer, while easier tasks are left to more inexperienced ones, who in turn use them to gradually gain credit. This gain of credit is of key importance as it fuels the meritocracy system that keeps FLOSS communities going. Secondly, heterarchially distributed authorities naturally drive the development in several directions at once, which in FLOSS projects is usually considered a good thing. Finally, as opposed to industrial settings, where change (i.e., developer's rotation) is seen as a menace to project stability, FLOSS minimises risks by embracing continuous change and reducing the dependence on a single individual.

Despite all the discussed strengths, FLOSS project coordination structure depends on some preconditions in order to run smoothly. First, and most importantly, communication must be horizontal and direct among peers, without intermediaries. This way, peer relations can be easily established independently of team and project boundaries. It is also important to have appropriate tool support so

conversations are permanent and feed the knowledge base (i.e. mailing list archives, how-tos, FAQ) and contribution ownership can be tracked (for instance, FreeBSD uses the svn blame[5] command for this (Jørgensen, 2001). Additionally, three main types of information must be abundant:

- Technical know-how, so that the necessary learning process associated to frequent developer reallocation does not flood the communication channels.
- Behavioural guidelines, so that community members know what is expected from them and what they can expect from others.
- Clear conflict resolution procedures, which, even if used sporadically, tells the community members what to expect in a worst case scenario, for instance, when a discussion is taking too long and is blocking further development.

In contrast, Projects A and B at Ericsson also have a considerable documented knowledge base, but our conducted survey shows that its functionality is limited. 62% of the developers answered that the information is hard to retrieve, 41% that it is unclear or incomplete, and 34% that the information exchange tools are inadequate.

4.3. Requirements

In Streamline, requirements are handled in a traditional way. Strategic Product Managers gather and prioritise requirements from several sources (e.g., customers, marketing, internal). Technical Product Managers then divide the requirements into sets that can be implemented by a single team in a single iteration. To support this, a cross-functional board of System Managers and Node Architects provides technical insight and first effort estimates.

In contrast, the FLOSS projects studied do not have formal requirements development processes, but rather rely on developers choosing to work on new features. This approach works because developers of these projects are often also prominent users, which is not the case for Projects A and B. Streamline validates requirements by reviewing them with an expert board while FLOSS projects expose them to public discussion. For example, in JBoss, pre-requirement artefacts (feature requests) are exposed to the user base for validation and enrichment. Through the issue tracker, users can add new requirements and vote on existing ones. Then, requirements are allocated to development iterations in a similar fashion as Streamline does. Volunteer contributors can then pick tasks that they want to contribute to. This way, JBoss obtains a balance between community driven development and steering.

Requirement validation can go as far as requiring an actual working implementation before accepting a new feature ('code then decide'). Most notably in the Linux kernel community, discussions concerning new features are even silenced until a working implementation is attached. Then, unambiguous discussion can follow, working with actual source code and real world benchmarking results. Even then, Torvalds might not add the feature to the main source code tree because "nobody takes out features, they are stuck once they get in" (Iannacci, 2003). Real users must be using the feature from one of the development trees for it to be considered for the production release. This way features, are much more mature when released and experimentation is kept away from the stable kernel.

4.4. Planning and Control

In the Ericsson projects, the planning is performed in a formal way, where a specific amount of time has to be allocated for a given set of requirements. We consider three aspects of this process, first how work is estimated, second, how work is tracked and finally, how deadlines are enforced.
First, Streamline teams are responsible for generating their own planning estimates, considering their own experience and knowledge of the project. That said, in Project A, given its size, there is a project manager in charge of planning. However, the final estimations are still carried out by the development

[5] SVN: Subversion is a common version control system in the FLOSS community. See http://subversion.tigris.org

teams. The estimation in Streamline projects is done in two phases. First, a quick study is performed in order to receive an approximation of the effort needed to implement each requirement. After that, resources are allocated and the requirements are prioritised and assigned to the development teams. The second estimation phase is delegated to the assigned team members. They can then provide a more detailed estimation based on their own experiences and capabilities. One drawback with this approach is its rigidity. For example, emerging requirements can arise from development, e.g. the creation of a support domain specific tool. However, the deadline has already been established and it is too late to allocate resources for these activities, so they compete with the rest of the planned tasks. These activities happen below the radar, as 77% of the developers in projects A and B do not ask for permission and just 38% of them have enough time to complete them. Except for JBoss, the studied FLOSS projects do not perform an estimation activity. Instead, features are handled individually and no specific schedule is set by the project (of course, an individual developer may set his/her own personal schedule).

Second, Streamline provides a set of techniques for controlling the development progress of each team. The most important ones are the task board and the burn down chart. The task board contains three columns with the unstarted, ongoing, and finished tasks and is updated continuously. The burn down chart shows the tasks completed over time and provides a quick view of the development progress. When looking at the FLOSS projects, JBoss is again the only one that has some kind of progress monitor for the development. Even the external contributors are recommended to add their tasks to the issue tracker stating the expected completion dates and the dependencies with other tasks.

Finally, in Streamline, the deadlines are strict and a lot of effort is put on meeting them. In contrast, the FLOSS projects studied do not set specific deadlines for their developers. In fact, no matter how long a feature development takes, it will not be accepted until the expected quality level is met. As an example, the FreeBSD and Linux kernel pre-commit reviewing phases enforce this. Thus, a code patch has to be fixed as many times as it takes to pass this phase. An exception to this behaviour is JBoss. There, an estimation phase is done in order to split the requirements among the development teams (composed by hired core developers and external contributors) to meet the delivery deadline. Nevertheless, they still perform a parallel reviewing phase where the possible shortcuts are detected and fixed so that the required code quality can be ensured.

The short iteration approach and the pressure to meet deadlines can create problems for the development. Specifically, as mentioned in Appendix B.1.5, this pressure can encourage shortcuts concerning code maintenance that lead to accumulation of technical debt, that is, a backlog of deferred technical problems. Design shortcuts made during one iteration to achieve the expected functionality cannot be fixed in the next because new requirements and deadlines pop up. Streamline makes use of a team called Design Follow Up that is supposed to fix these kinds of shortcuts. However their duties also include fixing Trouble Reports, which are prioritised over code clean up. An employee working in Project A confirmed this problem: “we should educate and encourage developers to refactor code they modify in line of duty, [if not we will] let the code deteriorate and the technical debt increases.”

4.5. Information Availability

A significant area of difference between Streamline and FLOSS development concerns the public availability of information about the development. In industry, software artefacts are meant not to be visible until they are ‘finished’. In industry, this is usually enforced by some quality checks. In Streamline, these checks are called Quality Doors and are placed at key iteration points. Obviously, industry projects have very good reasons to keep information locked in-house (e.g., for intellectual property protection), but these limits also often stop engineers from working effectively on different projects in the same company. As many as 63% of the Ericsson survey respondents agree that information should be easier to retrieve and 38% that they do not receive all information they need. Also, there is close to zero visibility between projects, effectively blocking component reuse and collaboration between projects.

As is discussed in Section B.1.3, information openness is at the very core of the FLOSS way of working. While FLOSS projects' decidedly open attitude towards information can seem a bit extreme, it is of key importance to a project's success. In FLOSS projects, artefacts are publicly exposed (not only source code) from the day they are created. FLOSS projects' 'publish immediately policy' allows for a kind of informal lazy review process. For instance, anybody can enrich a feature request on the issue tracker, provide insight to a bug discussion or propose a modification on an already submitted source code patch, likely increasing the number of contributions that help in enhancing the quality. Outside the projects, this open door policy is also very useful. A potential user can easily access a lot of information about the on-going development, making it easy to evaluate the quality of the product, future plans, open bugs, etc. It is also usually just a matter of minutes to get a working copy of a FLOSS project along with documentation. This ease of evaluation facilitates the fact that FLOSS projects many times reuse as much as they can from other projects. For large software companies to be able to really reuse their software assets, the effort investment needed to find and evaluate component candidates must be as low as possible.

4.6. Verification and Validation

Verification and validation in both FLOSS and Ericsson projects is composed by two main phases. First, in Streamline a pair-programming approach is recommended, so developing and reviewing is done at the same time. In FLOSS, a pre-commit code review is often done, where faults are detected by reading the source code. The second validation phase concerns automated testing. Both FLOSS and Ericsson projects use techniques, like unit testing and regression testing, which are automated to a large extent. However, in projects like FreeBSD or the Linux kernel this automation is more difficult to handle than in JBoss and Ericsson projects as it is hard to test a complete operating system without human intervention. Therefore, these projects need to rely on a stricter reviewing phase before testing commences. This relation is also described further by Rigby and Germán (2006).

When looking at software verification and validation activities in FLOSS projects, we find an important characteristic that differs from Streamline. In industry, testing traditionally has been seen as a less qualified and simpler task (although during recent years, testing has gained first class citizen status among developers). However, a stigma persists which affects the motivation for performing that job. In Streamline in particular, testers are just responsible for finding bugs, leaving the bug fixing for developers. Developers get the recognition, and in turn, complain that testers just spot problems without contributing to the solution. In FLOSS projects, the task of finding a bug is often followed by the fixing of that same bug by the same person, and is rewarded by recognition by the community. Thus, not only fixing bugs, but also finding them turns to be a motivator for FLOSS developers (Jørgensen, 2001).

5. Adoption Opportunities

This section describes the main outcome of this study, i.e. a set of FLOSS practice adoption opportunities for software firms to consider. For each adoption opportunity, we analyse its expected advantages as well as its implementation requirements. These adoption opportunities were distilled from the comparison presented in the previous section and in Appendix B. With the final comparison done, a final analysis was done where detected weaknesses of Streamline were matched with relevant strengths of the FLOSS methodology. Where such a match was found, the knowledge gained though the comparison phase helped isolate the essence of the practice to see if it would be a candidate for an adoption opportunity. These candidates where later validated as discussed in Section 6.

Table 4 is provided for traceability. The columns of the table correspond to the different adoption opportunities described in this section, while the rows refer to the comparison results and the corresponding section or appendix where they were described. An X in the cell indicates that evidence from that section of comparison guided development of the particular adoption opportunity. In the remainder of this section, we describe each adoption opportunity in more detail.

Table 4. Which Comparison Result Topics Influenced Which Adoption Opportunity

	RTD[a]	DEP[b]	IAV[c]	ACDM[d]	WoW[e]	ATS[f]
Background, App. B.1						
History, App. B.1.1						
Objectives, App. B.1.2						
Principles, App. B.1.3	X			X	X	X
Paradigm, App. B.1.4					X	
Usage, App. B.1.5	X		X			
Model, App. B.2		X	X		X	
Life cycle, App. B.3	X					
Rules and tools, App. B.4			X	X		
Participation, App. B.5	X	X	X	X	X	X
Activities. 4					X	
Decision-making. 4.1		X	X	X	X	
Coordination and communication. 4.2		X		X	X	
Requirements. 4.3			X		X	X
Planning and control. 4.4	X				X	X
Information availability. 4.5	X		X	X	X	X
Verification and validation. 4.6	X	X			X	

Note:
[a] Reduce technical debt
[b] Define an entry path for newcomers
[c] Increase information availability and visibility
[d] Embrace asynchronous tools for communication and decision-making
[e] Let practitioners influence ways of working
[f] Allow task selection

5.1. Reduce Technical Debt

The first FLOSS practice adoption opportunity identified is a set of strategies to reduce technical debt. Commercial development is unavoidably forced to prioritize particular development tasks to meet promised deadlines. However, it is important not to let the urgent displace the important. With an iterative approach in particular, which places a deadline every few weeks, quick solutions often replace elegant ones as technical debt accumulates, the architecture deteriorates and the code becomes harder to maintain and bugs become more expensive to fix (see Appendices B.1.5 and B.3). Reduction of the technical debt will pay off in terms of maintainability. In addition, new developers can be introduced faster to the current code base if it is easier to understand (see Appendix B.5).

In general, two strategies can be discerned when dealing with technical debt. First, situations that create technical debt need to be kept at a minimum. Technical debt, as bugs, is cheaper to eliminate nearer to the creation point. It is important to build developers' consensus on the minimum quality of the source code additions made to the code base. While code reviews are very useful to reveal design shortcuts, it is also expensive. We can learn from FLOSS projects that simply having public exposure to a broader audience is a cheap and very effective way to invoke some benefits of code reviews, thus filtering shortcuts that would otherwise pass unnoticed (see Appendices B.3 and 4.5–4.6).

Second, plan in advance for technical debt. No matter how much effort is put on avoiding it, some amount will always accumulate so there needs to be a pre-emptive allocation of efforts dedicated to reduce technical debt, e.g., code clean-up and removal of design shortcuts. While the relative size of

this activity will be small compared to the main development effort, it needs to be isolated from release deadline stress. Having allocated resources would certainly help in decreasing the technical debt. For example, when one has new personnel involved in a project, they could be assigned initially to cleaning up code and fixing minor issues.

This sort of initial assignment comes naturally in a FLOSS project, as will be discussed below. The main strength that FLOSS projects have in reducing technical debt is that they publicly review (expose) all code submissions to the repository. A submission may therefore turn into a discussion back and forth between the committer and the core team, with the results that successful code submissions tend to be polished and there is lower initial technical debt.

5.2. Define an Entry Path for Newcomers

A second finding is that it is important to have a predefined path that allows new developers to learn while doing productive activities (see Section 4.2). If this issue is left unattended, there is a risk of placing newcomers in positions for which they are unqualified or making their learning curve unnecessarily long. With proper support from experienced developers, bug fixing and technical debt reducing activities are a good entry point for new developers. Such tasks allow new developers to familiarise themselves with the software architecture, perform tasks with different difficulty levels and to be productive from the start. Following this strategy, they would be ready to be incorporated sooner in regular development project activities. Additionally, resourceful developers would have a greater chance to stand out sooner, reducing employee frustration derived from not being able to deliver their full potential (see Appendix B.5 and Section 3.6).

5.3. Increase Information Availability and Visibility

Third, a common problem in large software development organisations is that knowledge sharing between departments often happens only at a managerial level. Often, the same component or tool is developed in more than one department without realisation that redundant work is being performed. Such problems can be avoided with greater information visibility.

The first and most important step in order to achieve information visibility is that it should be easy to locate. A good automated tool must index information sources and allow for centralised searching (see Appendix B.4). The whole software source code asset of an organisation should also be searchable. It is especially important for technical roles to have read access to other projects' resources. This would make reusability possible, not only for software components but also concerning technical know-how. This public exposure would also favour modularised designs, allow for external contributions and increase overall code quality, as is so often the case in FLOSS projects (see Appendices B.2, B.5 and Section 4.5 in this regard).

In order to cope with intellectual property (IP) issues that limit the exposure of particular software, a black-list approach is proposed, where each department explicitly locks IP sensitive assets. This approach can be more beneficial and efficient in the long-run than a white-list approach, where all assets are locked by default and sharing an element requires an explicit and likely time consuming approval process (see Appendix B.1.5 and Section 4.5 regarding the handling of IP).

Additionally, there are a number of documents that would benefit from being created and edited online. When a documentation artefact becomes an online object, interested engineers can observe and subscribe to it before it is finished, providing feedback during its whole evolution. While it may seem counterintuitive to publish information before it is complete, FLOSS projects have shown that with proper tool support, the quality of the final artefact increases (see Section 4.5). Simply put, people deal better with incomplete information rather than no information at all.

5.4. Embrace Asynchronous Tools for Communication and Decision-Making

Fourth, FLOSS development practices suggest the value of asynchronous communications tools. It is a common software industry practice to use tools to automate some development tasks as much as

possible. However, communication is usually overlooked in this regard. Agile methodologies themselves embrace face-to-face conversation as the most effective method for conveying information[6] (see Appendix B.1.3): a meeting can be the best way to get something decided within a given deadline when desired participants are clearly identified. However, meetings also have serious drawbacks, in particular with regards to scalability. Meetings are time-consuming, fully blocking participant attention and ability to multitask. Many people can be interested in the results of a meeting but, usually the only outputs are summarised minutes in which the conversational context is lost. Also, a meeting can be poorly prepared, with uninterested attendants invited and/or interested parties left outside.

Asynchronous communication technologies allow an individual to go through a larger amount of information without blocking their attention at any given time (Appendix B.4 contains extensive discussions on the characteristics of tools used by FLOSS communities). Concurrent tasks that feed the conversation (like further information processing or gathering) can happen without having to reschedule a meeting. Additionally, a larger number of individuals, independently of location or time zone, can observe the conversation with full conversational context available. If it concerns on-going decision-making, it can not only be improved with feedback, but also drive interested parties to a bigger endorsement of taken decisions because they can assimilate the rationale behind them (see Section 4.1).

Most importantly, communication conducted by electronic means can be automatically archived without loss of information. Especially design and implementation decisions could benefit from being stored for later reference and retrieval. This archive would form a useful knowledge base that can be used to lower the learning curve for newcomers and ground further decision-making for experienced developers. Moreover, this knowledge would be permanent and independent of key employees leaving a project (see Sections B.5, 4.2 and 4.5 for discussions on the benefits of having a persistent knowledge base).

5.5. Let Developers Influence Ways of Working

Fifth, it is obvious that when executing a methodology, developers are the ones who see the processes, methods and tools from a closer perspective. While methodology creators have to deal with the whole organisation, developers are aware of local project realities. In FLOSS, developers have a great deal of autonomy is choosing processes, methods and tools. In contrast, developers in Streamline expressed some frustrations with one size fits all approaches. Streamline might benefit from additional input from individual developers. For example, a messaging board where anyone could propose improvements or state problems with the methodology is a simple method for management to know the practitioners' opinions. Adding a voting system for possible methodology changes could help to prioritise the modifications needed. This would drive the methodology to drop excessive weight in favour of the simplicity principle, as exposed in Appendix B.1.3, which can be seen in almost all activities in the FLOSS communities (see also Section 4.1).

5.6. Allow Task Selection

The final adoption opportunity is allowing some degree of self-assignment of tasks. Working on the same project, performing the same tasks over and over again, might decrease employee motivation and productivity. Curiosity and self-development are good motivators. Providing a way to channel these motives can be useful to keeping employee commitment high. Appendix B.5 contains collected data that clearly sustains this claim both in FLOSS and industry projects. One approach is to let employees dedicate part of their weekly working hours to a pool of diverse available tasks that cross project and department boundaries. However, this process needs to be implemented carefully, since:

(1) It is important that there are always engineers active in a project at all times, because contributors will need support.

[6] http://agilemanifesto.org

(2) There needs to be a permanent knowledge base that outside contributors can look-up without disturbing engineers with frequently asked questions. If the learning curve is too high, contributors will simply turn to other projects. (See Sections 5.3 and 5.4.)

(3) The task pool must contain activities of a size that are feasible to conduct during an employee's 'free time'. Additionally, to avoid rework, it also must reflect when somebody is already working on a task. Furthermore, it should be possible for a developer to propose a new task, as an outsider's view can be a source of very creative ideas (see Sections 4.3 and 4.5). A resident mentor who helps new contributors, as in the FreeBSD project, might lower the learning curve and ensure that contributions meet the locally required quality standards.

This kind of strategy could place significant stress on software development, e.g., if developers tend to avoid certain tedious tasks or if the development team does not posses varying expertise that naturally divides the tasks between them (Beck and Fowler, 2000). However, this stress could help develop ways of working to a state that is more efficient for resident employees too. For instance, a constant flow of contributors from outside the particular project might help spot issues that unnecessarily raises the bar concerning the learning process and which, consequently, residents have learned to avoid (see Appendix B.1.3 for examples on how the simplicity principle works in FLOSS communities).

6. Static and Dynamic Validation

In the previous section, we described a set of six adoption opportunities, that is, elements of FLOSS development practice that might be transferred to industry practice. In this section, we discuss our efforts to validate these findings through interaction with industry partners. By collecting experiences from many projects over the years, the authors' research group has developed a research and technology transfer model (Gorschek et al., 2006) for identifying industry-relevant research issues. The model is shown in Figure 2. Formulating the problems and devising candidate solutions is done in close collaboration with industry participants to ensure relevance and scalability. The ultimate test of research relevance and applicability is done through several validation steps, all used for generating input to improve the research solutions proposed. The final step in this research cycle would be the live piloting and release of the solution in practice. For the current study, Steps 1–3 in Figure 2 correspond to the study described above, and publication and presentation of our results, to Step 4. In this sections we will cover Step 5 of the study in more detail, while later presenting Step 6, which is still on-going.

6.1. Static Validation of Findings

By static validation of findings, we mean independent review of the findings without putting them in practice. Static validation of our findings (Step 5) was conducted through a series of workshops that were held at three different large-sized multinational corporations that are currently using agile methodologies in their projects.

The first corporation was Ericsson AB, where the case study had been conducted (Company A). The second company, based in Sweden, is part of an international conglomerate, with headquarters in Switzerland, focusing on software and hardware development for the space industry (Company B). The third, and final company, located in Sweden, is part of a Swiss-Swedish multinational corporation that focuses on power and automation technology (Company C). All workshops were held in the spring 2010. Table 5 provides an overview of the participants and the number of workshops that were held. The second column in Table 5 provides the total number of workshops given. The third, fourth and fifth columns covers the number of participants categorised in upper, mid, lower management and experts; mid-management is usually responsible for complete product portfolios, low-management is involved in traditional project management activities, while experts are focusing on specific issues dealing with methodologies at each company. The final column provides the total number of participants in each company.

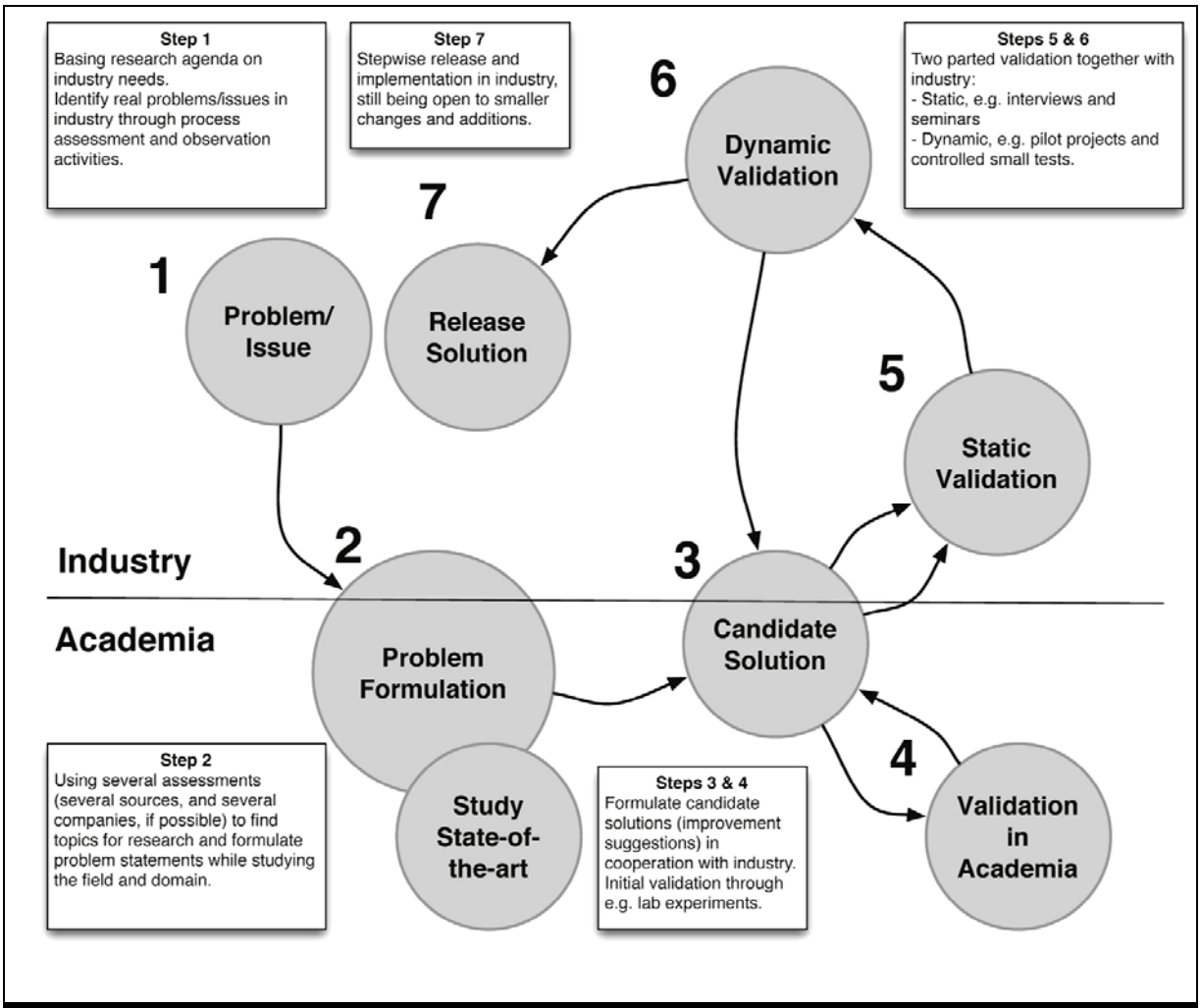

Figure 2. Validation Process for Research Findings (Gorschek et al., 2006)

Table 5. Descriptive Statistics for the Workshops for the Static Validation

	# Work–shops	Upper mgmt.	Mid mgmt.	Lower mgmt.	Experts	# partici–pants
Company *A*	3	2	4	6	2	14
Company *B*	1	1	1	3	1	6
Company *C*	2	2	4	3	3	12
					Σ:	32

At each workshop, an introductory presentation was given including the background, aim, purpose and results from this study. The results, in the form of adoption opportunities (see Section 5) were covered in detail. Participants then discussed pros and cons of introducing these innovations into their organisations. As a starting point, to allow discussion to quickly focus on the most relevant issues, cumulative voting, or the $100 method as it is sometimes referred (Leffingwell and Widrig, 1999), was performed on the six adoption opportunities. The idea with cumulative voting is to provide input to discussions by forcing the participants to rank issues or statements by spending $100 on each category of issues or statements. In our particular case we offered each participant six adoption opportunities that they then spent $100 on. The participants could spend everything from $0 to $100 on an adoption opportunity (obviously, if they spent all $100 on one adoption opportunity then no

other adoption opportunity would receive any money). The results from the voting led to three adoption opportunities having the majority of the votes, while the other three were deemed to be less important. In Table 6, the ranking of each adoption opportunity (in each company) is listed. The letters in the table stands for: a = Reduce technical debt, b = Define an entry path for newcomers, c = Increase information availability and visibility, d = Embrace asynchronous tools for communication, e = Let developers influence ways of working, f = Allow task selection. Note that there was a good degree of agreement about the highest and lowest ranked opportunities, suggesting the generalisability of these results.

Table 6. The Ranking of the Six Adoption Opportunities

	1st	2nd	3rd	Runners-up
Company *A*	*b*	*c*	*e*	*a,d,f*
Company *B*	*c*	*e*	*d*	*f,b,a*
Company *C*	*b*	*c*	*d*	*e,f,a*

While all participants agreed on the potential benefits of all six adoption opportunities, some implementation roadblocks were identified at all three companies. These were classified into three main groups: economy of scale, localism, and departmental protectionism. These three sets of problems were prime issues for the ranking of the adoption opportunities and will be discussed further.

First, some benefits of FLOSS practices do no really show until there is a critical mass of practitioners. Practices like implementing an entry path for newcomers, building a permanent knowledge base or allow early feedback on documentation artefacts require some economy of scale to truly show its potential. Pilot projects, which are by definition done at a small scale, can mask or even prevent the delivery of real benefits.

Second, a tradition of collocated development teams, meeting-based culture and agile methodologies, which embrace face-to-face meetings, makes it difficult for practitioners to voluntarily use asynchronous tools for communication and decision-making. When the teams are collocated, it is much easier to setup a stand-up meeting for quick resolution of an issue. However, when international teams have to interact, it will be much easier for practitioners to see the benefit of a permanent knowledge base of past decisions and know-how.

Finally, it is a common practice at big corporate settings to setup departments so there is a certain level of competition between them. This fosters efficiency but has the side effect of managers becoming protective of their own department assets, in some cases affecting the efficiency of the organisation as a whole. This practice may lead to a certain degree of resistance to practices based on sharing or public exposure of artefacts. Upper management commitment is necessary to drive a cultural change, so that department reputation is not based on the value of locked assets but on the quality of the final output.

6.2. Dynamic Validation of Findings

By dynamic validation (Step 6 in Figure 2), we mean testing of the research findings in a real implementation. All three companies are currently evaluating the usefulness of some of the adoption opportunities (especially the ones they ranked the highest) and have begun to move forward on implementation. Specifically, Ericsson has introduced a workgroup to develop a set of guidelines that define an entry path for newcomers, introduced a new asynchronous collaboration tool, is looking into how employees will be able to influence ways of working to a larger extent and has restarted the work on controlling technical debt (Tomaszewski et al., 2008). Company B is currently working on increasing information availability and visibility through the usage of a wiki and setting up workgroups to investigate possibilities to increase the opportunities for practitioners to influence ways of working (and allowing for task selection to a larger extent). Company C follows the same approach as

Ericsson concerning the development of guidelines for newcomers, in addition to consolidating and opening up information sources inside the company and investigating the issue of technical debt in some of their products.

Obviously, the end goal of the above activities is that, after having refined the solutions to the adoption opportunities in their particular settings, they will finally be released in live projects (Step 7 in Figure 2).

7. Threats to Validity

This section will present the threats affecting the validity of this research. We will follow the recommendations from (Creswell, 2008) for addressing validity issues. In the following we will discuss possible internal and external validity threats as well as reliability issues, and the methods implemented to minimize these threats. We identified three threats to internal validity: the possible inaccuracy of the information found, bias of the information with respect to reality, and the epistemological assumptions caused by our judgment.

To assure the accuracy of the information collected, whenever possible we have triangulated findings, contrasting several information sources and diverse information gathering methods. At Ericsson, three different methods have been used: project documentation review, interviewing key personnel, and a survey. The information found was aggregated using field memos to find convergence. To reduce incorrect interpretations of the data collected at Ericsson, periodic checks were performed. Two representative members of both studied projects reviewed the data collection and analysis to assure its correctness. For FLOSS projects, information found in project documentation was contrasted with published research. However, it must be acknowledged that reliance on different kinds of data for the two kinds of projects may introduce some discrepancies.

The second threat to internal validity has been addressed in a similar way. To ensure that data collected at Ericsson reflected the reality of the methodology as actually executed, we performed thirteen interviews (six per project and one Ericsson-wide role) and one survey. Having a broad scope (concerning roles), when selecting the respondents, allowed for the identification of valuable and trustworthy data. For FLOSS data, we are reliant on the efforts of previous researchers to ensure that their published work accurate describes FLOSS practices.

Finally, to minimise the information bias introduced by our judgement, we worked as closely as possible with the practitioners as recommended by Creswell (2008). Another valuable source in this respect was the survey results, which confirmed and also refuted several assumptions from our previous research. A limitation of this approach is that we were not able to survey FLOSS developers in a similar way.

External validity addresses issues that may affect the generalisability of the findings. In this study three threats have been taken into account: the adequateness of the selected project targets, the consequences of the appropriateness of the data analysis, and the generalisability of our findings.

As noted above, the selection of the FLOSS target projects was performed considering two main prerequisites. First, they needed to resemble the available Ericsson projects to obtain an equitable comparison. Second, they had to be successful projects with a considerable user and community base to provide valuable points of differences. To ensure a better representativeness of the FLOSS world, we chose well-known projects with different levels of industry involvement. However, it is important to note the delimitations in generalisability for this study. As stated previously, this study is a case study and it does not try to produce conclusions that can be applicable to the whole FLOSS universe as only a few projects have been selected in this research. This limitation was appropriate, as the study focuses on producing adoption opportunities that suit this study's particular context.

The usage of a good data analysis method, which ensured the consideration of as many methodology issues as possible, was also important to ensure that results could be generalised to other

development settings or FLOSS projects. To address this concern, we designed a method that leverages a formal comparison framework (Avison and Fitzgerald, 1995) and adapted it to FLOSS methodology characteristics while keeping industrial practice input at the principle level. Although the comparison phase was performed using specific information from the Ericsson projects, the generated adoption opportunities should be applicable to other large software development companies that use agile methodologies in large projects. It was also important to ensure that the adoption opportunities were extracting FLOSS best practices as opposite to common FLOSS traits that may or may not contribute to success. As stated before, to ensure this, the analysis of the FLOSS projects is based on a number of previous research efforts.

Finally, whenever possible, FLOSS practices and traits, as identified in the studied projects, have been contrasted with previous research targeting wider FLOSS project samples. However, it is outside the scope of this paper to generate findings that represent the whole FLOSS universe and that can be generalised to any kind of industrial development setting. What is important to remember in this case is that the methodology we used, and the framework we developed, can be applied to any industrial setting with the aim of introducing FLOSS practices, techniques and methods. Furthermore, the indications from the static validation showed that the results were of general interest for various companies developing software in quite disparate domains, leading to all participating companies entering a dynamic validation of several adoption opportunities (see Section 6).

8. Conclusions and Future Work

This research has shown that, while being very different, bridges can be built between industry and FLOSS projects so best practices can be imported. By implementing our adoption opportunities, we expect an increase in development productivity as well as better adaptability to today's geographically distributed development projects and highly competitive markets. The results from this study have, through the validation phase, shown a certain degree of generalisability, and consists of six adoption opportunities: reduce technical debt, define an entry path for newcomers, increase information availability and visibility, embrace asynchronous tools for communication, let developers influence ways of working, and allow task selection.

In addition, we have provided a basis for future studies in this direction by developing a comparison framework especially tailored to spot differences between FLOSS projects and any other development methodology. Leveraging this framework, FLOSS practices, techniques and methods can be further explored, revealing further adoption opportunities when compared with other development methodologies used in industry.

Future lines of research could design a concrete implementation strategy to introduce suggested practices in a specific industrial setting. This should be done taking into account the risks identified in the validation phase and the experience drawn from the dynamic validation already performed in industry. We believe that the introduction of openness concepts in particular can be viewed as a challenge to established corporate culture and, hence, must be implemented with care.

References

Amor, J. J., Robles, G., González-Barahona J. M., and Herraiz, I. (2005) "From pigs to stripes: A travel through Debian." in DebConf5 (Debian Annual Developers Meeting).

Anonymous, (2009a) "Linux kernel maintainers file." http://lxr.linux.no/linux/MAINTAINERS.

Anonymous, (2009b) "JBossWiki." http://www.jboss.org/community/wiki/Main.

Avison, D. E., and Fitzgerald, G. (1995) "Information systems development: Methodologies, techniques and tools." New York, NY: McGraw-Hill.

Avison, D. E., and Fitzgerald, G. (2003) "Where now for development methodologies?" Communications of the ACM, 46(1), 78–82.

Avison, D.E. and Taylor, V. (1997) "Information systems development methodologies: A classification according to problem situation". Journal of Information Technology. 12(1), 73–81.

Beck, K. and Fowler, M. (2000) Planning Extreme Programming (1st ed.). Boston, MA: Addison-Wesley Longman.

Brooks, Jr., F. P. (1975) "The mythical man month and other essays on software engineering." Boston, MA: Addison-Wesley Longman.

Conklin, M., Howison, J., and Crowston, K. (2005) "Collaboration using OSSmole: A repository of FLOSS data and analyses." in MSR '05: Proceedings of the 2005 International Workshop on Mining Software Repositories, 1–5.

Creswell, J. W. (2008) "Research design: Qualitative, quantitative, and mixed methods approaches." New York, NY: Sage Publications.

Daffara, C. (2009) "Free/Libre open source software: A guide for SMEs." http://guide.conecta.it.

Damm, L.-O., Lundberg, L., and Wohlin, C. (2006) "Faults-slip-through—A concept for measuring the efficiency of the test process." Software Process: Improvement and Practice, 11(1), 47–59.

Davis, G. B. (1982) "Strategies for information requirements determination". IBM Systems Journal, 21(1), 4–30. DOI 10.1147/sj.211.0004.

Dinkelacker, J., Garg, P. K., Miller, R., and Nelson, D. (2002) "Progressive open source." in ICSE 2002: Proceedings of the 22nd International Conference on Software Engineering, 177–184.

Eisenhardt, K. M. (1989) "Building theories from case study research." The Academy of Management Review, 14(4), 532–550.

Fogel, K. (2005) "Producing open source software: How to run a successful free software project." Sebastopol, CA: O'Reilly Media.

Gacek, C. and Arief, B. (2004) "The many meanings of open source." IEEE Software, 21(1), 34–40.

Germán, D. M. (2003) "The GNOME project: A case study of open source, global software development," Software Process: Improvement and Practice, 8(4), 201–215.

Glass, L. (2003) "A sociopolitical look at open source." Communications of the ACM, 46(11), 21–23.

González-Barahona, J. M., Ortuño Pérez, M. A., de las Heras Quirós, P., Centeno González J., and Matellán Olivera, V. (2001) "Counting potatoes: The size of Debian 2.2." Upgrade, 2(6), 60–66.

Gorschek, T., Garre, P., Larsson, S., and Wohlin, C. (2006) "A model for technology transfer in practice." IEEE Software, 23(6), 88–95.

Guimarães, L. R, and Souza Vilela, P. R. (2005) "Comparing software development models using CDM." in SIGITE '05: Proceedings of the 6th Conference on Information Technology Education, 339–347.

Herraiz, I., Gonzalez-Barahona, J. M., and Robles, G. (2007) "Forecasting the number of changes in Eclipse using time series analysis." in MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories, 32–.

Hove, S. E., and Anda, B. (2005) "Experiences from conducting semi-structured interviews in empirical software engineering research." in METRICS '05: Proceedings of the 11th IEEE International Software Metrics Symposium, 23–.

Hurson. T. (2007). Think Better: An Innovator's Guide to Productive Thinking. New York, NY: McGraw-Hill,

Iannacci, F. (2003) "The Linux managing model." First Monday, 8(12).

Iannacci, F. (2005) "Coordination processes in open source software development: The Linux case study." Emergence: Complexity and Organization, 7(2), 21–31.

Jørgensen, N. (2001) "Putting it all in the trunk: Incremental software development in the FreeBSD open source project." Information Systems Journal, 11(4), 321–336.
Kroah-Hartman, G., Corbet, J., and McPherson, A. (2008) "How fast it is going, who is doing it, what they are doing, and who is sponsoring it." http://www.linuxfoundation.org/publications/linuxkerneldevelopment.php.
Lattemann, C., and Stieglitz, S. (2005) "Framework for governance in open source communities." in HICSS '05: Proceedings of the Proceedings of the 38th Annual Hawaii International Conference on System Sciences, 192-201.
Leffingwell, D., and Widrig, D. (1999) "Managing software requirements: A unified approach." Boston, MA: Addison-Wesley Longman.
Li, P. L., Herbsleb, J., and Shaw, M. (2005) "Forecasting field defect rates using a combined time-based and metrics-based approach: A case study of OpenBSD." in ISSRE '05: Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering, 193–202.
Martin, P. Y., and Turner, B. A. (1986) "Grounded theory and organizational research," The Journal of Applied Behavioral Science, 22, 141–157.
Paulson, J. W., Succi, G., and Eberlein, A. (2004) "An empirical study of open-source and closed-source software products." IEEE Transactions on Software Engineering, 30(4), 246–256.
Raymond, E. S. (2001) "The cathedral and the bazaar: Musings on Linux and open source by an accidental revolutionary." Sebastopol, CA: O'Reilly Media.
Rigby, P. C., and Germán, D. M. (2006) "A preliminary examination of code review processes in open source projects." Tech. Rep. DCS-305-IR, University of Victoria, Victoria, BC, Canada.
Robles, G., Scheider, H., Tretkowski, I., and Weber, N. (2001) "Who is doing it? A research on libre software developers." http://ig.cs.tu-berlin.de/oldstatic/s2001/ir2/ergebnisse/OSE-study.pdf.
Scacchi, W. (2002) "Understanding the requirements for developing open source software systems." IEE Proceedings Software, 149(1), 24–39.
Song, X., and Osterweil, L. J. (1994) "Experience with an approach to comparing software design methodologies." IEEE Transactions on Software Engineering, 20(5), 364–384.
Squire, M., Crowston, K., and Howison, J. (2009) "Flossmole." http://www.flossmole.org.
Tomaszewski, P., Berander, P., and Damm, L.-O. (2008) "From traditional to streamline development — Opportunities and challenges." Software Process: Improvement and Practice, 13(2), 195–212.
Torkar, R., and Mankefors, S. (2003) "A survey on testing and reuse." in SWSTE '03: Proceedings of the IEEE International Conference on Software-Science, Technology & Engineering, 164-173.
Warsta, J., and Abrahamsson, P. (2003) "Is open source software development essentially an agile method?," in Proceedings of the 3rd Workshop on Open Source Software Engineering, 143–147.

Acknowledgements

We would like to thank Ericsson AB, and the other two companies participating in the validation for providing the means that made this study possible. We are especially grateful to all the people at Ericsson AB who made time for us in their busy schedules and who provided a wonderful work environment.

We also acknowledge Dr. Oscar Dieste, Universidad Politécnica de Madrid, and Dr. Barbara Russo, Free University Bozen/Bolzano, who provided valuable feedback during the course of this study. Furthermore, Prof. Kevin Crowston and editors at IMD provided feedback and polished this paper in its final stages.

Appendices

Appendix A.

The following table of contents describes the comparison framework we developed that covers the important characteristics of a development methodology. The framework was created by clustering information we collected about the two development methodologies, as described in Section 3. As we used Avison and Fitzgerald's (1995) frameworks to guide the information collection, there is some resemblance in the topics covered, though with additions and different arrangement. The framework contains six main topics, as shown below.

1. Background
   (a) History
   (b) Objectives
   (c) Principles
   (d) Paradigm
   (e) Usage
2. Model
3. Life cycle
4. Rules and tools
5. Participation
6. Activities
   (a) Decision making
   (b) Coordination and communication
   (c) Requirements
   (d) Planning and control
   (e) Information availability
   (f) Verification and validation

This framework is the first contribution of this paper. It can be used as a way to describe a development methodology from its origins to its practical application, and also as a guide to compare two or more development methodologies. This comparison can be used to choose the methodology that better addresses each topic, or as a way to mix the strong points of each one developing a new and presumably better methodology. In the study presented in this paper, the framework was used to identify the most significant differences between Ericsson's Streamline and FLOSS methodologies. The most interesting differences for the current paper are points where FLOSS is stronger and can provide a possible improvement for Streamline. In the remainder of this Appendix, each topic is explained in more detail.

A.1. Background

This section contains five different topics that, together, define the basis of a methodology. Analysing the background of a methodology is crucial to understanding its assumptions and fundamental beliefs. When comparing two methodologies, comparison of these background features can reveal fundamental points of disagreement between them. Clearly identifying these issues is of key importance to the final objective of the comparison, that is, to adopt best practices from one methodology to the other.

A.1.1. History

This section describes where the methodology comes from. Having a historical context is important to understand the driving forces and needs that shaped the methodology during its initial stages. Other interesting points to consider here, for example, is whether it was created to substitute an existent methodology or to improve it. Moreover, no methodology is created from vacuum. All methodologies have roots in other methodologies that should be mentioned as well.

A.1.2. Objectives

When talking about the objectives of a methodology, we are referring to the aims their creators had or the needs it was meant to solve. For example, some methodologies have as a main objective to reduce, as much as possible, the product's time to market, while others intend mainly to improve software quality. Also, even if high-level objectives are the same, methodologies often disagree on the means to achieve them. For example, while two methodologies may focus on improving software quality, one may insist on a more extensive design phase and another in continuous feedback. Usually, methodologies are thought to improve not only one, but many aspects of the software development. All these objectives should be mentioned in this section.

A.1.3. Principles

The principles are one of the most important things to consider when studying a development methodology. Discovering the beliefs a methodology relies on provides a better understanding of its rules or recommendations. It would also help the user to identify the essential activities that should take place when adopting it, and to clarify the reason behind all that activities. As examples of methodology principles, agile methodologies rely on small team development and continuous communication between development stages. FLOSS methodology, on the other hand, believes in information openness and short release time.

A.1.4. Paradigm

At an even higher level than the principles, we find the paradigm the methodology is enclosed in. The paradigm discussion can be exercised from a multitude of viewpoints. In this thesis, we have adhered to the viewpoints proposed by Avison and Fitzgerald (1995), discussing the ontology (realism or nominalism) and science or systems paradigm.

A.1.5. Usage

This section serves two purposes. First, it gives the reader an overview of the relative size of both methodologies' existing implementations. Second, when observing a methodology being practiced, some shortcomings might be revealed. For example, some methodologies are suitable for small software development projects and fail when applied to large software projects. Other methodologies are intended to solve specific problems like development of information systems or web applications. These kinds of constraints have to be taken into account when describing a development methodology. Often, these shortcomings are not part of the original design but are instead discovered over time.

A.2. Model

To understand a methodology, there is the need to first understand the terms a methodology uses to describe itself. This may not be an issue when comparing two methodologies from a common family, but it certainly is when there is a considerable conceptual gap between them. The terms used, the concepts behind these terms and the use the methodologies give them form an implicit declaration of a methodology's view of the world. This view will influence all activities performed by a methodology and thus needs to be analysed early.

A.3. Life Cycle

The aim of this section is to describe the development life cycle proposed by the methodology. All the development stages considered and their interactions should be covered here, as well as the order in which the stages are executed and whether they are iterative or not. This description will help understand the interaction between roles and frame the activities covered later.

A.4. Rules and Tools

Usually methodologies declare a set of rules that may or may not translate to concrete activities but provide a conceptual framework practitioners can use to face unexpected situations. For example, one of the Extreme Programming[7] rules is continuous testing: work produced must be continuously validated through testing. SCRUM[8], on the other hand, has its continuous meetings as a main rule to increase the teams' speed to deliver work. In the concrete case of FLOSS projects, an important set of rules are those for communications, sometimes referred to as netiquette, which are tightly coupled with the use of tools for communication. For this reason, this section explores both methodology's rules and the use of the tools that enforce these.

A.5. Participation

An important characteristic of a development methodology is how it manages the participation of its participants: how the practitioners are organized to work in a project, which roles take part in the actual code development and how the newcomers adapt to the development process, are topics to be covered in this section. This concern is especially relevant for FLOSS projects, as they rely heavily on volunteer participation.

[7] http://www.extremeprogramming.org/
[8] http://www.scrumalliance.org

A.6. Activities

This section includes six subsections covering the activities in a methodology. These activities do not try to be exhaustive but cover the most interesting ones from our concrete perspective. When executing our method with another set of products or methodologies, the activities highlighted here are likely to change.

A.6.1. Decision-Making

Each methodology defines or implies different policies for decision making. Which roles have the responsibility of development decisions and how the decision-making process is managed are crucial topics. In traditional methodologies, decisions may be made in a top-down manner, leaving very little decision power to the developers. In contrast, some agile methodologies encourage the development teams to decide how the development has to be performed and who is in charge of each task.

A.6.2. Coordination and Communication

Probably the more defining aspect of a methodology is how it arranges the workforce assigned to a project. Responsibilities need be clearly defined to avoid conflict and redundant or overlapping work. For this, the communication structures used for coordination among practitioners play a key role. Some methodologies require specific communication techniques. SCRUM, for example, requires a daily meeting between the development team and the Scrum Master. While this can improve communication efficiency locally, it could also cause problems when developers are distributed among several sites.

A.6.3. Requirements

A crucial stage of a development project is the requirements management. Some methodologies address this issue at the beginning of the project, requiring the team to agree on the specific requirements with the customer and write a requirement specification and expecting that requirements do not change during the development. Other methodologies accept that initial requirements are likely to change and use a more flexible approach, continuously gathering requirements and refining the product expectations during the entire development. How the methodology addresses this issue should be discussed here.

A.6.4. Planning and Control

After the workforce, the most valued and scarce resource a methodology must handle is time. Every methodology states in some way or another how the planning and control of the development should be performed. Some have a very light planning phase, focusing on starting the implementation early, relying on short term planning. Others recommend concrete planning techniques or tools. Usually the control of the development is linked with an explicit planning phase, but sometimes, as in many FLOSS projects, the planning is done concurrently with the decision-making.

A.6.5. Information Availability

Having the right information at the right time is crucial for all methodology practitioners. Furthermore, the quality of the final output of each process is largely dependent on the quality of the required input information. The technical documentation of a project as well as development guidelines and project artefacts should be easily and timely available for the interested roles. High information availability, including logs of past decisions and properly indexed information sources, provides a way to facilitate reusability and increases the development productivity and quality. Which measures are adopted by the methodology to improve information availability, and which drawbacks these measures imply, should be covered here.

A.6.6. Verification and Validation

There are many techniques for detecting bugs and increase product quality. From code reviews to regression testing, how a methodology ensures product quality should be described here. Not only the specific techniques used but also when they are performed, how bugs are fixed, and especially how testing and development interact.

Appendix B.

In this appendix, we present the complete details of the comparison between Streamline and FLOSS development methodologies with accompanying analysis. The comparison follows the outline of the comparison framework given in Appendix A.

B.1. Background

B.1.1. History

Both the Linux kernel and the FreeBSD projects belong to the early stages of the FLOSS movement. JBoss by contrast, is a good example of a project born during the rise of FLOSS in mid 1990s.

Streamline was created in 2005 by Ericsson. Development of this methodology was driven by a benchmark exercise that showed the need for a new methodology to improve certain aspects of the development processes. Streamline was then designed with the objective of dropping the classical waterfall concepts in order to embrace a more agile way of working. Later, Streamline was upgraded to Enhanced Streamline, with a strong emphasis on early fault detection and test automation (Damm et al., 2006). Streamline is a rather new methodology, but it intensively leverages agile methodology assets. The Agile manifesto was written in 2001 but the agile philosophy has its roots in what is commonly known as lightweight methods, developed mainly in the mid-90s. This is usually understood as a return to how software was developed before the rise of waterfall based methodologies (Tomaszewski et al., 2008), when the FLOSS way of working was shaped.

Although the histories of the FLOSS and Streamline development methodologies are fundamentally different, there is at least a connection point in their common history, i.e. the pre-waterfall era. In the next sections, we will further explore the consequences of this relation.

B.1.2. Objectives

Another significant point of divergence is in the methodology objectives. As explained in the previous section, Streamline was created as a response to benchmark results. The benchmark showed some clear issues and Streamline was meant to solve them. Streamline's high-level objectives can be summarized as:

- Reduce time to market.
- Increase R&D efficiency (Lower development costs to allow for resource reallocation.)
- Improve employee satisfaction and motivation.
- Overall increase in development flexibility, predictability, and product quality.

In contrast, FLOSS methodologies are not designed with any objectives in mind but rather evolve from the particular needs of FLOSS development itself. These needs are better reflected as FLOSS development principles rather than objectives, as we will discuss in the following section.

B.1.3. Principles

As previously mentioned, there is a fundamental difference between Streamline and FLOSS origins. While Streamline has been designed to bring an agile way of working to Ericsson, the FLOSS methodologies have been evolving from the characteristic needs of different FLOSS communities. Streamline's agile roots can be seen in its principles:

- Small, efficient and self-organizing teams, fully responsible from pre-study to delivery.
- Highest prioritized requirements always selected for the next project.
- Predictability by a clear scope. Defined specifications and deadlines to fit into three-month development efforts, although this iteration time is now being further reduced.
- Use of anatomy plans for design, requirements and project management.
- An LSV (Latest System Version) team to work on the product baseline for maintenance, new features, and customizations.
- Decoupling development project's execution from commercial release.

On the other hand, the following two well known principles usually appear when talking about FLOSS

(Raymond, 2001):

- Given enough eyeballs, all bugs are shallow.
- Release early, release often.

These two principles are crucial to understanding the role of information openness and participation in FLOSS communities. The first one, also known as Linus' Law, focuses on parallel debugging. As Raymond (2001) states "adding more beta-testers may not reduce the complexity of the current 'deepest' bug from the developer's point of view, but it increases the probability that someone's toolkit will be matched to the problem in such a way that the bug is shallow to that person." Here, the toolkit must not be confused with a debugging toolset, but rather as e.g. providing a different angle to the problem. While this could appear to cause a huge duplication of work, this seems not to be an issue in the Linux community.

However, the sole application of these two principles fails to explain recurrent aspects of FLOSS development projects. During this study, another principle was revealed, namely simplicity. In short, when a decision needs to be taken, FLOSS projects tend to always choose the simplest solution. Simplicity is a clear principle when looking at how FLOSS communities organize themselves. The preference for the simplest solution extends from software design to tools configuration, all the way up to coordination and communication rules. FLOSS projects tend to choose the path of least resistance. FLOSS has a very good reason to avoid complexity wherever possible, i.e. the need to enable external contributors to make contributions. Unfortunately, this is better reflected in smaller FLOSS projects compared to the sample used in our study, as one of the selection criteria was to look for well-established, long-lived and large-sized projects. Even then, this tendency towards simplicity clearly appears when compared to industry's methodologies in general.

If we temporarily turn our focus to the numerous small and young FLOSS projects (not selected for the case study but often mentioned in the literature reviewed), we observe that a new FLOSS project usually only needs a text editor and a compiler to start. This ease of starting a FLOSS project is behind its perceived innovative attitude. If the project is successful and starts attracting contributors, more infrastructure is added as needed, e.g. a mailing list and an issue tracker. If the number of project contributors keeps growing and the communication channels start to overload, the community begins to develop behavioural guidelines while splitting teams and communication channels to handle this growth (Fogel, 2005). These behavioural guidelines are designed and enforced by the community as a whole. There is a need for the community to perceive given guidelines as valuable or their use would not withstand the coming and going of contributors. The same policy is similarly applied to the usage of tools. Contributors are free to select their toolkit hence, as a consequence, only tools that provide a real value stand in the community.

FLOSS projects, thus appear to follow an approach of least resistance when selecting tools and rules. When a behavioural rule or development tool loses its value it is quickly dropped from the scene. However, in industry, modifications to the ways of working (e.g., the addition or removal of a tool, or simply a change in behavioural conventions) is an activity that usually requires an investment of some kind. This causes complexity to accumulate until this investment is justified. Thus, it must be understood that, along with participation and frequent releases, a preference for simplicity is also a FLOSS principle.

B.1.4. Paradigm

Avison and Fitzgerald (1995) discuss methodology from different paradigmatic points of view. As they state, the real benefit is not to absolutely classify a methodology but rather to gain the insight that such a discussion can provide. On a first level, a methodology can be seen as belonging to the science paradigm, based on decomposing the problem, or to the systems paradigm, based on the holistic belief that the whole is more than the sum of its parts. Avison and Fitzgerald (1995) are clearly focused on information systems so they lead the discussion towards the approach a given methodology takes to decompose the problem to be solved using an information system. In our case, we will lead the discussion towards how the methodology faces the actual development activities, not

the problem per se.

At a process level, both FLOSS methodologies in general and the Streamline methodology endorse the system paradigm, avoiding the waterfall orientation of decomposing the building of a system into linear stages. By using iterations and frequent releases, both methodologies understand that development is more than the sum of all its processes. Interactions between processes are as important as the processes themselves.

If we concretize this orientation to the requirements and design processes, even if Streamline does not really address how design should be done, a science paradigm emerges. Both FLOSS and Streamline, in practice, rely on a divide-and-conquer approach to requirements implementation. For instance, in our FLOSS projects, especially in the Linux kernel and FreeBSD, contributed source code must be split in obviously correct patches that fix a single bug or implement a single feature. Streamline has certain roles dedicated to splitting requirements to fit into 4–6 weeks development efforts.

Another relevant comparison point is whether the methodology adopts a nominalist or realist ontological view of software development. While realism postulates that the universe comprises objectively given, immutable objects and structures, nominalism affirms that reality is not a given immutable 'out there' but rather socially constructed. Here, FLOSS and Streamline offer two radically different orientations. Traditionally, as shown by Iannacci (2005), all "production processes depend on pulling together individual efforts in a way that they add up" Although FLOSS is quite particular in this sense because "authority within a firm and the price mechanism across firms are standard means to efficiently coordinate specialized knowledge in a complex division of labour—but neither is operative in open source." In a FLOSS project the core team, usually formed by the project founder and experienced developers, maintains the project vision and a, usually very vague, roadmap. However, there is no particular effort to enforce this roadmap on the community members, who are free to choose where to focus their efforts. The reality for a FLOSS project is clearly nominalistic, being socially constructed rather than imposed from a single point of authority. In industry methodologies, such as Streamline, where the ratio between effort and production is of most importance, it is easy to detect several realist mechanisms to ensure that all participants' efforts point at the same direction. These two radically different ways of understanding the reality of a project is, in our opinion, behind most failures when commercial firms try to exploit FLOSS methods, techniques and tools.

B.1.5. Usage

This section presents usage information for the FLOSS and Streamline development methodologies. Specifically, it characterizes the kind of projects that implement these methodologies, the trends that many of them follow, and the limitations found in their usage.

Streamline is used by approximately 100 projects within Ericsson, with 5 to 100 developers in each project. FLOSS methodologies of various kinds are found in more than 18,000 active projects (Daffara, 2009), ranging from 1 to 1,000 participants. However, the majority of the FLOSS projects involve a small group of developers (Squire et al., 2009).

A small team trend is confirmed in the majority of FLOSS projects, where the developers tend to split in groups of no more than five to six developers (this is especially the case in larger projects (Germán, 2003)). This characteristic forces the systems being developed to be more modular, so that small teams can work concurrently on one artefact. The establishment of a FLOSS project entails very low investment effort allowing easy exploitation of a given opportunity as well as entry into niche domains. Even sectors that are not profitable, and thus there is no company exploiting them, often have some kind of FLOSS presence.

There is no specific rule a project must follow to apply FLOSS methods, tools and techniques besides the obvious openness of FLOSS. In contrast, projects that wish to adapt the Streamline methodology to their concrete needs must meet a 60-point checklist. This checklist ensures that the project is consistent with the methodology and that some crucial roles and processes take place accordingly,

e.g., that the LSV team is present and maintains a product baseline (see Appendix B.3). Examples of different Streamline customizations include ICE (merging the changes made by each team when the development project reaches a milestone), used by Project A, and One-Track (using only one code branch where all developers are active), used by Project B.

With regard to possible drawbacks encountered when applying FLOSS and Streamline methodologies, there is a coincidence. As an interviewed Streamline expert stated, and Tomaszewski et al. (2008) corroborated, Streamline usage might lead to long-term architecture deterioration. Its short iteration approach and the focus on adding functionality leads to an increased risk of taking shortcuts leading to the accumulation of technical debt. Furthermore, when a big architectural change is needed, the short project iterations become a drawback, due to the difficulty in splitting such an effort into small tasks. However, these problems are not limited to Streamline. In the FreeBSD project, when the community faced the architectural change of implementing SMP[9] support, they found similar difficulties due to a similar continuous integration approach (Jørgensen, 2001).

Another drawback of adopting the FLOSS model is the protection of the intellectual property. When a company needs to secure its intellectual property, adopting a FLOSS approach might seem counterintuitive. Also, when looking at FLOSS projects, there is another characteristic that might exclude several software markets from the FLOSS world. Usually, the developers of a FLOSS project are also users of its output, i.e. the artefact being developed. This increases their knowledge about the project and also their motivation. Therefore, when this is not the case, the adoption of the FLOSS model might fail.

B.2. Model

Even if few methodologies state it explicitly, all assume a concrete model or view of how to organize the development process. For instance, as Streamline has an agile heritage, it endorses an iterative approach. Terms like processes, procedures and roles are so common that they belong to any software engineer's vocabulary, but the specific use that a methodology gives to these concepts forms an implicit declaration of its view of the world.

FLOSS methodologies are quite different from other well-known industry methodologies in the explicitness of these descriptions. There have been several research efforts to identify traditional software engineering processes in FLOSS, with different level of success. In FLOSS, while it is certainly straightforward to identify development as a process, but requirements, design and planning resist a process modelling approach. When modelling FLOSS communities (the most famous example being the onion model (Conklin et al., 2005)), one has to rely almost exclusively in how they define roles and responsibilities, due to the scarce use that FLOSS makes of explicit defined processes and procedures. For instance, in the Linux kernel project, the only explicitly specified procedure is the patch submission process. This procedure was not defined from the beginning but was later added as a way of handling the growth of the number of developers contributing with patches. Even then, this procedure is not formally defined nor enforced but written in a series of how-tos and guidelines by experienced patch contributors. In FreeBSD the situation is even more relaxed as contributors can commit the patches themselves. However, the contributors generally follow the pre-commit review process because there is a consensus that this benefits the overall code quality and the project as a whole (Jøgensen, 2001). The FreeBSD and JBoss projects do maintain comprehensive project documentation efforts[10]. However, this documentation is written more as guidelines than specifications and most topics cover recurrent technological issues rather than procedures. Its objective is clearly more educational than regulatory, even though JBoss has a slightly higher use of procedure specification due to requirements from Red Hat, the firm selling JBoss products and services.

In place of defined processes, when we analysed the Linux kernel and the FreeBSD projects, we found that both communities relied mostly on clearly stating who does what. Examples of this are

[9] SMP: Symmetric Multiprocessor Support. The ability of an OS kernel to schedule processing threads to several CPUs so they run concurrently.
[10] http://dl.dropbox.com/u/2437798/JAIS/rawdata.pdf.

MAINTAINERS files (Anonymous, 2009a), which play an important role in the patch submission process, the responsibility distribution and, ultimately, in the process of decision-making. Additionally, in the Linux kernel, with its huge number of contributors, the MAINTAINERS file is used as a scalability mechanism to avoid flooding higher roles with too much information. The details of these activities are further explained in Section 4. Of course, methodologies used in industry also make intensive use of role definitions. The difference lies in that industry, Streamline specifically, trusts a combination of roles and procedures, while in FLOSS communities procedures are rarely defined.

B.3. Life Cycle

When looking at the development life cycle of the studied projects, we find that FLOSS and Streamline approaches are significantly different. However, FLOSS and Streamline share an iterative life cycle approach that makes the comparison possible. The complete Streamline life cycle is shown in Figure B.1. (LSV is short for latest system version and NIV stands for network integration verification).

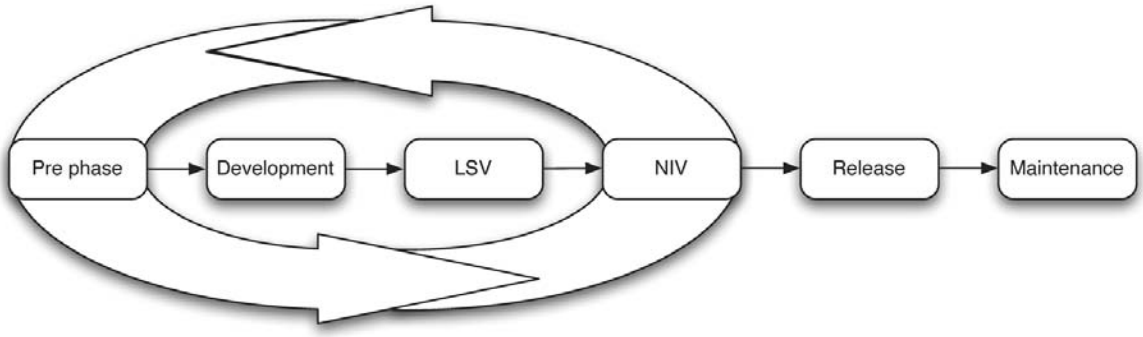

Figure B.1. The Streamline Life Cycle

The first box in Figure B.1 represents the Pre phase that has the prioritized requirements as an input and includes the first planning activities. The development phase includes the actual implementation of the requirements and delivers the finished code to the Latest System Version (LSV) team, which handles the test coordination and controls the project's baseline. The NIV (Network Integration Verification) phase is when end-to-end testing is performed. These first four phases are executed in an iterative way until all desired functionality reaches release status. The following step is then to package the product and make it commercially available. Finally, the maintenance phase includes processing 'Trouble Reports' from clients, which are retrofitted to the next development iteration, and for delivering 'Correction Packages' when necessary.

In contrast, the Linux kernel and FreeBSD development life cycles are centred on code changes, showing the path a code change must follow to be included in a release. Based on the life cycle model of Linux (Iannacci, 2003) and FreeBSD (Jørgensen, 2001), one can clearly see that both life cycles are very similar and can be summarized as in Figure B.2. When a code patch is ready, it is sent to the community for parallel reviewing. The contributors review the code and give feedback to the patch creator until it is considered to have reached a certain degree of quality. At this point FreeBSD has a pre-commit phase to integrate the patch in the source code repository and perform local testing to ensure that it does not break the build. Interestingly, the Linux kernel project does not have this phase because only the project leader Torvalds has the right to commit code to the main source tree (in some specific cases this is relaxed). Therefore, no patch is added to the official release until Torvalds, or someone appointed by him, personally accepts it. Instead, the patch might be incorporated in a development source tree for testing. In the case of the Linux kernel, explicit endorsements of the patch from all the involved module maintainers and at least one of Torvalds' lieutenants is needed for Torvalds to even consider it. In FreeBSD instead, any committer can directly apply the patch, provided nobody has complained about it during the public pre-commit review phase.

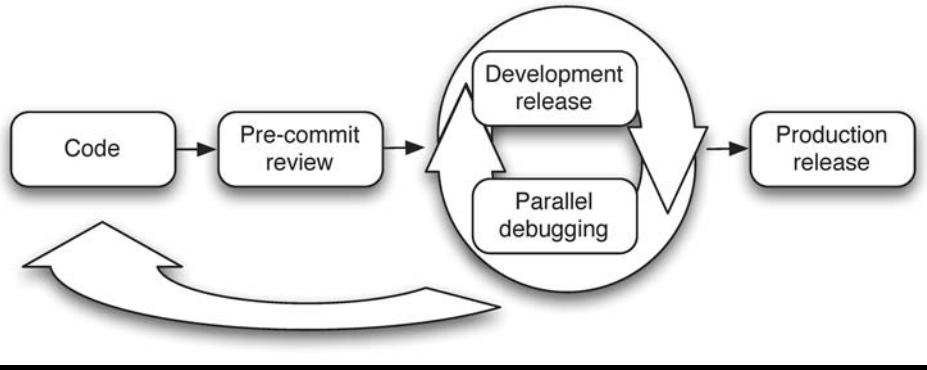

Figure B.2. The Linux and FreeBSD Life Cycle

Finally, in both cases, several cycles of parallel debugging are performed on development releases until a production release is made available. These production releases happen every 18 months in the FreeBSD project and every 3 months in Linux kernel. The release engineer, Torvalds in the case of the Linux kernel, is responsible for orchestrating the re-base between the development and production code branches as well as the needed code freeze, during which all patches, which are not bug fixes, are refused. In fact, this code freeze is usually the only deadline commonly found in FLOSS life cycles. Concerning FreeBSD, Jørgensen (2001) states that the code freeze helps to establish a common goal for the entire project; Iannacci (2005) even affirms that the Linux kernel development process may be decomposed into a sequence of feature freeze cycles, each signalling the impending release of a stable version. A typical FLOSS life cycle can, thus, be understood as VCS-centric , where iterations are not delimited by discrete requirements inflow but rather by the possibility to commit to the VCS or not.

The Linux kernel and FreeBSD examples can be considered as typical, if somewhat formal, FLOSS development approaches. Although they do not explicitly cover requirements or planning phases in their life cycle picture, it does not mean these phases are incompatible with the FLOSS methodology. A good example of mixing the open source way of working with more commercially oriented activities is found in the JBoss Application Server project. In Figure B.3. the JBoss Application Server life cycle is summarized. Despite the remarkable similarities between JBoss and Streamline life cycles, it should be noticed that JBoss' commercial processes must follow FLOSS rules. That is, as the community participates in all stages, they need to be equally open and transparent.

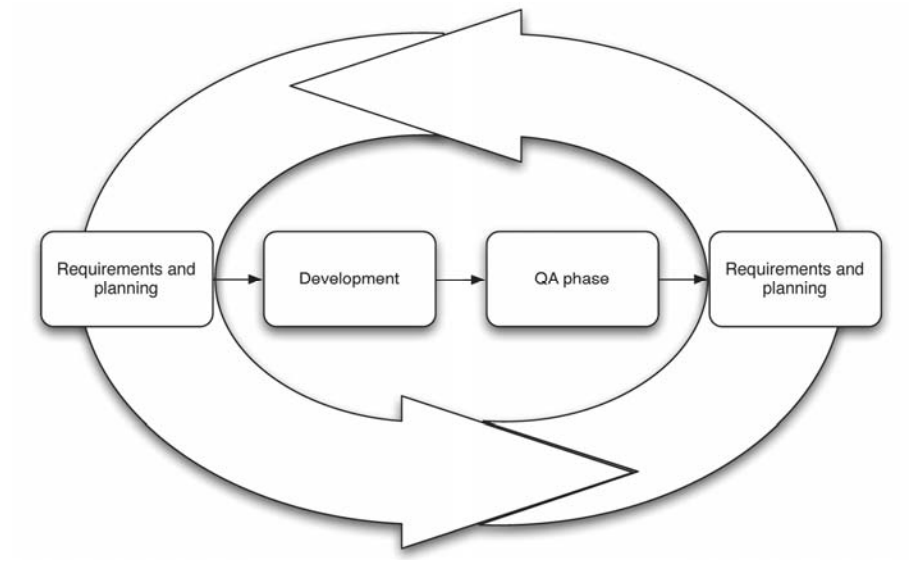

Figure B.3. The JBoss Life Cycle

B.4. Rules and Tools

As mentioned earlier, one of the means by which Streamline achieves a short iteration time is by its emphasis on automation tools, specifically, test automation. This is a trend that FLOSS projects are following as well: all three FLOSS projects included in this study rely on test automation.

However, FLOSS projects rely on additional categories of tools to support development. The tools usually found in FLOSS communities include (Fogel, 2005):

- Web site
- Mailing lists.
- Version control system
- Bug tracking
- Real-time text chat (usually IRC)

Of course, these tools are also found in commercial settings. According to Fogel (2005), the difference resides in that in FLOSS, "elaborate systems have evolved for routing and labelling data appropriately; for minimizing repetitions so as to avoid spurious divergences; for storing and retrieving data; for correcting bad or obsolete information; and for associating disparate bits of information with each other as new connections are observed."

These systems are obviously based on tools, but the success of the systems comes from the way these tools are used. The distributed nature of FLOSS development is obviously behind the need to use tools for communication, but this need is so heavily rooted in the FLOSS communities that it is intrinsically mixed with the community behavioural rules. In the 1980s, when FLOSS communities started to appear, the limitation of the available tools (most importantly, text-based and low bandwidth) had an important shaping effect on the norms for use of the tools. For instance, VoIP and videoconferencing software is widely available nowadays, but FLOSS projects rarely make use of them because these technologies do not fit with the norms of FLOSS communities. A conversation is not automatically logged and indexed for later retrieval: it is easier to search the last message topics on a mailing list looking for relevant topics than listen to several potentially non-interesting VoIP conversations.

In industry, with its intensive use of face-to-face meetings, the advantage of communication tools that automatically store the conversation is not leveraged. Instead, all the meeting documentation has to be written down immediately, which is time consuming. Besides, past discussions and decisions are not always stored, which causes discussions to be repeated when, for instance, people are joining or leaving the project.

Furthermore, FLOSS communities have a clear preference for tools that allow for asynchronous communication as these kinds of tools allow a community member to handle a larger amount of information, regardless of the time zone. In FLOSS projects, having important conversations that the community cannot observe or participate in, goes against the norms. This combination of tools and practices gives FLOSS a way to address common scalability issues (usually referred to as Brooks' Law (Brooks, 1975)). Finally, the openness and the fact that most FLOSS developers participate concurrently in several projects have facilitated quick propagation of best practice and feedback. Thanks to this, a collective agreement on tools and rules exists and is quite consistent across FLOSS projects, despite that it has not been formally written anywhere.

It is also noticeable that this combination of tools and norms scales down to small projects. The mentioned tools are easy to setup for small projects and their configuration complexity will not appear until the project starts to experience real growth. Similarly, the norms used at small projects are simple. For big projects, norms start to evolve with the lessons learned and may reach the point where a project starts to write some documentation to explain the local behavioural rules to newcomers. All three of the projects studied have this kind of documentation.

On the other hand, with Streamline, 65% of the practitioners in our survey stated that they have the power to select some tools (mostly local tools), but not the important ones (such as source code repository, bug tracker, wiki, etc.). This finding was later confirmed in the interviews: there is a general perception that adding a new tool to the global Ericsson toolset is too bureaucratic. Despite this, 57% of respondents had a generally good opinion on the tools used. There are good reasons for Ericsson to enforce some enterprise-wide common tools, for instance, for IT maintenance. However, some developers have stated that this one size fits all approach is problematic, and that each project should be able to select a set of tools according to its own local needs. Another interviewee stated that the important tools are developed outside the company, which leads to too long lead time between Ericsson needing a feature and receiving it.

B.5. Participation

This section exposes characteristics of the studied FLOSS and Ericsson projects regarding developer's organization, project participation and newcomers' adaptation.

Development in the Ericsson projects studied is done by small self-organizing teams. These teams are multidisciplinary, being responsible for the design, implementation, testing and technical documentation of the development project. Streamline handles concurrent development by tracking dependencies with an anatomy plan. This plan helps in managing the component dependencies so the tasks can be planned in an efficient manner, i.e. avoiding collisions and architectural deviations. However, Streamline does not take special measures to enforce modularity per se.

As is mentioned in Section B.1.5, FLOSS projects tend to split in small teams as well. This characteristic forces the systems that are being developed to become modular, so that small teams can work concurrently. For instance, the high modularity of the Linux kernel allows hundreds of developers to work at the same time in different parts of the source code structure with minimum overlap.

A difference between Ericsson and FLOSS is that FLOSS developers are often members of multiple teams. A survey conducted among 1,136 FLOSS developers by Robles et al. (2001) showed that, on average, a single developer contributes to 2.7 FLOSS projects at the same time. Interestingly, Rigby and Germán, (2006) found the same number when looking at the modules that each Mozilla developer works on. Taking into account that modules in FLOSS can be considered independent projects, this shows the trend of developers being involved in more than one project. The willingness to participate on multiple projects was found in our survey of Ericsson developers. Approximately 70% of the respondents were willing to either change their common tasks or to have more variety. Furthermore, 57% express their interest in participating in other projects, given the possibility. It should be noticed that approximately 28% of the respondents reported being willing to abandon their current project for another.

Some FLOSS projects have explicit means to channel and encourage participation. In FreeBSD experienced developers offer their help and supervision to newcomers, reviewing their code and providing advice in the Mentor matching process. Similarly, the Linux Janitor project provides code reviews, fixes unmaintained code, and does other cleanups for the Linux kernel, tasks that are usually suitable for newcomers. This project also provides a TODO list, an IRC channel and helpful information for those who want to start contributing. Streamline does not provide any special treatments for new developers. Instead, a common policy is that newcomers start off by being software testers to familiarize themselves with the project but most of them stay permanently in this role. Another drawback for newcomers, as indicated by the Ericsson survey results, is the long learning period they face. Half the respondents spent more than two weeks and 38% spent more than one month to learn the Streamline methodology.

B.6. Activities

The final set of topics in the framework - the activities followed in the methodology - was presented as Section 4 of the paper.

About the Authors

Richard TORKAR is an Associate Professor at the School of Computing at The Blekinge Institute of Technology. His focus is on quantitative research methods in the field of software engineering.

Pau MINOVES studied telecommunications at the Polytechnic University of Catalonia. In September 2007, Pau enrolled in the European Master on Software Engineering program provided by the Polytechnic University of Madrid and the Blekinge Insitute of Technology. In 2009, he joined the i2CAT foundation as team lead for the Manticore II development effort. He currently leads the technical executive committee of the Mantycore 7th Framework Programme research project and its software development work package.

Janina GARRIGÓS holds a degree in Telecommunications Engineering from the Polytechnic University of Catalonia, and a Master's degree from the European Master on Software Engineering program from the Polytechnic University of Madrid and the Blekinge Institute of Technology. In 2009 she joined the i2CAT Foundation where she is coordinating the development of R&D projects in the eHealth field. She contributed in the creation of a FLOSS Framework for eHealth R&D projects and presented it recently at Med-e-tel 2010.