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

Janina Garrigós
Pau Minoves

School of Engineering
Blekinge Institute of Technology
Box 520
SE – 372 25 Ronneby
Sweden
This thesis is submitted to the School of Engineering at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Master of Science in Software Engineering. The thesis is equivalent to 2 x 20 weeks of full time studies.

Contact Information:
Author(s):
Janina Garrigós
Address: Av. Catalunya 4, 43311 Vilanova d’Escornalbou, Spain
E-mail: janina.gs@gmail.com

Pau Minoves
Address: Rda. Camprodon 14 3er 1a, 08500 Vic, Spain
E-mail: pau.minoves@gmail.com

University advisor(s):
Dr. Richard Torkar
School of Engineering, BTH

Dr. Oscar Dieste
Facultad de Informática, UPM

School of Engineering
Blekinge Institute of Technology
Box 520
SE – 372 25 Ronneby
Sweden

Internet : www.bth.se/tek
Phone : +46 457 38 50 00
Fax : + 46 457 271 25
ABSTRACT

Today’s software companies face the challenges of highly distributed development projects and constant changing requirements. To be competitive, the software time to market has to be reduced as much as possible while keeping the expected quality. Development methodologies try to address this challenges by introducing new practices, techniques and methods for communication, requirements management, quality assurance, etc. This thesis proposes the adoption of relevant Free/Libre/Open Source Software practices to improve industrial developments.

Many FLOSS projects have proven very successful, producing high quality products with steady frequent releases. The selection of the FLOSS best practices that would benefit industrial developments, and its adaptation for a corporate environment is the aim of this study.

To achieve this goal, a framework to compare FLOSS and industrial development methodologies has been created and executed. Three successful FLOSS projects where selected as study targets, as well as two Ericsson’s projects. The framework served to identify FLOSS methodology strengths and compare them with the Ericsson projects. Analysing the significant differences resulting from this comparison, FLOSS best practices were tailored to fit industrial development environments. The final results of the thesis are six adoption opportunities that aim to improve software quality and overall development productivity while increasing practitioners’ motivation and commitment.

Keywords: Software Development Methodologies, Comparison, Free/Libre/Open Source Software
AKNOWLEDGEMENTS

We would like to thank our academic supervisor Dr. Richard Torkar for his support and inspiring feedback. We are also very grateful to our Ericsson supervisors for their time and invaluable guidance. We are very thankful to our EMSE supervisor Dr. Oscar Dieste for his comments and suggestions, and to Aimar González who provided very useful feedback on the paper.

Also, to Ericsson AB for providing the means that made this thesis possible. Especially, we are very grateful to all the people of Ericsson AB who restlessly found some time for us in their busy agendas and provided a wonderful work environment.
CONTENTS

Paper ............................................................................................................................................. 1

Appendix A  Comparison framework ............................................................................................ A-1
  A.1 Table of contents explained .................................................................................................... A-1
    A.1.1 Background ...................................................................................................................... A-1
    A.1.2 Objectives ........................................................................................................................ A-2
    A.1.3 Principles .......................................................................................................................... A-2
    A.1.4 Paradigm ............................................................................................................................ A-2
    A.1.5 Usage ................................................................................................................................ A-2
    A.1.2 Model ................................................................................................................................ A-2
    A.1.3 Life cycle ............................................................................................................................ A-2
    A.1.4 Rules and tools .................................................................................................................... A-3
    A.1.5 Participation ....................................................................................................................... A-3
    A.1.6 Activities ............................................................................................................................. A-3
      A.1.6.1 Decision-making ............................................................................................................. A-3
      A.1.6.2 Coordination and communication ................................................................................ A-3
      A.1.6.3 Requirements ................................................................................................................ A-3
      A.1.6.4 Planning and control ....................................................................................................... A-4
      A.1.6.5 Information availability ............................................................................................... A-4
      A.1.6.6 Verification and validation ............................................................................................ A-4

Appendix B  Survey ......................................................................................................................... B-1
  B.1 Motivation ................................................................................................................................ B-1
    B.2 Survey results ....................................................................................................................... B-1
      B.2.1 Profile ............................................................................................................................... B-1
      B.2.2 Tasks ................................................................................................................................ B-2
      B.2.3 Tools ................................................................................................................................ B-3
      B.2.4 Tools (Domain specific) ..................................................................................................... B-4
      B.2.5 Way of Working ................................................................................................................ B-5
      B.2.6 Decision making ............................................................................................................... B-7
      B.2.7 Communication ................................................................................................................ B-8
      B.2.8 Documentation .................................................................................................................. B-9

B.3 Analysis ................................................................................................................................. B-10
  B.3.1 Profile ................................................................................................................................ B-10
  B.3.2 Tasks ................................................................................................................................ B-10
  B.3.3 Tools ................................................................................................................................ B-11
  B.3.4 Way of working ..................................................................................................................... B-11
  B.3.5 Communication and documentation ................................................................................... B-11
Adopting Free/Libre/Open Source Software Practices, Techniques and Methods for Industrial Use

Janina Garrigós
School of Engineering
Blekinge Institute of Technology
PO Box 520, S-372 25 Ronneby, Sweden
janina.gs@gmail.com

Pau Minoves
School of Engineering
Blekinge Institute of Technology
PO Box 520, S-372 25 Ronneby, Sweden
pau.minoves@gmail.com

ABSTRACT
Today’s software companies face the challenges of highly distributed development projects and constantly changing requirements. To be competitive, the software time to market has to be reduced as much as possible while keeping the expected quality. Development methodologies try to address these challenges by introducing new practices, techniques and methods for communication, requirements management, and quality assurance among others. This thesis proposes the adoption of relevant Free/Libre/Open Source Software practices to improve industrial developments.

Many FLOSS projects have proven very successful, producing high quality products with steady frequent releases. This study aims to select the FLOSS practices that industrial software developers would benefit from and adapt them for a corporate environment.

To achieve this goal, a framework to compare FLOSS and industrial development methodologies has been created and applied. Three successful FLOSS projects where selected as study targets, as well as two projects at a large software company. The framework served to identify FLOSS methodology strengths and compare them with the projects in industry. Analysing the significant differences resulting from this comparison, FLOSS best practices were tailored to fit industrial development environments. The final results of the thesis are a set of key adoption opportunities that aim to improve software quality and overall development productivity while increasing practitioners’ motivation and commitment.

Keywords
Software Development Methodologies, Comparison, Processes, FLOSS, Open Source, Industry, Case Study

1. INTRODUCTION
In recent years, the software market has faced several structural changes, raising a whole new set of challenges; one of them being that software companies tend to split their software development among globally distributed teams. Also, to be competitive, time to market has to be reduced as much as possible. The need to facilitate developer’s reallocation, and the learning curve associated to switching to a new project, have a high impact on time to market. In the telecom industry, with the rise of mobile-based services, an increasing number of stakeholders demand much diversified functionalities. Thus, this additional complexity impacts development even more.

A software development methodology is, as stated by Avison and Fitzgerald in [2], “... a recommended collection of phases, procedures, rules, techniques, tools, documentation, management and training used to develop a system”. There are many available formal methodologies that a company can follow to drive its development; ranging from the traditional waterfall model to modern ones with a more agile approach. For the last 25 years the Free/Libre/Open Source Software (FLOSS) way of producing software has been evolving in parallel with these, often, more formal methodologies. Noticing the success of several FLOSS projects, research has started to study the peculiarities of FLOSS development. At the same time, industry has shown interest in how massively distributed development teams manage to deliver high quality, modular and flexible FLOSS.

Free/Libre/Open Source Software has been an object of research for some years now. Most successful FLOSS projects have been analyzed mainly from two perspectives: Product and development methodology. Studies investigating FLOSS as a product focus on characteristics such as defect density, software packaging statistics, software growth and number and types of contributors. Target projects of such studies have been Apache [30], the Linux kernel [30], GCC [30], Debian [16]1, OpenBSD [28] and Eclipse [19]. In the second group we find studies investigating FLOSS as a way of producing software. These studies can be broadly categorized by their focus on community culture [9][15], organizational model [27][13], and processes and methods [34][37].

How companies can benefit from adopting open source processes and methods is still an area which, by large, is untouched. One example of an initiative in this direction is HP’s Progressive Open Source [8]. In [8] they focus on the FLOSS openness, trying to adapt it for an industry environment. They create an infrastructure for HP’s projects to open their source codes within the organization; trying to harness the benefits of FLOSS code visibility.

Ericsson AB, one of the major software producers of telecommunication systems in the world, has shown interest in FLOSS. Based on the hypothesis that industry can adopt some best practices from the FLOSS methodologies; the aim of this study is to collect, identify and analyze these practices and then present them as adoption opportunities to industry. The development methodology used in Ericsson’s projects is called Streamline. It has been created by and for Ericsson and is nowadays widely implemented with some project specific variations. This methodology has its roots in agile methodologies, concretely adopting an iterative approach and small-team
development. It focuses on early fault detection and testing automation, leveraging other agile techniques for communication and planning.

To achieve the main goal of the thesis, provide adoption opportunities from FLOSS that can benefit industrial developments, a group of five software projects were studied. Two of them belonged to Ericsson and three were known successful FLOSS projects. The study performs a comparison of FLOSS' and Ericsson’s development methodologies at use in these projects, and identifies the differences where FLOSS can provide an improvement. Thus, this thesis will have a broad scope, setting up a case study with an emphasis on development practices, methods and techniques.

This paper provides two contributions. First, we propose a framework to compare development methodologies based on FLOSS special characteristics. Thus, this framework is specially tailored for comparing FLOSS with other methodologies. Second and most importantly, we expose a list of suggested adoption opportunities that can improve industrial software developments. Additionally, as intermediate step, we provide an analysis of the differences between FLOSS and Streamline using the mentioned framework.

The paper is structured as follows: in Section 2 we explain the research methodology and the creation of the comparison framework. The expected outcome, a list of adoption opportunities a company can use to improve its processes will be exposed in Section 3. Section 4 analyzes the rationale behind these adoption opportunities and presents the concrete comparison between the two methodologies. Finally, Section 5 discusses the significance of the results found and the associated validity issues.

2. RESEARCH METHODOLOGY
To accomplish the final goal of finding FLOSS practices that can be adopted by Ericsson this study requires the adoption of qualitative research techniques. There is a need for a research method that allows the collection of a voluminous amount of data and provides a way to filter the valuable information among the rest. Considering that the study is performed in-house at Ericsson, our experience and judgment when studying their development methodology has been used to tailor the research.

These are the four steps chosen to produce the expected adoption opportunities. The first two steps follow a grounded theory approach, where data collection precedes theory formulation:

1. Collect information about the development methodology applied at Ericsson and in FLOSS projects.

2. Analyze, synthesize and categorize the information in order to allow a side-by-side comparison of FLOSS and Ericsson’s methodologies.
   a. Grounded theory equivalent: memo building, 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 for Ericsson developments.
   a. Comparison analysis and industry feedback.

The research has been framed as a case study, where two projects at the telecom company Ericsson are studied and compared with three FLOSS projects. Due to the exploratory aim of this research, where no previous information was available to direct the study towards concrete development methodology issues, a framework that favoured the consideration of all the methodology viewpoints was desired. Unfortunately, none of the frameworks found were adequate. Therefore, we focused our efforts on creating a custom comparison method that allowed the maximum coverage of the methodology perspectives.

The next section discusses the suitability of the comparison frameworks found, the reasons why they were discarded for our study and the finally chosen framework that was used as a base for creating our own comparison method. Section 2.2 contains the case study selected projects and its main characteristics as well as information gathering sources.

2.1 Methodology comparison framework
The desired outcome of the comparison phase is a list of differences between Ericsson and FLOSS project development. This is a requirement not all analyzed frameworks provide. One of our requirements was a comparison framework that helped to elicit non-obvious issues as well as the capacity to elicit issues at all levels (activities, principles, roles, etc.) We considered several comparison frameworks: Davis’, Avison and Taylor, Avison and Fitzgerald, and NIMSAD as exposed at [3] as well as CDM as applied at [17].

The NIMSAD and Avison and Taylor comparison frameworks assume the existence of a problem the methodology is trying to address. Thus, they focus on choosing the more appropriate methodology depending on the encountered problem. We did not want this problem-oriented approach, our need was to find differences at different levels, and thus these frameworks were discarded. Similarly, Davis’ framework was discarded for being too oriented towards requirements engineering.

CDM was discarded for two main reasons. On the one hand, it is used to compare methodologies of the same family, assuming low delta and focusing on spotting little differences. For this reason, it is unsuitable for comparing two potentially very divergent ways of working. On the other hand, its purpose is only to find the best methodology among some candidates, based on some subjective criteria. We were not interested in finding whether FLOSS is better than Streamline, but to find concrete differences between the two methodologies, so this framework was discarded too. We finally found that the Avison and Fitzgerald framework provided the desired balance between guidance, coverage and customizability.

2.1.1 Avison and Fitzgerald customization
Avison and Fitzgerald [3] identify two reasons for comparing methodologies. An academic reason, that is, to understand the nature of methodologies. And a practical reason, which implies
the selection of a methodology for a given development. Taking these premises as starting point they propose two frameworks.

The first one provides a list of ‘ideal-type’ criteria that a methodology should fulfil. Although this is a long list, with twenty-eight criteria, which covers many software development issues, it makes a number of assumptions that may force a low score on some modern methodologies that are proven successful. For instance, the known preference of Agile methodologies for having minimum documentation is considered a bad trait by this framework, which is, at least, debatable. The criteria list is available in Table 1.

Table 1. Avison and Fitzgerald assessment criteria.

<table>
<thead>
<tr>
<th>Criteria</th>
<th>Avison and Fitzgerald Assessment Criteria</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rules</td>
<td>Information System Boundary</td>
</tr>
<tr>
<td>Total Coverage</td>
<td>Designing for Change</td>
</tr>
<tr>
<td>Understanding of the Information Resource</td>
<td>Effective Communication</td>
</tr>
<tr>
<td>Documentation Standards</td>
<td>Simplicity</td>
</tr>
<tr>
<td>Validity</td>
<td>Ongoing Relevance</td>
</tr>
<tr>
<td>Early change</td>
<td>Automated Development Aids</td>
</tr>
<tr>
<td>Inter-stage communication</td>
<td>Consideration of User Goals and Objectives</td>
</tr>
<tr>
<td>Effective problem analysis</td>
<td>Participation</td>
</tr>
<tr>
<td>Planning and Control</td>
<td>Relevance to practitioner</td>
</tr>
<tr>
<td>Performance Evaluation</td>
<td>Relevance to application</td>
</tr>
<tr>
<td>Internal satisfaction</td>
<td>Scan for Opportunity</td>
</tr>
<tr>
<td>External satisfaction</td>
<td>Separation of analysis and design</td>
</tr>
<tr>
<td>Visibility</td>
<td></td>
</tr>
<tr>
<td>Teach-ability</td>
<td></td>
</tr>
</tbody>
</table>

The second proposed framework provides seven basic points of view from which a methodology can 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. Even though, being a good summary for describing a development methodology, this second approach was too abstract for our needs, ignoring some details that could uncover attributes very valuable for the comparison phase, such as communication and coordination issues or life-cycle details.

Table 2. Avison and Fitzgerald comparison framework table of contents.

<table>
<thead>
<tr>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Philosophy</td>
</tr>
<tr>
<td>o Paradigm</td>
</tr>
<tr>
<td>o Objectives</td>
</tr>
<tr>
<td>o Domain</td>
</tr>
<tr>
<td>o Target</td>
</tr>
<tr>
<td>Model</td>
</tr>
<tr>
<td>Techniques and tools</td>
</tr>
<tr>
<td>Scope</td>
</tr>
<tr>
<td>Output</td>
</tr>
<tr>
<td>Practice</td>
</tr>
<tr>
<td>o Background</td>
</tr>
<tr>
<td>o User base</td>
</tr>
<tr>
<td>o Participants</td>
</tr>
<tr>
<td>Product</td>
</tr>
</tbody>
</table>

As Avison and Fitzgerald themselves state “... the above criteria [Table 1] form a useful checklist but clearly need to be tailored for a particular purpose”. In our particular case, both mentioned frameworks were combined and customized to reinforce their strengths and provide input to our comparison method. Concretely, our comparison framework was modified to include aspects such as life cycle, participation and an increased focus on certain activities. This custom table of contents also provides the structure for Section 4.

Figure 1 offers an overview of the comparison method. Readers familiar with the Osborn-Parnes CPS (Creative Problem Solving) method will easily recognize a divergent-convergent thinking pattern. The first steps, up to the field memo were planned to generate as much ideas as possible. The Avison and Fitzgerald matrix was used for this purpose and it is explained in Section 2.1.1.1. The last steps were designed to synthesize all this information and reveal both important issues and information structure. The clustering technique chosen is explained in Section 2.1.1.2. This custom table resulted in the following comparison framework: First, a more appropriate table of contents used for analyzing the differences, becoming our own comparison framework (see Appendix A). Second, a list of knowledge gaps with respect to the methodologies. Filling these knowledge gaps brought the comparison to its final state.

2.1.1 Divergent thinking

As stated before, the purpose of the first steps of the comparison method was to elicit as much information as possible without a concrete goal in mind. This lack of focus was especially chosen to allow for broader thinking and discussion. On the initial phase, several pages of notes were taken summarizing interviews and documentation analysis. Topics covered were as broad as software engineering itself.

To extend these notes, two matrices were created: One containing information from Ericsson and the other from the FLOSS projects. 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 discussion about the methodologies from a huge variety of viewpoints, producing unstructured but interesting ideas. It also spotted several knowledge gaps overlooked during the initial study phase. Both these ideas and the knowledge gaps were captured in a field memo, a structured document that serves as a vehicle to articulate our findings.

2.1.1.2 Convergent thinking
The field memo was summarized in a number of sticky notes. All sticky notes were placed on a whiteboard without any special order. Collaboratively among all researchers, sticky notes were grouped by topic, generating issue clusters.

Quickly, a topic structure emerged, making it easy to spot both relations and the relative size of each issue. This topic structure became the foundation of the comparison framework to use when comparing Streamline and FLOSS methodologies (see Appendix A). Moreover, for each cluster where we detected a lack of information, a set of questions was written to find overlapping knowledge gaps 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.

Finally, with all the collected information, the rich side-by-side comparison could be performed. This comparison is exposed in Section 4.

2.2 Case study setup
This research was executed at an Ericsson development site allowing for high interaction between researchers and practitioners. Two projects inside this development unit were available for the case study. Due to confidentiality reasons we will refer to them as projects A and B.

To achieve a list of suitable adoption opportunities from the FLOSS world, which can be applied in Ericsson project development, a set of FLOSS projects were identified. These FLOSS projects were analyzed together with A and B projects and used as an input for our comparison method.

There are many successful FLOSS projects, with likely very diverse configurations. Provided that a deep qualitative research was intended, only a small subset (three projects) could be selected. A sample this small could never be representative of the whole FLOSS universe, and this was never the intention. A decision was made to favour a sample that resembled the targeted Ericsson projects, as it would be more likely to reveal suitable adoption opportunities.

To make the appropriate choice, initial Ericsson project information was collected. This information served as an initial contact with Streamline methodology and more importantly, as a way to characterize Ericsson projects in terms of size, number of developers, domain, etc. These gathered characteristics were lately used to select FLOSS projects that resembled the Ericsson ones. The following sections cover the concrete attributes of projects A and B and the selected FLOSS projects.

2.2.1 Industry project characteristics
Several attributes were considered to characterize Ericsson’s software development processes and methods. However, the final list was synthesized to only four:

- **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 work full time at project A (around 14,400 man-hours per month), 40 work at project B (around 6,400 man-hours per month) with this number going to double in less than a year. Considering the inequality of dedication between industry and FLOSS developers (on average FLOSS developers work less than 10 hours a week [33]) we were looking for FLOSS projects with over 100 active contributors.
- **Legacy.** The project A development started seven years ago and it thus carries a high amount of legacy code. Project B, while it was started three years ago, reuses and maintains the code of a nine years old project. Therefore, legacy is a significant property of both projects.
- **Similar domain.** As all software projects, A and B design decisions are driven by, not always explicit, non-functional requirements. Some of them, like robustness and performance, could be elicited but many more subtle ones would remain hidden. In order to implicitly import all this prioritization of non-functional requirements, FLOSS projects from a similar domain had to be selected. This domain was stated as: Enterprise server-side project with an emphasis on high availability and performance.

2.2.2 Selected FLOSS projects
The final FLOSS study targets have been chosen because they match the industry project characteristics and are especially interesting from a development methodology point of view.

The selected FLOSS projects are the Linux Kernel, JBoss Application Server and FreeBSD. They are all supported by a large group of contributors that have managed to create large and established code bases through several years.

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, paid and non-paid contributors coexistence in the same development community is a new relevant trend.

| Table 3. Software project characteristic values |
|---|---|---|---|---|---|
| Lines of Code (thousands) | Project A | Project B | Linux Kernel | JBoss AS | FreeBSD |
| Developers | 90 | 40 | 1,000 | 50 | 200 |
| Legacy (years) | 7 | 9 | 15 | 10 | 13 |
| Domain | Enterprise server-side |
The table above contains the project characteristic values for all projects. However, other attributes of interest were also considered to select the projects:

- **Linux Kernel Project.** With an impressive community of over 1,000 developers, 70% of which serve the interests of over 100 companies [26], the Linux Kernel Project is probably the largest example of FLOSS software engineering at work. How this stakeholder interest diversification is managed to keep up with functionality requests while maintaining robustness and performance is of special interest in our study.

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

- **FreeBSD.** FreeBSD development philosophy has similar principles with industry’s Continuous Integration practices applied at projects A and B [24]. This has been a decisive attribute to select FreeBSD as a project to study.

### 2.2.3 Information gathering

An important part of this research is the information gathering phase. All the information found from the studied projects was collected using field notes that were then processed through our comparison method. To perform a side-by-side comparison we aggregated the methodology information about the three FLOSS projects and also the information about the two Ericsson projects. In the following, when mentioning Streamline development, we are referring to both A and B Ericsson projects, as it is the methodology they both use. The information gathering was performed with two objectives in mind:

- **Define the development methodology of the five studied projects.** The Avison and Fitzgerald matrix was used to ensure a broad coverage of the methodology characteristics.

- **Elicit Ericsson development challenges.** Practitioners’ subjective opinion 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 on written form is overwhelming both at Ericsson and in a typical FLOSS project. However, a completely different issue is how close this information is to reality once theory is put to practice and time passes. For this reason, whenever possible, we have used practitioners’ knowledge to confirm and extend the information found. Below, the sources of information for industry are detailed.

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

- **Interviews,** twelve different key roles offered different points of view over the complete life cycle of the project development. Responsible engineers in the following areas were interviewed: Requirements, design, testing, project management, product management and development methodology. The interviews were conducted in a semi-structured manner as presented in [20].

- **Survey,** once in the comparison phase it became necessary to gather the point of view of the developers and testers about several key topics. It was considered unpractical to interview a representative sample of the 130 developers of both projects so a survey was designed and executed.

These topics covered task assignment, tools, ways of working, documentation, communication and decision-making. The survey also provided a practitioner’s opinion overview used in final stages of the method to identify which FLOSS and Streamline differences were especially important.

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%). Raw result data as well as analysis can be found in Appendix B.

When studying FLOSS we relied on the following information sources:

- **Documentation,** including projects web page, Wikis, guidelines, handbooks and mailing list archives. The open nature of FLOSS projects allows for a lot of information to be publicly available. This information was used to extract process, roles and activities information.

- **Research literature,** as we selected three well-known FLOSS projects, there was a certain amount of previous research available. Although, as stated, this research is usually quantitative, it was 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 relied on previous research literature, an already conducted survey [33] and first-hand documents from FLOSS communities.

### 3. RESULTS

The adoption opportunities resulting from the thesis are exposed below. Each adoption opportunity is showed together with the according FLOSS-Streamline differences that sustain it. These differences are grouped according to the relevant subsection of the analysis (Section 4).

The entire comparison, analysing both the significant differences that produced an adoption opportunity and the ones that did not, can be found in Section 4. Further discussion on the adoption opportunities details, motivation and benefits is located in Section 5.
1. Reduce technical debt.

Technical debt is a dangerous property of a source code asset that will hamper all future developments. FLOSS projects are very reluctant to add technical debt to the code base. However, the situation is different in a deadline-based industrial setting. Being subjected to frequent deadlines might lead industrial projects to technical debt accumulation.

Table 4: Relevant differences for reducing the technical debt.

<table>
<thead>
<tr>
<th>FLOSS</th>
<th>Streamline</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.3 Life-cycle</td>
<td>4.6.4 Planning and control</td>
</tr>
<tr>
<td>Parallel reviewing phase before commit. No patch is committed without community acceptance.</td>
<td>Strict deadlines. Features delivered not always meeting the expected quality. Harder bug-fixing phase.</td>
</tr>
<tr>
<td>There are no fixed deadlines to implement a feature. Technical debt is strictly avoided.</td>
<td>Strict deadlines lead to technical debt accumulation. There is no planned effort to reduce previous technical debt.</td>
</tr>
</tbody>
</table>

2. Define an entry path for newcomers.

As FLOSS projects growt depend largely on attracting contributors, there is a well established path for newcomers to gain competency inside the community. In industrial settings the workforce is believed to be more static. However, this is rapidly changing because of globally distributed development or the need to change developers from one area of competency to another. There is a need to proactively setup entry paths for newcomers that both lowers the learning curve and allows them to be productive sooner.

Table 5: Relevant differences to define and entry path for newcomers.

<table>
<thead>
<tr>
<th>FLOSS</th>
<th>Streamline</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.5 Participation</td>
<td>4.6.6 Verification and validation</td>
</tr>
<tr>
<td>Mentor matching and Janitor project help introducing newcomers.</td>
<td>Newcomers often initially assigned to less qualified tasks.</td>
</tr>
<tr>
<td>Finding and fixing bugs is performed close in time and usually by the same person.</td>
<td>Testers are in charge of finding bugs, and developers of fixing them.</td>
</tr>
</tbody>
</table>

3. Increase information availability and visibility.

Enriching information artefacts by exposure is one of the main characteristics of FLOSS. While there must be some limitations on exposure, a similar attitude can provide many benefits for industrial developments as well.

Table 6: Relevant differences to increase information availability and visibility.

<table>
<thead>
<tr>
<th>FLOSS</th>
<th>Streamline</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1.3 Principles</td>
<td>4.6.5 Information availability</td>
</tr>
<tr>
<td>“Given enough eyeballs, all bugs are shallow”</td>
<td>Development sites strictly keep their code and information locally.</td>
</tr>
<tr>
<td>Information publicly exposed as it is created.</td>
<td>Information locked inside projects. Documents are not shared until they are finished.</td>
</tr>
</tbody>
</table>

4. Embrace tools for communication and decision-making.

FLOSS communities rarely make use of meetings. Conversation and decision-making regularly happen in virtual public areas such as mailing lists. This way, conversations can not only cross geographical and time zone borders but are automatically archived for later reference. In industry, widespread use of meetings has both advantages and drawbacks. A balance between the two techniques can provide many benefits.

Table 7: Relevant differences for embracing tools for communication and decision-making.

<table>
<thead>
<tr>
<th>FLOSS</th>
<th>Streamline</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.4 Rules and tools</td>
<td>4.6.2 Coordination and communication</td>
</tr>
<tr>
<td>Tools support for storing past decisions and creating a body of knowledge.</td>
<td>Tools used to asynchronous communication. Permanent conversations used to feed knowledge base.</td>
</tr>
<tr>
<td>Tools for testing automation and parallel development. Past decisions are usually not stored.</td>
<td>Intensive use of meetings. Survey shows that documentation is hard to retrieve.</td>
</tr>
</tbody>
</table>

5. Let practitioners influence ways of working.

Actual practitioners should have a way to influence their application of the methodology. It will not only help to reduce local inefficiencies but increase embracement by building a consensus on how to do things.

Table 8: Relevant differences to let practitioners influence ways of working.

<table>
<thead>
<tr>
<th>FLOSS</th>
<th>Streamline</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1.3 Principles</td>
<td>4.6.1 Decision-making</td>
</tr>
<tr>
<td>Simplicity. Methodology guidelines designed and enforced from the community.</td>
<td>Methodology decisions left to management.</td>
</tr>
</tbody>
</table>
6. Allow task selection.

Allow employees to self-assign a percentage of their weekly work hours to a pool of available tasks. This would not only increase employee motivation by allowing for self-development, but also shape the development methodologies to embrace employee mobility instead of fearing it.

Table 9: Relevant differences for allowing task selection.

<table>
<thead>
<tr>
<th>FLOSS</th>
<th>Streamline</th>
</tr>
</thead>
<tbody>
<tr>
<td>In average, developers work in 2,7 projects at the same time.</td>
<td>Developers performing fixed tasks in the same project. Survey shows willingness to have more task variety.</td>
</tr>
<tr>
<td>“Given enough eyeballs, all bugs are shallow”</td>
<td>Development sites strictly keep their code and information locally.</td>
</tr>
<tr>
<td>Coordination structure based on heterarchies and role responsibility.</td>
<td>Planning and control done through iterations and fixed teams.</td>
</tr>
</tbody>
</table>

4. ANALYSIS

This section analyses the differences found between FLOSS and Ericsson development methodologies. These differences are structured using the table of contents of the created comparison framework (see Appendix A).

4.1 Background

4.1.1 History

The historic backgrounds of FLOSS and Streamline are profoundly different. The FLOSS concept was officially born in 1985 when the Free Software Foundation (FSF) was created by Stallman. To understand the motivations of Stallman we have to look at the two previous decades. In the 1960s and the 1970s operating system development efforts were mainly done at university laboratories. In this academic culture, sharing the source code was inherently derived from the scientific culture. If somebody did an improvement they were expected to publish it. This situation started to change with the rise of proprietary operating systems in late 1970s and early 1980s. By buying the software and releasing it under a “closed” license, the source code sharing practice disappeared through legal imperatives. The FSF and the seminal FLOSS license, the GPL, were forged as a reaction to this trend. The objective was to create a complete software ecosystem that remained free. Both the Linux Kernel and the FreeBSD projects belong to the early stages of the FLOSS movement. JBoss instead, is a good example of a project born during the rise of FLOSS in mid 90s.

Streamline, on the other hand, was created four years ago, by Ericsson. Before that, a benchmark 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 classic waterfall concepts in order to embrace a more agile way of working. Later, Streamline was upgraded to produce Enhanced Streamline, with a strong emphasis on early fault detection and testing automation [7].

Even if Streamline is a rather new methodology, it intensively leverages agile methodology assets. The Agile manifesto was written in 2001 but the agile philosophy has its roots in what is usually known as lightweight methods, developed during mid-90s. This is usually understood as a return to how software was developed before the rise of waterfall based methodologies [32], when the FOSS way of working was shaped. Therefore, even if FLOSS and Streamline development methodologies history is fundamentally different, there is at least a connection point on its common roots on the pre-waterfall era. In next chapters, we will further explore the consequences of this relation.

4.1.2 Objectives

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

- Reduce product’s 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 methodology is not designed with any objectives in mind but rather evolves from the particular needs of FLOSS development itself. However, these needs are better reflected as FLOSS development principles rather than objectives.

4.1.3 Principles

As previously exposed, 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 methodology has 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 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.
- A LSV (Latest System Version) team to maintain the product baseline for maintenance, new features and customizations.
• Decoupling development project execution from commercial release.

On the other hand, the following two well known principles usually appear when talking about FLOSS, both stated by Raymond in [31]:

• Linus’ Law: “Given enough eyeballs, all bugs are shallow”.

• “Release early, release often”.

These two principles are crucial to understand the role of information openness and participation in FLOSS communities. The first one, Linus’ Law, focuses on parallel debugging. As Eric Raymond 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, toolkit must not be confused with a debugging toolset, but rather a “slightly different perceptual set and analytical toolkit, a different angle on the problem”. While this could appear to cause a huge duplication of work, “In practice, [this] almost never seems to be an issue in the Linux community. One effect of a ‘release early and often’ policy is to minimize such duplication by propagating feedback fixes quickly”.

However, the sole application of these two principles fails to explain recurrent aspects of FLOSS developments. During the course of this research, another principle was revealed:

• When a decision needs to be taken, FLOSS projects tend to choose always 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 developments tend to choose the path of least resistance. FLOSS has a very good reason to avoid complexity wherever possible: The need for allowing external contributors to do actual contributions. Unfortunately, this is poorly reflected in our chosen FLOSS projects, as one of the selection criteria was to look for well-established, long lived and large-sized projects. Even then, this tendency stands when comparing with industry methodologies.

If we temporarily turn our focus to the numerous small and young FLOSS projects, we often see that a new FLOSS project usually only need 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, such as a mailing list, issue tracker, etc. as the need arises. 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 [11].

These behavioural guidelines are designed and enforced from the community as a whole. There is a need for the community to perceive given guidelines as valuable or their use would not stand the come and go of contributors. Same policy is similarly applied to tools usage. Contributors are free to select their tool set, so only tools that provide a real value stand in the community.

FLOSS projects, thus, follow a approach of less resistance to select tools and rules. When a behavioural rule or development tool loses its value it starves of usage and is quickly dropped out of the scene. However, in industry, modifications to the ways of working (being the addition or removal of a tool, or simply a change in behavioural conventions) are 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.

4.1.4 Paradigm

In [3], Avison and Fitzgerald propose the discussion of a methodology from different paradigmatic points of view. As the authors state, the real benefit is not to absolutely classify a methodology but rather to gain the insight that this methodology 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 believe that the whole is more than the sum of its parts. Avison and Fitzgerald’s book is clearly focused on information systems so they orient the discussion towards the approach a given methodology takes to decompose the problem to be solved with that information system.

In our case, we will lean the discussion to how the methodology faces the development, not the problem.

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

If we concretize 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 the Linux Kernel and FreeBSD, contributed source 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 up to fit into 4-6 weeks development efforts.

Another relevant comparison point is whether the methodology exercises an ontological nominalist or realist view to 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 socially constructed.” Here, FLOSS and Streamline offer two radically different orientations. Traditionally, as exposed by Iammacci [21], 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 top developers, maintains the project vision and a, usually very vague, roadmap. Anyhow, there is no real effort to enforce this on the community members, who are free to choose where to point their efforts to. A FLOSS project reality is clearly nominalistic, being socially constructed rather than
irradiated 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.

4.1.5 Usage
This section explores usage information about FLOSS and Streamline development methodologies. Specifically, it characterizes the kind of projects that implement these methodologies, the trends that most of them follow, and the limitations found in their usage.

Streamline is applied approximately on 100 projects within Ericsson, with about 5 to 100 of developers in each project. FLOSS instead, is found in more than 18,000 active projects [6], ranging from 1 to 1,000 participants each. However, the majority, of the FLOSS projects involve a small group developers [10].

The small team trend is confirmed in almost all FLOSS projects, where the developers tend to split in groups of no more than 5-6 developers, especially in larger projects [14]. This characteristic forces the systems being developed to be more modular, so that small teams can work concurrently in each module.

The establishment of a FLOSS project entails very low investment effort allowing to easily exploit a given opportunity and to enter 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, Streamline projects have the possibility to adapt the methodology to their concrete needs as long as they fulfill 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. the LSV team is present and maintains a product baseline (see Section 4.3 for more information). Examples of different Streamline customizations are ICE and One-Track integration approaches. Project A uses ICE, an approach that consists on merging the changes of each team when the development project is finished. On the other hand, project B uses One-Track where there is only one code branch where all the development teams work at the same time.

With regard to possible drawbacks encountered when applying FLOSS and Streamline, there is a coincidence. As an interviewed Streamline expert stated and [35] confirms, Streamline usage might lead to long-term architecture deterioration. Its short iteration approach and the focus on functionality addition cause the 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 impossibility of splitting such effort in small tasks. This is not only a Streamline situation. In FreeBSD development, when the community faced the architectural change of implementing SMP1 support, they found similar difficulties due to their continuous integration approach [24].

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 is counterintuitive. 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 it. This increases their knowledge about the project and also their motivation.

4.2 Model
Even if few methodologies state it directly, they all assume a concrete model or view of how to organize the development process. For instance, as Streamline has an agile heritage it basically endorses an iterative model approach. Terms like processes, procedures and roles are so common that they belong to any software engineer’s vocabulary. The use that a methodology gives to these concepts forms an implicit declaration of its view of the world. In this sense, FLOSS is completely different to well known industry methodologies.

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; requirements, design and planning resist a process modelling approach. When modelling FLOSS communities, (the most famous example being the onion model [4]), one has to rely almost exclusively in roles and responsibilities, due to the scarce use that FLOSS makes of processes and procedures.

When this research approached both the Linux Kernel and the FreeBSD projects, it was revealed that both communities relied mostly on clearly stating who does what. Examples of this are the MAINTAINERS files [12],[29], which play an important role in the patch submission process, the responsibility distribution and, ultimately, in the decision-making process. 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 will be further explained in Section 4.6.

Of course, industrial methodologies also make intensive use of role definitions. The difference lies in that while industry, and concretely Streamline, trusts a combination of roles and procedures; in FLOSS communities procedures are rarely defined. For instance, in the Linux Kernel, the only specified procedure is the patch submitting process. This procedure was not defined from the beginning but added as a way of handling the growth of the developers’ base. Even then, this procedure is not formally defined nor enforced but written in a series of how-tos and guidelines by veteran kernel contributors. In FreeBSD the situation is even more relaxed as contributors can commit the patches themselves. The contributors stick to the pre-commit review process because there is a consensus that this benefits the overall code quality and the project as a whole [24].

Both FreeBSD and JBoss maintain comprehensive project documentation efforts [12] [23]. 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. JBoss does a slightly higher use of procedure specification due to the needs of Redhat; the firm selling JBoss based products and services.

---

1 SMP: Symmetric Multiprocessor Support. The ability of an OS kernel to schedule processing threads to several CPUs so they run concurrently.
Summarizing, by using procedures only for conflict resolution and letting contributors find their way themselves on how to get the job done, FLOSS communities have built an efficient working environment where people is attracted to contribute without monetary rewards.

4.3 Life cycle

When looking at the development life cycle of the studied projects, we found 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 2.

![Figure 2: Streamline's life cycle.](image)

The first box in Figure 2 represents the Pre phase. It has the prioritized requirements as an input and performs the first planning activities. The development phase performs the actual implementation of the requirements and delivers the finished code to the Latest System Version (LSV) team that handles the test coordination and controls the project’s baseline. The NIV (Network Integration Verification) phase is where the end-to-end testing is performed. These first four phases are executed in an iterative way until all the desired functionality reaches a release level. The following step is to package the product and make it commercially available. Finally, the maintenance phase is responsible for processing “Trouble Reports” from clients, which are retrofitted to next development iteration, and delivering “Correction Packages” when necessary.

On the other hand, 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 [22] and FreeBSD [24], one can clearly see that both life cycles are very similar and can be summarized as in Figure 3.

![Figure 3: Linux and FreeBSD life cycles.](image)

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 satisfactory. At this point FreeBSD has a pre-commit phase to integrate the patch in the code repository and perform local testing to assure that it does not break the build. Interestingly, the Linux Kernel does not have this phase because only the project leader Torvalds has permission to commit code to the source tree. Therefore, no patch is added to the development release until he personally accepts it. In the case of the Linux Kernel, explicit endorsement of the patch from all the involved module maintainers and at least one of Torvalds’ lieutenants is needed to 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.

Finally, in both cases, several cycles of parallel debugging are performed via development releases until a production release is published. 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 rebase between the development and production code branches as well as the needed code freeze, during which all patches that are not a bug fix are refused. In fact, this code freeze is usually the only deadline commonly found in FLOSS life cycles. Concerning FreeBSD, [24] states that it helps to “establish a common goal for the entire project”, and [21] even affirms that “The Linux kernel development process, therefore, 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 the possibility to commit to the VCS or not.

The Linux Kernel and FreeBSD examples can be considered as typical 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 the JBoss Application Server. In Figure 4 the JBoss AS life cycle is summarized. Although there are remarkable similarities between the JBoss and Streamline life-cycles, it should be noticed that JBoss’ commercial processes must follow FLOSS rules. That is, as community participates in all stages, they need to be equally open and transparent.

![Figure 4: JBoss's life cycle.](image)

4.4 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, concretely testing automation. This is a trend that FLOSS follows as well. For instance, all our three studied projects rely on

---

2 VCS: Version Control System. Software tool used to track changes made to a source code tree.
testing automation too. However, the reliance of FLOSS on tools brings the issue to a completely new level.

The distributed nature of FLOSS development is obviously behind the need to use tools for communication, but this need is so heavily rooted in FLOSS that it is intrinsically mixed with the community behavioural rules. In the 1980s, when FLOSS started, the limitation of the available tools (most importantly, text based and low bandwidth) had an important shaping effect on what was lately known as netiquette. For instance, the tools usually found in FLOSS communities are [11]:

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

However, these tools can also be found in commercial settings. According to Fogel [11], 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 a big component of them is enforced by the before mentioned netiquette. For instance, VoIP\(^3\) and videoconferencing software is widely available nowadays, but FLOSS projects rarely make use of them. This is because these technologies do not fit well with the netiquette of FLOSS communities. A conversation is not automatically logged and indexed for later retrieval, and it is easier to scan the last message topics on a mailing list looking for relevant topics than listen to several potentially non-interesting VoIP conversations. FLOSS communities have a clear preference for tools that allow for asynchronous communication, as they allow for a community member to handle a bigger amount of information, independently of the member’s time zone.

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 expressly, consuming time. Besides, past discussions and decisions are not stored, which causes discussions repetition when different or new people are involved, for instance, because some interested parties are in a different time zone.

In FLOSS projects, having relevant conversations where the community can not observe or participate goes against the netiquette. This combination of tools and netiquette form the FLOSS cornerstone against common scalability issues, usually referred as Brooks’ Law effects. This combination has been evolving for 20 years, clearly influenced by FLOSS principles (see Section 4.1.3), the need to be simple enough to attract contributors and the remarkable size some projects have achieved. Also, the openness and the fact that most FLOSS developers participate concurrently in several projects has facilitated quick best practice propagation 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 rules scale quite well 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 suffer growth stress. Similarly, the netiquette rules used at small projects is simple and easily derived from common sense. For big projects, this netiquette starts 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 our three projects have this kind of documentation.

On the other hand, at Streamline, 65% of practitioners 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 was lately confirmed in the interviews, as there is a general perception that adding a new tool to the global Ericsson’s 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, like for instance, IT maintenance. However, some developers have stated that this one size fits them 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 fact that the important tools are developed outside, leads to too long lead time between Ericsson needing a feature and getting it.

4.5 Participation

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

The development at the studied Ericsson projects is handled by small self-organizing teams. These teams are multidisciplinary, being responsible for the design, implementation, testing and technical documentation of the development project. As is mentioned in Section 4.1.5, FLOSS developments tend to split in small teams as well. This characteristic forces the systems being developed to be more modular, so that small teams can work concurrently. For instance, the high modularity of the Linux Kernel allows more than one thousand developers to work at the same time in different parts of the source code structure with minimum overlapping.

Streamline handles concurrent development tracking dependencies with an anatomy plan. This planning artefact helps managing the component dependencies so the tasks can be planned in an efficient manner, avoiding collisions and architectural deviations. However Streamline does not take special measures to enforce more modularity.

While the projects in FLOSS and at Ericsson include several teams developing specific parts of the project, in FLOSS the development efforts are usually individual. A survey conducted among 1,136 FLOSS developers by Robles et al. [33] showed that, on average, a single developer contributes to 2.7 FLOSS projects at the same time. Interestingly, in [32] found the same number when looking at the number of 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 development. This trend is confirmed in another survey, this time conducted at

\(^3\) VoIP: Voice over Internet Protocol. Name given to the set of protocols used to offer telephony services over IP networks.
Ericsson especially for this study (see Appendix B). It shows that a 70% of the respondents are willing to either change their common tasks or have more variety. Furthermore, 57% express their predisposition to participate also in other project developments, given the possibility. It should be noticed that only a 28% of the respondents would abandon its current project for another.

Some FLOSS projects have means to channel and encourage participation. Mentor matching in FreeBSD or the Linux Kernel Janitor project is a good start for new contributors. In FreeBSD experienced developers offer their help and supervision to newcomers reviewing code and providing advice. Similarly, the Janitor project provides code reviews, fixes unmaintained code, and does other cleanups for the Linux Kernel; tasks usually adequate for newcomers. The project also provides a TODO list, an IRC channel and helpful information for those who want to start contributing. Streamline does not provide special treatments to new developers. Instead, a common policy is that newcomers start being testers to familiarize themselves with the project. As a result, most of them stay permanently in this role with no chance to become a developer. Another drawback for newcomers shown in the Ericsson survey results is the long learning curve they face. Half the respondents sent more than two weeks and a 38% more than one month to learn Streamline methodology.

To channel the participation, FreeBSD and Linux Kernel, as well as most FLOSS projects, resort on the role of module maintainers. Such a role is not found in Streamline development. This will be covered in the next Section.

4.6 Activities

4.6.1 Decision-making

Decision-making is done across several levels at Streamline projects. As stated before, Streamline has a principle on self-organizing teams. This is reflected in several decision-making processes. Implementation related issues are fully handled by development teams, including the design of the solutions. However, a SAT (System Architecture Team) participates in this decision-making to ensure that the overall project architecture is respected. Similarly, teams are also responsible to generate 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 being done by the development teams. This is covered with more detail in Section 4.6.4.

Even with the considerable number of decisions that are left to development teams, our conducted survey shows that 62% of the respondents think that participating 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 are both bad”. Interestingly, this is how decision-making is done at FLOSS.

Concretely, decision-making at FLOSS is done around the role of the maintainer, and ultimately the benevolent dictator. Fogel [11] describes the benevolent dictator as “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 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 usually several module maintainers but just one benevolent dictator.

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.

The discussions being public have several beneficial effects for the community. First, even if compared to Streamline, the decision-making process can seem slow and cumbersome, in the long run, decisions are usually more elaborated incorporating more points of view and, hence, 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) having all the rationale available. To avoid discussions on recurrent topics that have already been decided, FLOSS communities make extensive use of FAQ pages and mailing list archives. Also, 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, 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” [22].

4.6.2 Coordination and communication

In Streamline, coordination is achieved by having small teams performing small (below 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, these goals have already been prepared beforehand to avoid conflicts and overlapping. By the use of anatomy plans on requirements, planning and a clear roadmap, the implementation tasks are allocated so that teams can work concurrently. A traversal SAT team provides assistance to avoid issues from an architectural point of view and progress is monitored by the means of burn down charts. When a development team needs some resource that cross the project boundaries (such as approval to acquire some third-party product) a set of procedures are established to do so.

While the FLOSS structure looks simpler at first sight, the underlying communication interactions are quite more complex. [21] describes the coordination and communication structures in the Linux Kernel as heterarchical, that is: “interconnected, overlapping, often hierarchical networks with individual components simultaneously belonging to and acting in multiple networks, and with the overall dynamics of the system both emerging and governing the interactions of these networks”. If we analyse FLOSS from a hierarchical point of view we will soon notice that there are rarely more than three levels. Usually a heterogeneous contributors 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 (although we
have already explained that its main role is delegation), while the position of the rest is relative to the interaction at hand. If we analyse the typical evolution of a developer in a FLOSS project, we will see that (s)he is sometimes acting as a maintainer, while other times acting 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 organizational structure can appeal more ordered and efficient, FLOSS communities extract several benefits of heterarchies. Firstly, by the means of the maintainer’s system, tricky tasks are usually automatically routed to the best available developer, while easier tasks are left to more inexperienced ones, who in turn use them to gradually gain credit. This credit gain is of key importance as it fuels the meritocracy system that keeps the FLOSS development going. Secondly, heterarchies distributed authorities naturally drive the development in several directions at once. As Torvalds put it, while “having somebody who shows you the ‘one true path’ may be very nice for getting a project done, [...] every once in a while [that path] it’s definitely the wrong thing to do. And if you only walk in single file, and in the same direction, you only need to make one mistake to die. In contrast, if you walk in all directions at once, and kind of feel your way around, you may not get to the point you thought you wanted, but you never make really bad mistakes, because you always ended up having to satisfy a lot of different opinions. You get a more balanced system”. Finally, as opposed to industrial settings, where change (i.e. developer’s rotation) is seen as a menace to project stability, FLOSS minimizes risks by embracing continuous change, reducing dependence on a single individual.

Despite these strengths, FLOSS 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-to’s, FAQ) and contribution ownership can be tracked (for instance, FreeBSD uses the svn blame command for this [24]). Additionally, three main kinds of information must be abundant:

- **Technical know-how**: so that the necessary learning process associated to the 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, keeping behaviour violations to a minimum.
- **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 to long and is blocking further development.

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

### 4.6.3 Requirements

Requirements are handled in Streamline in a traditional way. Strategic Product Managers gather and prioritize requirements from a number of sources (customers, marketing, etc). Then, Technical Product Managers divide them into smaller requirements that can be implemented by a single team in single iteration. For doing this, a cross-functional board of System Managers and Node Architechts provides technical insight and first effort estimates. It is important to notice here that while requirements are treated radically different in FLOSS, the fundamental validation mechanism is the same. Streamline validates by exposing requirements artefacts to an expert board while FLOSS exposes them to public discussion.

However, at FLOSS, requirement validation goes as far as requiring an actual working implementation. We have already exposed the *code then decide* attitude of FLOSS. In some communities, most notably 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 will not add the feature implementation to the main tree because “nobody takes out features, they are stuck once they get in. Which is exactly why my job is to say ‘no’, and why there is no ‘accepted unless proven bad’” [22]. 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 production kernel.

The above is possible because developers of these projects are also prominent users. This is not the case for projects A and B (or JBoss for that matter). In JBoss, however, pre-requirement artefacts (feature requests) are also 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. Non paid contributors can then pick tasks which they want to contribute to. This way, JBoss obtains a balance between community driven development and steering.

### 4.6.4 Planning and control

At the studied 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. The deadlines are strict and a lot of effort is put on meeting the deadlines. The short iteration approach and the pressure to meet deadlines may lead to taking shortcuts concerning code maintenance. As has been mentioned in Section 4.1.5 this can cause the accumulation of technical debt. The design shortcuts made during one iteration, to achieve the expected functionality, cannot be fixed in the next one because new requirements and deadlines pop up. Streamline makes use of a team called Design Follow Up (DFU), which is supposed to fix these kinds of shortcuts. However their duties also include fixing Trouble Reports, which of course are prioritized over code clean up. An employee working with project A confirmed this problem: “[…] we should educate and encourage developers to refactor code they modify in line of duty, [if not we] let the code
deteriorate and the technical debt increases. I'm not talking about big restructuring, but fixing something every time you commit”.

In contrast, FLOSS does not set specific deadlines to 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, FreeBSD and Linux Kernel pre-commit reviewing phase enforces 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) and meet the delivery deadlines. Anyway, they perform a parallel reviewing phase where the possible shortcuts are detected and fixed so the required code quality can be assured.

The estimation in Streamline projects is done in two phases. First, a quick study is performed to have an approximation of the effort needed to implement each requirement. After that, the resources are allocated and the requirements are prioritized 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 experience, capabilities and previous design solution. A drawback with this approach is its rigidity. Eventually, emerging requirements arise from the development, for instance, 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 do not ask 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, it is handled individually and no specific schedule is set.

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 non started, ongoing and finished tasks and is updated continuously. The burn down chart shows the tasks completed over time and provides a quick insight on 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.

4.6.5 Information availability

As has already been discussed in Section 4.1.3, information openness is at the very core of the FLOSS way of working. Obviously, industry projects have very good reasons to keep information locked in-house, but often this lacking hinders employee’s work. 63% of 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 cross-project collaboration.

While FLOSS’ absolutely open attitude towards information can seem a bit extreme, it is of key importance to project success. In FLOSS artefacts are publicly exposed (not only source code) from the day they are created, as opposed to industry, where 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. In contrast, FLOSS’ publish immediately policy allows for some kind of informal lazy review process, most likely increasing the number of contributions that help in enhancing the quality. For instance, anybody can enrich an enhancement request on the issue tracker, provide insight to a bug discussion or propose a modification on an already submitted source code patch.

Outside the projects, this open door policy is also very useful. A potential user can easily access a lot of information about the ongoing development, making it easy to evaluate the quality of the product, future plans, open bugs, etc. It is also 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 usually 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.

4.6.6 Verification and validation

Verification and validation in both FLOSS and Ericsson projects is composed by two main phases. First, in FLOSS, a pre-commit review is done, where faults are detected by reading the source code among many contributors. Similarly, in Streamline a pair-programming approach is used, so developing and reviewing is done at the same time. The second validation phase is the automated testing. Both FLOSS and Ericsson projects use techniques like unit testing and regression testing that are automated to a large extent. However, in projects like FreeBSD or 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, they need to rely on a stricter reviewing phase before testing commences. This relation is also mentioned by Rigby and German in [32] as: “The amount, quality, and type of testing appeared to be related to how easy it is to automate tests. Fewer automated tests appear to force a project to do more formal code review”.

When looking at software verification and validation in FLOSS, 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 FLOSS projects the task of finding a bug is often followed by the fixing of that same bug. As it is usually done by the same person, it rewards recognition by the community. Thus, not only fixing bugs, but also finding them turns to be a motivator for the FLOSS developers [24]. In Streamline instead, testers are just responsible for finding bugs leaving the bug fixing for developers who get the recognition, who in turn, complain that testers just spot problems without contributing to the solution.

5. DISCUSSION

This section contains further discussion on the six resulting adoption opportunities of this thesis. For each adoption opportunity we analyse its expected advantages as well as its implementation requirements.

1. Reduce technical debt.

Commercial developments are unavoidably surrogated to meet promised deadlines. However, is important not to let the urgent displace the important. With an iterative approach,
which places a deadline every few weeks, often quick solutions replace elegant ones. As technical debt accumulates the architecture deteriorates, the code becomes harder to maintain and bugs are more expensive to fix. Two strategies can be discerned when dealing with technical debt.

First, situations that create technical debt need to be kept to a minimum. Technical debt, as bugs, is cheaper to eliminate near to the insertion point. It is important to build developers consensus on the minimum quality of the source code additions to the code base. Code review by a broader audience is a good way to filter shortcuts that would otherwise pass unnoticed.

Second, plan in advance for technical debt. No matter how much effort is put on avoiding it, some amount will always accumulate. There needs to be a pre-emptive allocation of time for efforts made 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.

A reduction of the technical debt will pay-off in terms of maintainability and make it easier to introduce new developers.

2. **Define an entry path for newcomers.**

It is important to have a predefined path that allows new developers to learn while doing productive activities. If this is let unattended, there is a risk of displacing the newcomers to unqualified positions, making their learning curve unnecessarily high.

For instance, with proper support from experienced developers, bug fixing and technical debt reducing activities are a good entry point for new developers. It would allow them to familiarize with the software architecture, perform tasks with different difficulty levels and to be productive from the very beginning.

Following this strategy, they would be ready to be incorporated sooner to the regular development project activities. Additionally, bright developers would have a bigger chance to standout sooner, reducing employee frustration derived from not being able to deliver his full potential.

3. **Increase information availability and visibility.**

A common problem of big software companies is that knowledge sharing between departments often happens only on a managerial level. Often the same component or tool is developed in more than one department without even realizing that redundant work is being performed.

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 centralized searching. It is a mistake to limit information sources to documentation. The whole software source code asset of an organization should be also searchable. It is especially important for technical roles to have read access on other project’s resources. This would make reusability possible, not only for software components but also for technical know-how. This public exposure would also favour modularized designs, allow for external contributions and increase overall code quality.

Additionally, while offline office suites are not going to disappear anytime soon, there are a number of documents that would benefit of 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. Simply put, people deal better with incomplete and early information than late and immutable information.

4. **Embrace tools for communication and decision-making.**

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 (http://agilemanifesto.org). While it is true that it can be the most effective, it also has serious drawbacks, in particular effecting 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 output is a summarized minutes where the conversational context is lost. Also, meeting participants can be badly setup, with uninterested attendants and/or interested parties left outside. A meeting can be the best way to get something decided within a given deadline when desired participants are clearly identified.

Asynchronous communication technologies (like e-mail, where receive and reply are disassociated in time) allows an individual to go through a bigger amount of information without blocking his/her attention at any given time. Concurrent tasks that feed the conversation (like further information processing or gathering) can happen without having to reschedule a meeting.

Additionally, a bigger number of individuals, independently of location and time zone, can observe the conversation with full conversational context available. If it is about an ongoing decision-making, it can not only be improved with feedback, but drive interested parties to a bigger endorsement of taken decisions by assimilating the rationale behind the decision taken.

Most importantly, communications done by electronic means can be automatically archived without loss of information. Especially design and implementation decisions would benefit of being stored for later reference and retrieval. This would form a useful knowledge base that would lower the learning curve for newcomers and ground further decision-making by experienced developers. Moreover, this knowledge would be permanent, independently of key employees leaving a project.

5. **Let practitioners influence ways of working.**

It is obvious that the actual methodology practitioners are the ones who see processes, methods and tools from a closer
6. **Allow task selection.**

The routine of working with the same project, and performing the same tasks over and over again, might decrease the motivation and lower the productivity of the employees. Curiosity and self-development are good motivators. Providing a way to channel them is important to keep employee commitment high.

One solution 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 needs to be implemented carefully:

- While always prioritizing own project’s tasks, employees have to be able to distribute the free time they commit to another project throughout the week. It is important to prevent that everybody is outside their project at the same time because contributors will need support from residents.

- There needs to be a per project permanent knowledge base that outside contributors can look-up without disturbing residents with frequently asked questions. If the learning curve is too high, contributors will simply turn to other projects. See adoption opportunity *Increase information availability and visibility and Embrace tools for communication and decision-making*.

- The task pool must contain activities 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 and who to contact to propose a new task. An outsider’s view can be a source of creative ideas. A resident mentor that help outside contributors would certainly lower the learning curve and ensure that contributions meet the locally required quality standards.

This kind of strategy places significant stress on several aspects of software development. 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 outside contributors would help spot issues that unnecessarily raises the bar concerning the learning process.

### 5.1 Static validation

In order to ensure the trustworthiness of the results, as well as the feasibility to implement the adoption opportunities in a near future, a series of interviews were performed. These interviews targeted managers of both projects A and B as well as a Streamline expert to assess the generalizability of the results.

While all interviewed managers agreed on the potential benefits of the adoption opportunities, some implementation roadblocks were identified that could predate the early ripping of benefits. These were classified in three main groups: economies of scale, localism and departmental protectionism.

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 economies of scale to truly show its potential. Pilot projects, which are by definition done at small scale, can mask or even prevent the delivery of real benefits.

Additionally, a tradition of collocated development teams, meeting based culture and agile methodologies, which embrace face to face meetings, make it difficult for practitioners to voluntarily use 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 on their own department assets, even in detriment of the efficiency of the organization as a whole. This may certainly produce certain degree of resistance to change when implementing practices based on sharing and public exposure of artefacts. Upper management commitment is necessary to drive a cultural change, so department position is not based on the value of locked assets but on the quality of the final output.

### 5.2 Threats to validity

This section will present the threats affecting the validity of this research. We will follow the recommendations from Creswell [5] for addressing validity issues in our qualitative study. In the following we will expose the internal and external validity threats as well as reliability issues.

Regarding the internal validity we can identify three threats: The accuracy of the information found, the 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 contrasted several information sources and used diverse information gathering methods. At Ericsson, three different methods have been used: Project documentation review, key roles interviews and a survey. The information found has been aggregated using a field memo to find convergence. At FLOSS, information found in project documentation has been contrasted to literature research. To reduce the incorrect interpretation of the data collected at Ericsson, periodical member checks have been performed. Two representative members of both studied projects have been reviewing the data collection and
analysis to assure its correctness. Furthermore, to validate the results of the study and check their applicability at Ericsson projects a static validation phase has been performed, see Section 5.1.

The second threat to internal validity has been addressed in a similar way. To ensure that data collected at Ericsson reflected reality, we performed twelve interviews and one survey. Having a broad scope (concerning roles) when selecting the respondents provided different points of view on the same issues, allowing for the identification of valuable and trustworthy data.

Finally, to minimize the information bias introduced by our judgement we worked as close as possible with the practitioners as Creswell [5] recommends, “The epistemological assumption of the qualitative paradigm is based on minimizing the distance between the researcher and the informant”. Another valuable source was the survey results, which confirmed and also refuted several assumptions from our previous research.

The external validity addresses issues that may affect the generalizability of the findings. As mentioned in [5], “the intent of qualitative research is not to generalize findings but to form a unique interpretation of events”. However, two threats have been taken into account to achieve a limited generalizability of our findings: The adequateness of the selected project targets and the appropriateness of the data analysis.

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 differences. To ensure a better representativeness of the FLOSS world, we chose well-known projects with different levels of industry involvement.

The usage of a good data analysis method, which assured the consideration of as much methodology issues as possible, was another threat to external validity. To address this, we designed a method that leverages a formal comparison framework from Avison and Fitzgerald [3] and adapts it to FLOSS methodology characteristics. Although the comparison phase is performed using specific information from the Ericsson projects, the generated adoption opportunities can be generalized to any software development company, because they address typical problems in software industry.

6. CONCLUSIONS AND FUTURE WORK

This thesis has shown that, while being very different, bridges can be build between industry and FLOSS so best practices can be imported. By implementing our suggested practices, we expect an increase in development productivity and software quality, as well as better adaptability to today’s geographically distributed developments and highly competitive markets.

We have provided a ground for future studies in this direction, building a comparison framework especially tailored to spot differences between FLOSS and any other development methodology. Leveraging this framework, FLOSS practices, techniques and methods can be explored, revealing further adoption opportunities when compared with other industrial development methodologies.

Future lines of research could design a concrete implementation strategy to introduce the suggested practices in a specific industrial setting. This should be done taking into account the risks identified in our static validation phase (see Section 5.1). Especially, the introduction of openness concepts supposes a challenge to corporate culture and must be implemented carefully.

7. ACKNOWLEDGMENTS

We would like to thank our academic supervisors Dr. Richard Torkar and Dr. Oscar Dieste for their support and inspiring feedback; and to our Ericsson supervisors for their time and invaluable guidance.

Also, to Ericsson AB for providing the means that made this thesis possible. Especially, we are very grateful to all the people of Ericsson AB who restlessly found some time for us in their busy agendas and provided a wonderful work environment.

8. REFERENCES


[23] JBoss.org community wiki. Available at http://www.jboss.org/community/wiki/Main


[29] Linux Kernel maintainers file. Available at http://lxr.linux.no/linux/MAINTAINERS


[31] Raymond, E. The cathedral and the bazaar: musings on Linux and Open Source by an (Jan 1, 2001), 241.


Appendix A  COMPARISON FRAMEWORK

A.1 Table of contents explained

The following table of contents forms a comparison framework that covers the most important characteristics of a development methodology. It has been created from Avison and Fitzgerald comparison framework to fit FLOSS methodology characteristics. It might 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 between them. 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 our concrete study it has been used to identify the most significant differences between Ericsson’s Streamline and FLOSS methodologies. The most significant differences are the ones where FLOSS is stronger and can provide an improvement for Streamline. The framework contains six main topics, as shows the figure A-1. Below, each category is explained.

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

Figure A.1. Comparison framework table of contents

A.1.1 Background

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

A.1.1.1 History

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

A.1.1.2 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 to improve the software quality. Also, even if high level objectives are the same, methodologies usually disagree on the means to achieve them. While two methodologies may be focus on improve quality, one may insist in 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 this objectives should be mentioned in this section.

A.1.1.3 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 teams development and continuous communication between development stages. FLOSS methodology, on the other hand, beliefs on information openness and short release time.

A.1.1.4 Paradigm

At an even higher level that 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 viewpoint proposed by Avison and Fitzgerald, discussing the ontology (realism or nominalism) and science or systems paradigm.

A.1.1.5 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 on practice, some shortcomings might be revealed. For example, some methodologies are suitable for small developments but might fail when applying them 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 discovered with the experience.

A.1.2 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 who share a common family but it certainly is when there is a considerable conceptual gap between them. The terms used, the concepts behind this terms and the use the methodologies gives to them form an implicit declaration of the methodology view of the world. This view will influence all activities performed by a methodology and thus needs to be analysed early.

A.1.3 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 would help understand the interaction between roles and frame the activities covered later.

A.1.4 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 rules is the continuous testing, it enforces that the work produced must be continuously validated through testing. SCRUM, 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, these rules are commonly referred as netiquette, and it is tightly coupled with the use of tools, concretely for communication. For this reason, this section explores both methodology rules and explores the use of the tools enforced by these.

A.1.5 Participation

An important characteristic of a development methodology is how it manages the participation. How the practitioners organize to work on a project, which roles take part on the developments and how the newcomers adapt to the development process are topics to be covered in this section. This is especially relevant for FLOSS as it heavily relies on volunteer participation.

A.1.6 Activities

This section is composed by six different subsections covering the activities of a methodology. These activities do not try to be exhaustive but to cover the most interesting ones for our concrete comparison. When executing our method with another set of products or methodologies, the activities to highlight here are likely to change.

A.1.6.1 Decision-making

Each methodology defines or implies different policies for decision making. What roles have the responsibility of the development decisions and how the decision-making process is managed are crucial topics. In traditional methodologies the decisions are 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.1.6.2 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 work overlapping. 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 among the development team and the ScrumMaster. While this can improve communication efficiency locally, it could also cause problems when the developments are distributed among several sites.

A.1.6.3 Requirements

A crucial stage of a development project is the requirements management. Some methodologies address it at the beginning of the project, agreeing on the specific requirements with the customer, writing a requirement specification and expecting that they don’t 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 should be discussed here.
A.1.6.4 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 of them have a very light planning phase, focusing on start 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 FLOSS; the planning is done concurrently with the decision making.

A.1.6.5 Information availability

Having the right information at the right time is crucial for all methodology practitioners. Not only that, 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, provide a way to facilitate reusability and increase the development productivity and quality. Which measures adopt the methodology to improve information availability, and which drawbacks these measures imply, should be mentioned.

A.1.6.6 Verification and validation

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

B.1 Motivation

The aim of the survey was to gather the point of view of developers and testers about several key topics related to the development methodology. It allowed us to focus the research on interesting fields were FLOSS could provide meaningful adoption opportunities. It was considered unpractical to interview a representative sample of the 130 developers of both Ericsson projects, so a survey was designed and executed.

The survey topics covered task assignation, tools, ways of working, documentation, communication, and decision-making. It was designed to retrieve information about Streamline that filled our remaining knowledge gaps. Additionally it provided a practitioner’s opinion overview used in final stages of the method to identify which FLOSS and Streamline differences were especially important.

B.2 Survey results

B.2.1 Profile

1) Which product do you belong to? (Choice - Single answer) Required

<table>
<thead>
<tr>
<th>Product</th>
<th>Percentage</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Product A</td>
<td>61%</td>
<td>48</td>
</tr>
<tr>
<td>Product B</td>
<td>39%</td>
<td>31</td>
</tr>
<tr>
<td>Other</td>
<td>0%</td>
<td>0</td>
</tr>
</tbody>
</table>

2) Select one or more of the following roles: (Choice - Multiple answers) Required

<table>
<thead>
<tr>
<th>Role</th>
<th>Percentage</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Developer</td>
<td>70%</td>
<td>55</td>
</tr>
<tr>
<td>Tester</td>
<td>30%</td>
<td>24</td>
</tr>
<tr>
<td>Requirements</td>
<td>3%</td>
<td>2</td>
</tr>
<tr>
<td>Design</td>
<td>33%</td>
<td>26</td>
</tr>
<tr>
<td>Management</td>
<td>3%</td>
<td>2</td>
</tr>
<tr>
<td>LSV Team</td>
<td>8%</td>
<td>6</td>
</tr>
<tr>
<td>Other, please specify:</td>
<td>10%</td>
<td>8</td>
</tr>
</tbody>
</table>

Team Leader
CPI Team
Developers do testing, design and programming...
Technical Coordinator
System Manager
Customer Product Information (CPI)
Technical writer
System Manager, requirements and other
3) **How much time have you been working on the computer science field?** (Years) *(Open ended text - One line) Required*

See Figure B.1.

4) **How much time have you been working at Ericsson?** (years) *(Open ended text - One line) Required*

See Figure B.1.

5) **How much time have you been working at product A / B?** (years) *(Open ended text - One line) Required*

See Figure B.1.

![Graphs](image)

Figure B.1 Scatter graphs comparing time at the computer science field, Ericsson and studied product.

### B.2.2 Tasks

6) **Do you think your assigned tasks match your background?** *(Choice - Single answer) Required*

<table>
<thead>
<tr>
<th>Yes</th>
<th>78%</th>
<th>58</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>22%</td>
<td>16</td>
</tr>
</tbody>
</table>

7) **Do you think your assigned tasks match your interests?** *(Choice - Single answer) Required*
8) **Would you like to participate in other tasks?** *(Choice - Single answer) Required*

<table>
<thead>
<tr>
<th>Choice</th>
<th>Yes (%)</th>
<th>No (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes, I would like to change my tasks for more interesting ones</td>
<td>24%</td>
<td>18</td>
</tr>
<tr>
<td>Yes, I would like to have more variety of tasks</td>
<td>46%</td>
<td>34</td>
</tr>
<tr>
<td>No, I would like to keep my current tasks</td>
<td>26%</td>
<td>19</td>
</tr>
<tr>
<td>Other, please specify:</td>
<td>4%</td>
<td>3</td>
</tr>
</tbody>
</table>

- **Team/project leader**
- **I want more responsibility**
- **Increased focus on…**

9) **Given the possibility, would you prefer to switch to another product development?** *(Choice - Single answer) Required*

<table>
<thead>
<tr>
<th>Choice</th>
<th>Yes (%)</th>
<th>No (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>28%</td>
<td>21</td>
</tr>
</tbody>
</table>

10) **Given the possibility, would you like to be able to participate in more than one product development?** *(Choice - Single answer) Required*

<table>
<thead>
<tr>
<th>Choice</th>
<th>Yes (%)</th>
<th>No (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>57%</td>
<td>42</td>
</tr>
</tbody>
</table>

11) **Imagine you find a bug on an open source library/component/tool your product relies on. If management allowed you to allocate some time for this task, would you like to contribute a bug fix to this library/component/tool?** *(Choice - Single answer) Required*

<table>
<thead>
<tr>
<th>Choice</th>
<th>Yes (%)</th>
<th>No (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>47%</td>
<td>35</td>
</tr>
<tr>
<td>Maybe, depends on the library/component/tool</td>
<td>50%</td>
<td>37</td>
</tr>
<tr>
<td>No, never</td>
<td>3%</td>
<td>2</td>
</tr>
<tr>
<td>Other, please specify:</td>
<td>0%</td>
<td>0</td>
</tr>
</tbody>
</table>

**B.2.3 Tools**

12) **What is your general opinion on the tools used?** *(Choice - Single answer) Required*

<table>
<thead>
<tr>
<th>Choice</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Good</td>
<td>16%</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>41%</td>
<td>30</td>
</tr>
</tbody>
</table>
13) How would you rate your freedom to choose whatever tool you think it's appropriate to do your job? (Choice - Single answer) Required

I can choose any tool that is important to my job 11% 8
I can choose some tools, but not the important ones 65% 48
I cannot choose any tool 19% 14
Other, please specify: 5% 4

I can choose some tools, but the important ones are globally chosen with well developed support which makes it more efficient for the whole organization.
Some (also important ones)
I do not know
Not sure

14) Did you ever have the need to create a domain-specific tool for your product? (Choice - Single answer) Required

No 64% 47
Yes, once 15% 11
Yes, several times 22% 16

B.2.4 Tools (Domain specific)

This survey page was only shown to respondents who select “Yes, once” or “Yes, several times” on question 14.

15) Did you need to ask permission? (Choice - Single answer) Required

Yes 23% 6
No 77% 20

16) Were you able to allocate time for the task? (Choice - Single answer) Required

Yes, enough 64% 47
Yes, but not enough 15% 11
No, I did it on spare time 22% 16
Other, please specify: 0% 0

17) If this tool was useful for other coworkers, did you have a way to publish it or make it available to others? (Choice - Single answer)
No, it is only on my disk 4% 1
Yes, if they ask I share it 42% 11
Yes, via eForge 12% 3
Yes, it is under version control 35% 9
Yes, by other means: 8% 2

Lots of tool available via std work environment.
Mail/shared folders.

B.2.5 Way of Working

18) How much time did you need to learn your product development's way of working? (Choice - Single answer) Required

2 days or less 5% 4
1 week or less 16% 12
2 weeks or less 22% 16
1 month or less 18% 13
More than 1 month 38% 28

19) In your opinion, do you think you received appropriate training? (including presentations, training material, support, etc) (Choice - Single answer) Required

Yes 41% 30
No, too few 49% 36
No, too much 4% 3
Other, please specify: 5% 4

A lot of information presented was not really relevant
Yes as I cannot see how it could be done differently with available resources and the current work pressure.
Enough but unstructured
There were presentations, but they weren't very good

20) Did you have previous experience with any Agile methodology? (Choice - Single answer) Required

Yes 59% 43
No 41% 30

21) If yes, it helped? (Choice - Single answer)

Yes 79% 38
No 21% 10
22) Which statement reflects better your opinion on the development methodology? (Choice - Single answer) Required

<table>
<thead>
<tr>
<th>Statement</th>
<th>Percentage</th>
<th>28</th>
</tr>
</thead>
<tbody>
<tr>
<td>It helps on my job</td>
<td>38%</td>
<td>28</td>
</tr>
<tr>
<td>Some times it gets on my way, but mostly it helps</td>
<td>48%</td>
<td>35</td>
</tr>
<tr>
<td>Some times it helps, but mostly it doesn't</td>
<td>8%</td>
<td>6</td>
</tr>
<tr>
<td>It doesn't help</td>
<td>3%</td>
<td>2</td>
</tr>
<tr>
<td>Other, please specify:</td>
<td>3%</td>
<td>2</td>
</tr>
</tbody>
</table>

*It's a concoction of various different methodologies and they also change over time and depending on who you ask. Usually I disregard WoW if it gets in the way of getting things done.*

*It's not Scrum, but some faulty "agile by Ericsson"-way.*

23) Which statements reflect better your opinion about improving the development methodology? (Choice - Single answer) Required

<table>
<thead>
<tr>
<th>Statement</th>
<th>Percentage</th>
<th>33</th>
</tr>
</thead>
<tbody>
<tr>
<td>I have some improvement ideas</td>
<td>45%</td>
<td>33</td>
</tr>
<tr>
<td>There are many things to improve but I don't know how to improve them</td>
<td>26%</td>
<td>19</td>
</tr>
<tr>
<td>There isn't much to improve but I feel like I can make it better</td>
<td>14%</td>
<td>10</td>
</tr>
<tr>
<td>There isn't much to improve neither I know how I could make it better</td>
<td>5%</td>
<td>4</td>
</tr>
<tr>
<td>There is nothing to improve</td>
<td>0%</td>
<td>0</td>
</tr>
<tr>
<td>Other, please specify:</td>
<td>10%</td>
<td>7</td>
</tr>
</tbody>
</table>

*There is things which have to be defined*

*I have a lot of ideas, and a continuously applies them*

*You should always try to improve the current processes, no process works forever*

*Don't know*

*I do not know*

*It's hard to do improvements, too many areas shall approve. Cost too much effort.*

24) If you have some improvement ideas, can you influence (and eventually change) the way of working? (Choice - Single answer) Required

<table>
<thead>
<tr>
<th>Statement</th>
<th>Percentage</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>No and I shouldn't</td>
<td>7%</td>
<td>5</td>
</tr>
<tr>
<td>No and I should</td>
<td>14%</td>
<td>10</td>
</tr>
<tr>
<td>Yes, but just minor things</td>
<td>48%</td>
<td>35</td>
</tr>
<tr>
<td>Yes, I really think my opinion is taken into account</td>
<td>27%</td>
<td>20</td>
</tr>
<tr>
<td>Other, please specify:</td>
<td>4%</td>
<td>3</td>
</tr>
</tbody>
</table>

B-6
Of course - good arguments are always welcome and listened to - it’s all about communication!

The manager doesn’t listen to us who is doing the actual work

I can influence the WoW but there are other things that I find more important/interesting.

B.2.6 Decision making

25) When it comes to decisions that affect your daily work, do you think they would be better if you have participated in the decision making? (Choice - Single answer) Required

<table>
<thead>
<tr>
<th>Option</th>
<th>Percentage</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>I participate in all decision making that affects my daily work</td>
<td>14%</td>
<td>10</td>
</tr>
<tr>
<td>No, I think I shouldn't participate in all that decision making</td>
<td>25%</td>
<td>18</td>
</tr>
<tr>
<td>Yes, in some cases (you can specify below)</td>
<td>37%</td>
<td>27</td>
</tr>
<tr>
<td>Yes, almost always</td>
<td>25%</td>
<td>18</td>
</tr>
</tbody>
</table>

26) This optional field lets you write additional comments on question above: (Open ended text - Essay) | View

It’s important that the right people influence each decision. The trend is all employees or no one which are both bad.

What tools to be used in Testing and when there are several ways of implementing a functionality, together with system management can choose the best option

In the tools, working environment, communication between teams etc

I think too many decisions are centralized towards management and there is no trust towards the engineers/teams to work in the way they see most efficiently.

Managers who doesn’t understand software development is the ones who is developing a development methodology. It will always fail

Concerning processes: Often the process is decided by the upper management ("this is how it should be done"), but without the possibility of affecting the outcome (even when we're actually trying to adhere to the process; "according to the process it should be this way... yes we know, you're right, but...").

Decision making is up to others but the input and improvements should come from us too.

As we work as subcontractors, our work is pretty constrained by the job description.
## B.2.7 Communication

### 27) In your opinion, which of the following statements, regarding information exchange between development stages (requirements, design, testing, etc), are true? *(Choice - Multiple answers) Required*

<table>
<thead>
<tr>
<th>Statement</th>
<th>Percentage</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>I receive too many small documents. If information was more compact I would have to read less</td>
<td>15%</td>
<td>11</td>
</tr>
<tr>
<td>Usually, I don't receive all information I need</td>
<td>38%</td>
<td>28</td>
</tr>
<tr>
<td>I wish the information I need would be easier to retrieve</td>
<td>62%</td>
<td>45</td>
</tr>
<tr>
<td>The documents I need to work with are usually unclear or incomplete</td>
<td>41%</td>
<td>30</td>
</tr>
<tr>
<td>The documents I need to work with change too often</td>
<td>15%</td>
<td>11</td>
</tr>
<tr>
<td>The documents I need are spread among too many places</td>
<td>37%</td>
<td>27</td>
</tr>
<tr>
<td>The information exchange between processes is too bureaucratic</td>
<td>23%</td>
<td>17</td>
</tr>
<tr>
<td>The information exchange between processes is too complex</td>
<td>19%</td>
<td>14</td>
</tr>
<tr>
<td>The information exchange between processes is too slow</td>
<td>25%</td>
<td>18</td>
</tr>
<tr>
<td>The tools we use for information exchange are not adequate</td>
<td>34%</td>
<td>25</td>
</tr>
<tr>
<td>None</td>
<td>10%</td>
<td>7</td>
</tr>
<tr>
<td>Other, please specify:</td>
<td>1%</td>
<td>1</td>
</tr>
</tbody>
</table>

### 28) You can add additional comments about the information exchange here: *(Open ended text - Essay)*

Due to agile way of working, sometimes the documents are not complete by the time the development starts and requirements are unstable.

Still too much reliance on transient documents.

Again, the process we should work by, fails.

I’m so used to using a Wiki to store information and use hyperlinks a lot.

Slow process between requirements analysis and design, would be good design team participates in earlier phases.

Use case oriented requirements instead of too much detailed ones.

Delivery reports are not efficient and too costly for developer’s team, too bureaucratic.

As developer I do not work so much with documents so I do not have so much opinion about this topic.

CDM is one of the worst document management systems I've ever seen. It's almost impossible to find a document if you don't know the exact name or document number. Fuzzy matching (match similar words, non case sensitive, etc) should be required.

It is also difficult to create new documents in CDM. I wish there were tailored "wizards" for the projects I work with that guided me through the creation of
a document in a safe and correct way.
The information is also spread into a bunch of different tools, CDM, Mars, PRIM/GASK, CPIStore, ClearCase, WIKI(s), just to name a few, which makes it very hard to find information when you are not 100% sure of what to look for and where.

I would like to see better search tools like Google mini or something similar. This to get search results from all databases like CDM, Wiki etc

**29) What about the meetings?** *(Choice - Single answer) Required*

Meetings are effective 18% 13
Some are effective, some are not 77% 56
Meetings are not effective 5% 4

**30) What about the number of meetings?** *(Choice - Single answer) Required*

Too few meetings 1% 1
Appropriate number of meetings 78% 57
Too much meetings 19% 14
Other, please specify: 1% 1

*Often bad prepared. Almost just a slogan and a bunch of people.*

**B.2.8 Documentation**

**31) Rate from 1 to 4 your level of agreement with the following statements:**

*The project's documentation...* *(Matrix - One answer per row) Required*

<table>
<thead>
<tr>
<th></th>
<th>1 Agree</th>
<th>2</th>
<th>3</th>
<th>4 Disagree</th>
</tr>
</thead>
<tbody>
<tr>
<td>is well structured</td>
<td>14% (10)</td>
<td>47%</td>
<td>29%</td>
<td>11% (8)</td>
</tr>
<tr>
<td>is easy to find</td>
<td>8% (6)</td>
<td>34%</td>
<td>36%</td>
<td>22% (16)</td>
</tr>
<tr>
<td>is up to date</td>
<td>14% (10)</td>
<td>53%</td>
<td>27%</td>
<td>5% (4)</td>
</tr>
<tr>
<td>is complete</td>
<td>8% (6)</td>
<td>59%</td>
<td>19%</td>
<td>14% (10)</td>
</tr>
<tr>
<td>its format is adequate</td>
<td>27% (20)</td>
<td>53%</td>
<td>14%</td>
<td>5% (4)</td>
</tr>
<tr>
<td>is correct</td>
<td>21% (15)</td>
<td>59%</td>
<td>19%</td>
<td>1% (1)</td>
</tr>
</tbody>
</table>

**32) How often do you voluntarily contribute to the collaborative documentation (Wiki, EriColl, How-to’s, etc)?** *(Choice - Single answer) Required*

Never 12% 9
Once 15% 11
At least once a month 40% 29
Once a week 19% 14
Several times a week 14% 10

**33) Final comments:** *(Open ended text - Essay)*
Nice work!

No Comment :)

This organization is to complex and slow. The way of implementing scrum and calling it "Agile by Ericsson" is by far the worst example I've seen. Teams are told to do fixed amount of work in fixed amount of time without the possibility to influence this themselves. Teams are told to be self organizing, but still need a team leader that is supposed to be responsible and will be replaced if the team does not perform according to expectations. More responsibility/trust should be released to engineers/teams so that we get competent engineers with a feeling of responsibility and an urge to improve things.

Good luck and I hope Ericsson will learn something as well from your work! :)

I strongly believe we should educate and encourage developers to refactor code they modify in line of duty, lest the code deteriorates and the technical debt increases. I'm not talking about big restructuring, but fixing something every time you commit.

Good job :)  

B.3 Analysis

The results of this survey were very useful to both understand developer’s predisposition to some of our planned adoption opportunities and to detect main discontent issues with the current methodology. Without the opinions of the biggest group of methodology practitioners, the developers, our view would have been incomplete.

In the following sections, we will expose the main conclusions that we extracted from the survey results above.

B.3.1 Profile

The survey was open from March 31st to April 15th. During these 22 days, a total of 92 practitioners visited the survey and 79% of them (73) completed it. Considering that our survey population was 130 practitioners (90 from product A and 40 from product B) we are very satisfied with the response rate.

Our intended audience was the designer / developer / tester role. As the survey invitations were delivered by mail to all members of the selected projects we needed a mechanism to make sure we were hitting our target. Questions 1 and 2 were written with this purpose. We can see that all but four respondents selected at least one design / development / testing related role.

Regarding Figure B.1, it is relevant to note that on product A, the majority of people have been there since they joined Ericsson, while product B, which is much younger, gathers people from more diverse development backgrounds.

B.3.2 Tasks

The purpose of the task related questions was to detect a generalized desire of the respondents to participate in more diverse tasks. In this sense, the answers were very satisfactory.

While 73% of the respondents agreed that their assigned tasks matched both their background and skills, a 43% stated that they would enjoy more variety. Similarly,
while 72% of the respondents said that they would not change their current project if given the opportunity, a remarkable 57% of them would like to participate in more projects at the same time. This willingness to participate in more than one development extends even to open source projects outside Ericsson, as 97% stated that they would consider contributing.

This provided us with substantial base to argument that the developers would enjoy adoption opportunity number six, that is, to be able to choose some of their tasks from a bigger pool of available ones.

B.3.3 Tools

When asked about their satisfaction with the tools provided by Ericsson to do their jobs, respondents agreed (84%) that they did not have enough freedom. However, the average perception on the quality of the tools is good.

Regarding the domain specific tools, that is, tools created inside Ericsson for project specific purposes; an interesting issue was revealed. While few needed to ask permission to start the development of a custom tool, the majority was unable to allocate enough time (23%) or even used their own spare time for the task (38%). This, combined with our research on the requirements engineering aspects of Streamline, showed that emergent requirements from the development were not considered and thus received no allocation of time. This had some serious side effects, mainly the accumulation of technical debt and that these tasks silently compete with planned tasks for time, out of sight from project managers’ planning and control.

B.3.4 Way of working

Respondents expressed a generalized satisfaction with the current methodology (52%), with a 45% considering that if mostly it helped it sometimes got on their way. When asked about their ability to influence the way of working, 56% stated that they had some improvement ideas and 31% that their opinion was taken into account. However, 52% of them considered that they could just influence minor things. Regarding the level of decision making they desired, the situation was quite even, with 42% considering that they had enough decision making power and 58% considering that they should have more.

B.3.5 Communication and documentation

The communication related questions revealed that practitioners considered the information availability insufficient in many ways. They mostly agreed on that they didn’t receive all information they need (31%) and that it was usually hard to retrieve (59%), unclear and incomplete (31%).

This supported our thesis on that a change on information availability was needed. Adoption opportunities three and four were formulated to contribute a solution to this issue.