SOFTWARE DEVELOPMENT PRODUCTIVITY

EVALUATION AND IMPROVEMENT FOR LARGE INDUSTRIAL PROJECTS

Piotr Tomaszewski

Blekinge Institute of Technology
Doctoral Dissertation Series No. 2006:05

School of Engineering

Software Development Productivity

Evaluation and Improvement for Large Industrial Projects

Piotr Tomaszewski

Blekinge Institute of Technology Doctoral Dissertation Series
No 2006:05
ISSN 1653-2090
ISBN 91-7295-090-0

Software Development Productivity

Evaluation and Improvement for Large Industrial Projects

Piotr Tomaszewski

Department of Systems and Software Engineering
School of Engineering
Blekinge Institute of Technology
SWEDEN

© 2006 Piotr Tomaszewski
Department of Systems and Software Engineering
School of Engineering
Publisher: Blekinge Institute of Technology
Printed by Kaserntryckeriet, Karlskrona, Sweden 2006
ISBN 91-7295-090-0

To Gosia

Abstract

Software development productivity can be improved by introducing improvements in many areas. In this thesis we investigate technology and process driven productivity improvements, i.e., productivity improvements that have sources in changes of technologies or in changes in development processes. The technology driven productivity improvement discussed in this thesis is the change of server platform from a standard general purpose platform to a specialized fault-tolerant platform. We discuss productivity implications of introducing such a platform as well as suggest ways of making the platform introduction process cost efficient. The process changes, which we discuss in this thesis, include improvements of fault detection processes as well as changes of the entire development process.

We analyze the implications of introducing new technology by performing case studies, in which we describe, analyse, and quantify the impact of the new platform on software development productivity. We show that there is a significant productivity decrease connected with introducing a new platform. We also show that the initial low productivity can be overcome by experience and maturity. We suggest a number of improvements for both the platform introduction process and the mature development on the specialized platform. Since some productivity decrease after introducing new technology is to a large extent unavoidable, we look for ways of minimizing it. We show that it is possible to minimize it by introducing the specialized platform gradually. We present an example of a hybrid architecture, which combines the specialized and the standard platforms. We show that such architecture is able to provide good technical characteristics for a significantly lower cost as compared to developing the entire application on the specialized platform.

As a process improvement suggestion we propose introducing fault prediction models with the goal of increasing the efficiency of fault detection. We suggest and evaluate several such models that are available at different stages of a software development process. The models are evaluated using data from a number of large software systems. Their predictions are also compared with the predictions made by human experts. We show that introducing our fault prediction models is likely to result in an improvement of fault detection efficiency. Another process related productivity improvement suggestion evaluated by us is the change of the development process. We present a case study in which we evaluate a new process concept. One of the goals of that process is to improve the company’s productivity.

Acknowledgements

First and foremost, I would like to express my sincere gratitude to my supervisors, *Professor Lars Lundberg* and *Dr. Håkan Grahn* for their patient guidance, invaluable comments, and support at every stage of my research education. Without their help and encouragement this thesis would never have been completed.

All the work presented in this thesis was possible only because of the close and fruitful co-operation with *Ericsson*. I would like to take this opportunity and thank all the members of the *Ericsson staff* who kindly took part in my studies and provided me with the input to my work. Especially, I would like to thank *Lars Angelin, Dr. PerOlof Bengtsson*, and *Sven Jonasson* for their contribution and valuable comments. My great thanks go to *Jim Håkansson*, not only for answering tones of my questions but also for co-authoring some of the papers.

I also would like to thank all my *colleagues* from *Blekinge Institute of Technology*, especially from the *PAARTS* group and the *BESQ* project, for creating a unique, both enjoyable and motivating work environment. In particular, I would like to thank *Dr. Daniel Häggander*, *Lars-Ola Damm,* and *Patrik Berander* for many hours of discussions and for joint work on some of the papers, *Dr. Johan Schubert*, *Dr. Mirosław Staroń, Jeanette Eriksson, Lawrence Henesey,* and *Simon Kågström* for their valuable comments on different pieces of the work presented in this thesis, and *Charlie Svahnberg* for being my source of advice on teaching.

As always, I am indebt to my *family* and *friends* for all the help I got from them. I am grateful to my *parents, Halina* and *Andrzej,* and to my *brother Dominik* for their constant support.

Finally, I would like to thank my *wife Gosia*. It is your never-ending optimism, determination, enthusiasm, and encouragement that helped me enormously during my studies and made this thesis possible. Thank you!

This work was partly funded by The Knowledge Foundation in Sweden under a research grant for the project "Blekinge - Engineering Software Qualities (BESQ)" (http://www.bth.se/besq).

List of Papers

Papers included in this thesis:

I. “Software Development Productivity on a New Platform - an Industrial Case Study”
Piotr Tomaszewski, Lars Lundberg,
Information and Software Technology journal, vol. 47/4, 257-269, 2005

II. *“*The Increase of Productivity Over Time - an Industrial Case Study”
Piotr Tomaszewski, Lars Lundberg
Information and Software Technology journal, vol. 48/9, 915-927, 2006

III. “Evaluating Real-time Credit-control Server Architectures Implemented on a Standard Platform”
Piotr Tomaszewski, Lars Lundberg, Jim Håkansson, Daniel Häggander
Proceedings of IADIS International Conference on Applied Computing, vol. 2, 345-352. Algarve, Portugal, February 2005

IV. “A Cost-efficient Server Architecture for Real-time Credit-control”
Piotr Tomaszewski, Lars Lundberg, Jim Håkansson, Daniel Häggander
Proceedings of the 10th IEEE International Conference on the Engineering of Complex Computer Systems (ICECCS), pp.166-175, Shanghai, China, June 2005

V. “Improving Fault Detection in Modified Code - A Study from the Telecommunication Industry”
Piotr Tomaszewski, Lars Lundberg, Håkan Grahn
To appear in: Journal of Computer Science and Technology, Special Issue on Advances in Software Metrics and Software Processes, 2006

VI. “A Method for an Accurate Early Prediction of Faults in Modified Classes”
Piotr Tomaszewski, Håkan Grahn, Lars Lundberg
To appear in: Proceedings of the 22nd IEEE International Conference on Software Maintenance (ICSM 2006), Philadelphia, USA, September 2006

VII. “Statistical Models vs. Expert Estimation for Fault Prediction in Modified Code – an Industrial Case Study”
Piotr Tomaszewski, Jim Håkansson, Håkan Grahn, Lars Lundberg
Submitted to a journal. This paper is an extended version of Paper XIII.

VIII. "Comparing the Fault-Proneness of New and Modified Code – An Industrial Case Study"
Piotr Tomaszewski, Lars-Ola Damm
To appear in: Proceedings of the 5th ACM-IEEE International Symposium on Empirical Software Engineering (ISESE 2006), Rio de Janeiro, Brazil, September 2006

IX. "From Traditional to Streamline Development - Opportunities and Challenges"
Piotr Tomaszewski, Patrik Berander, Lars-Ola Damm
To be submitted to a journal.

Publications that are related but not included in this thesis:

X. "Evaluating Productivity in Software Development for Telecommunication Applications"
Piotr Tomaszewski, Lars Lundberg
Proceedings of IASTED International Conference on Software Engineering, 189-195, Innsbruck, Austria, February 2004

XI. "The Accuracy of Early Fault Prediction in Modified Code"
Piotr Tomaszewski, Lars Lundberg, Håkan Grahn
Proceedings of the 5th Conference on Software Engineering Research and Practice in Sweden (SERPS), pp.57-63, Västerås, Sweden, October 2005

XII. "Increasing the Efficiency of Fault Detection in Modified Code"
Piotr Tomaszewski, Lars Lundberg, Håkan Grahn
Proceedings of the 12th Asia-Pacific Software Engineering Conference (APSEC), pp. 421-430, Taipei, Taiwan, December 2005

XIII. "The Accuracy of Fault Prediction in Modified Code –Statistical Model vs. Expert Estimation"
Piotr Tomaszewski, Jim Håkansson, Lars Lundberg, Håkan Grahn
Proceedings of the 13th Annual IEEE International Conference on the Engineering of Computer Based Systems (ECBS 2006), pp. 334-343, Potsdam, Germany, March 2006. The paper was awarded with the Best Paper Award.

Table of Contents

Paper II
The increase of productivity over time - an industrial case study

Paper VIII
Comparing the Fault-Proneness of New and Modified Code – An Industrial Case Study

Paper IX
From Traditional to Streamline Development – Opportunities and Challenges

Introduction

1. Background

In the era of globalisation, outsourcing, and fierce competition between different companies that produce software, the software development productivity issues are becoming increasingly important. Robert L. Glass notices in “The Realities of Software Technology Payoffs” [32], that productivity, from being a very hot topic in 80’s, lost some of its momentum in 90’s, when more emphasis was put on software quality. The IT crisis following Y2K made software developers very cost-aware and brought some of the attention back to the productivity issues.

Productivity is defined as a ratio of output units produced per unit of input effort [1]. Software development productivity describes how efficiently the software is produced. Higher productivity means faster and/or cheaper software development. Therefore, productivity improvement is often seen as a way of gaining market competitiveness by decreasing the cost of developing software, increasing return on investment, and improving time-to-market. High productivity largely facilitates, if not makes it possible, to quickly satisfy changing customer needs and still make money. Such abilities are what nowadays distinguish successful software companies from the less successful ones.

The productivity can be increased by improvements in three areas [37]: people, processes and technology. The people issues concern aptitude, competence, domain knowledge and experience. Better educated and experienced developers are able to deliver the software faster. Process issues concern work organization and work environment. Efficient work procedures save resources and make the development more productive. An appropriate technology makes it possible to deliver a product using less effort. The tools can take part of developer’s responsibility and automatically generate the code, support testing or improve the communication between the team members. Off-the-shelf software components and systems (e.g., database servers) largely facilitate development of certain kinds of software systems. All this technology support saves time and thus improves the development productivity.

The people, process, and technology issues are not independent. Technology will not improve the productivity if the work processes are not adapted to it. High skills of the developers do not contribute to the productivity if they are not provided with the technology they are skilled in. A technology does not improve the productivity if people do not know how to use it. Therefore, it is very important to evaluate any change in the development from the productivity perspective. It is rarely so that a people, process or technology change, is a “silver bullet” solution from the productivity point of view. It often depends on numerous circumstances if the overall productivity gains or looses because of the change. For example, decreasing the number of code reviews can increase the productivity. Relatively more time is spent on the code production compared to the other activities. However, the overall productivity may decrease if because of the decreased number of the code reviews the developers produce more faults that have to be removed.

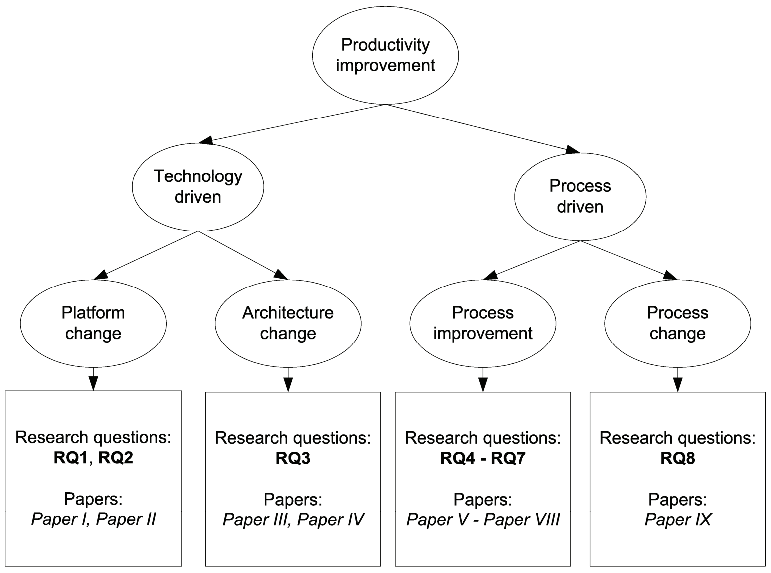
In our studies we evaluate a number of different productivity improvements. Some of those improvements were suggested by our industrial partners, some were identified in literature, other ones are our own suggestions. We focus on the technology and process driven productivity improvements, i.e., productivity improvements that are based on changes in technologies used for developing software systems and changes in software development processes. The technology change, which we evaluate, is the change of the server platform from a standard, general purpose one to a specialized fault-tolerant solution. The process changes suggested and/or evaluated by us include the improvement of fault detection process efficiency by introducing statistical fault prediction models as well as productivity improvement by a change of an entire development process.

The reminder of the *Introduction* section is structured as follows. Section 2 presents our specific research questions. In Section 3 we present related research. In Section 4 we describe the major contributions from this thesis. Section 5 focuses on research methodology issues. In Section 6 we present major conclusions from our work.

2. Research questions

The overall question in our studies concerned the ways productivity can be improved in software development projects. We were interested in finding and evaluating productivity improvements in different aspects of software development. As mentioned in the previous section, the productivity improvements can be classified as people, processes, and technology related [37]. The two areas of productivity improvements that are considered in this thesis are productivity improvements that are technology driven and productivity improvements that are process driven (see Figure 1).

Figure 1. Thesis outline. Mapping between topics, research questions and papers.


Productivity improvement
Technology driven
Process driven
Platform change
Architecture change
Process improvement
Process change
Research questions: **RQ1**, **RQ2**
Papers: *Paper I, Paper II*
Research questions: **RQ3**
Papers: *Paper III, Paper IV*
Research questions: **RQ4 - RQ7**
Papers: *Paper V - Paper VIII*
Research questions: **RQ8**
Papers: *Paper IX*

The technology change evaluated in our studies was the *change of server platform* from a standard, general purpose UNIX platform to a specialized fault-tolerant one. The context of the change was software development of telecommunication applications. The developers of telecommunication applications face rather stringent requirements concerning the reliability, performance, and availability of the systems they develop. A fault-tolerant platform can facilitate the development of such systems by taking at least partial responsibility for providing the required characteristics. However, introducing new platforms, as well any other technology, is likely to result in an initial productivity decrease due to the need of overcoming the learning curve connected with gaining competence in the new technology [33]. Therefore, we looked for some *architecture changes* that could decrease the negative impact of introducing new technology on the productivity of software development in which this new technology is used.

The process changes evaluated in our studies concerned the improvement of fault detection process efficiency as well as change the of an entire development process. The *process improvement* suggested and evaluated by us was the improvement of the efficiency of the fault detection process by introducing statistical fault prediction models. Such models, by predicting the most likely locations of faults, can potentially indicate where fault detection activities are likely to be most efficient. Another productivity improvement evaluated was a *process change*. It turned out that one of the significant productivity problems of our industrial partner was connected with changing customer requirements. A new process that is supposed to minimize this problem was suggested. We performed an early evaluation of this process.

In the reminder of this section we present the concrete research questions posed in our studies. Section 2.1 presents our research questions regarding the technology driven productivity improvement. Section 2.2 presents our research questions regarding the process driven productivity improvement. The numbering of the research questions corresponds to the numbering of research questions in Figure 1.

2.1 *Technology driven productivity improvement*

2.1.1 *Platform change*

The first productivity improvement evaluated by us was the introduction of a new technology. The new technology was a specialized fault-tolerant platform that was meant to make it possible to meet the high non-functional requirements posed on the telecommunication applications, like real-time performance, high availability and reliability. We anticipated that the productivity of software development for that platform should increase with time, when a certain experience is gained by the developers. However, the early software development productivity is also important since it is important to know what to expect in a short term from introducing a new platform. Therefore the first specific research question was:

***RQ1**: How does the introduction of a fault-tolerant platform impact productivity:*

- *just after the new technology adoption?*
- *when a certain maturity is gained?*

Once we evaluated the productivity we have focused on the improvements to the technology adoption process as well as on the improvements of the subsequent, more mature development on the new platform. We looked for the improvements in the ways the platform is used (e.g., the platform introduction process, the work organization, the development of platform related competence) as well as improvements of the platform deficiencies that impact productivity. To investigate the productivity improvement possibilities we formulated the second specific research question:

RQ2: *How can the productivity be improved in projects in which the fault –tolerant platform is involved:*

- o *by improving the way the platform is used?*
- o *by improving the platform itself?*

2.1.2 *Architecture change*

As the literature says it is reasonable to expect some productivity decrease when a new technology is adopted ([24, 29, 33, 38, 41]). Therefore, it is also reasonable to expect a higher cost of the software development in the first projects on the new platform. The cost of the projects can potentially be lowered if we use a hybrid platform that combines a fault-tolerant and a standard platform, instead of using the fault-tolerant platform only. The fault-tolerant platform can help providing good non-functional characteristics, while the standard platform can reduce the development cost. Therefore, our next specific research question was:

RQ3: *How can the fault-tolerant and the standard platforms be combined to provide the required non-functional characteristics and at the same time satisfactory development productivity?*

2.2 Process driven productivity improvement

2.2.1 *Process improvement*

Productivity improvements can also be achieved by introducing changes to the way the work is performed. It is commonly known that fault detection and removal activities constitute a significant part of a software development project [33]. Therefore, we looked for methods that can improve the efficiency of fault detection. Since faults are rarely distributed evenly in a system [11], one common idea is to build a fault prediction model that identifies the most fault-prone code and makes it possible to direct fault detection activities to where they are likely to be most efficient, i.e., to the code that is most likely to contain faults. We evaluated this approach from the perspective of increased fault detection efficiency. Our specific research question was:

RQ4: *What fault detection efficiency improvement can be expected from applying our fault prediction model?*

The prediction of fault proneness of the individual code units is based on the characteristics of those code units. The most complete and correct set of characteristics is available after the code units are actually implemented. However, there is an added value of having such fault predictions earlier in the development process, as that enables an efficient planning of some preventive measures that can reduce the number of faults in the code. Therefore, our next specific research question was

RQ5: *How to obtain accurate fault predictions early in the development process?*

Fault prediction models, as those suggested above; seem to be still more popular in academia than in industry. It also seems that an industrial standard for performing fault predictions is expert estimation. Therefore, our next goal was to compare the performance of our models with the performance of experts predicting fault localisations. For that reason, our next research question was:

RQ6: *Are statistical fault predictions more accurate than estimations done by experts?*

In our fault prediction studies we focused on modified code, mainly because in the projects we used a majority of faults was found in the modified code. However, this does not indicate that in a general case the modified code units are more fault-prone than newly developed ones. To further investigate this issue we formulated the following research question:

RQ7: *Is modified code more fault-prone than new code?*

2.2.2 Process change

Productivity improvement can also be achieved by a more radical change, e.g., by changing the entire development process. One commonly suggested process change that, apart from other benefits, is supposed to improve productivity of software development is the change from large long-lasting projects to smaller ones. The productivity improvement in such smaller projects is usually attributed to the fact that smaller projects are less exposed to the change of market demands, which is one of the most common sources of rework and waste in large projects. One project concept of this kind is Streamline Development, a process concept developed by Ericsson. In our studies, we evaluated this concept. Our specific research question was:

RQ8: *What are the opportunities and challenges when changing from traditional software development to Streamline Development?*

3. Related work

3.1 *The software development productivity improvement process*

The productivity improvement process consists of three steps [82]: measurement, analysis, and improvement. The role of the measurement is to find what the current productivity level is. Productivity measurement is also necessary to assess productivity improvements in the future. The analysis aims at finding factors that affect the productivity. The role of the improvement is to increase the overall productivity. The first step, productivity measurement, is considered the basis for the two remaining steps [82]. We describe the related work concerning productivity measurement topic in more detail in Section 3.1.1. The related work concerning productivity analysis and improvement is described in Section 3.1.2.

3.1.1 Productivity measurement

Productivity measurement research accounts for a large part of the productivity related research [56]. Productivity, apart from being an interesting metric in itself, is also a factor that must be known to perform numerous project related estimations, like cost [71] or lead-time [9, 10, 78] predictions. Since these two metrics are very important for many crucial decisions in the project, the topic of the productivity measurement has been in focus for quite a long time.

Despite a relatively simple equation (*product size/development effort*) and an easy-to-grasp meaning, the application of the productivity metric to software development is not straightforward and standardized. Therefore, as the literature points out [27, 55, 71], we must be very careful when comparing productivity between different projects. The important thing is to assure that we compare the same things [55]. For example, in one project the effort metric may include only the hours spent by the designers and testers, while in the other one it may contain the work hours of designers, testers, managers, and technicians. Comparing the productivity of these two projects using their understanding of the effort will not give any meaningful results. Another important thing to remember is that the productivity metric does not take quality aspects into account [27]. Therefore, when we compare the productivity, we should always keep the quality in mind, since the productivity can only be "interpreted in context of overall quality of the product" [1]. Lower productivity might be the price for a better, faster, more usable and easier to maintain product [27].

The two values that compose the productivity equation are product size and development effort. As the *development effort* usually the total number of person-hours or person-months spent on the project is used [27, 71]. Another approach could be to use the cost; however such a metric is often less adequate. The cost of work-hour varies among companies and changes in time. The *product size* metrics are usually divided into two groups [71]:

- *size-related metrics* – that describe a physical size of the code delivered. The most typical metrics in this class are the number of source lines of code (SLOC), the number of delivered instructions, classes, functions, files, etc.
- *functionality-related metrics* – that reflect the amount of functionality delivered. Examples of the metrics used to describe the functionality are different variations of function points, like IFPUG, Mark II, Feature Points, or COSMIC [27, 55, 71, 76].

There has been a lot of discussions concerning the ability of size-related metrics, like SLOC, to capture the project size [27, 56, 71]. Lines of code have some obvious drawbacks. They are incomparable when different programming languages are used [27]. It is hard to say in what way 100 lines of C++ code correspond to 100 lines in Assembler code. The number of lines of code also depends on the coding style. A more compact coding style results in smaller projects. Finally, the lines of code metric does not reflect the complexity and the utility of the program [27]. 100 lines of C code used for implementing a real-time system are definitely not the same as 100 lines of code used to implement some simple searching algorithm with no performance requirements.

Despite these obvious drawbacks the lines of code are widely used in practice [9, 10, 56, 70, 82]. According to [27] their biggest advantages are computational simplicity (values can be obtained automatically) as well as tangibility – it is very easy to understand and explain what a single line of code is, even though it is actually not well defined. There is an on-going discussion concerning the use of SLOC for productivity measures. Some researchers discard SLOC as an inaccurate metric [51]. There are, however, other opinions as well. In [73] it was observed that lines of code can be applicable when two projects were written in the same language. In [56] the authors compared lines of code productivity with *Process Productivity*, a complex metric that involved many aspects like a programming language, experience, management practices etc. They concluded that SLOC was actually a better metric.

The *functionality-related metrics* overcome the drawbacks of the SLOC measurement by measuring the size of the software as the amount of functionality it delivers. Although there exists a number of different functionality related metrics (see [53] for an overview), according to [71] the most popular functionality-related metric are function points. Since the function points measure program utility instead of length they are not affected by the different expressiveness of languages, which results in different amount of SLOC necessary to deliver the same functionality [27]. The function points are calculated as a combination of the amount of data that is manipulated, the number of interfaces, the amount of interactions with user as well as external inputs and outputs [51, 71]. Since, obviously, the complexity of these entities may vary the function points method introduces a complexity factor to compensate for this.

The complexity factor is often considered as one of the weaknesses of this method, since it adds some subjective assessment to the method. The function points are also considered quite biased towards applications with extensive data processing [71] and, therefore, not applicable for systems with a simple data processing but high non-functional requirements, like real-time systems [76]. There are some variants of function points that are said to overcome that problem (e.g., COSMIC FP [14]). All function point related measures suffer from computational complexity (*counting must be done manually*) and lack of tangibility (*what does it mean that the software has the size of 10 function points*?) [27, 71]. Because of the complexity, and the need for making estimations, the results of function point measurements depend on who performs the measurement and are more accurate and consistent when the measurement is performed by a trained individual (see [13] for an overview of function point variability studies).

3.1.2 Productivity analysis and improvement

The second step of the productivity improvement is the identification of the factors affecting productivity. The identification of the factors affecting the productivity is actually the second main direction of software productivity research [56]. As could be expected, the productivity of software development is affected by almost everything. It is well illustrated by the research, in which the impact of different factors on productivity is analysed Factors analyzed in these studies range from the use of specific tools (e.g., CASE tools in [22]), through the development practices [52], to the amount of office space available to the programmer [45]. Some researchers tried to classify issues that affect the development productivity, e.g., in [37] the productivity improvements are grouped into people, processes, and technology issues.

In [71], the most important factors affecting the productivity of individuals working in an organisation are summarized. The first factor is the application domain experience. Usually the most productive members of software development teams are those with the best domain knowledge. The second important factor is the process quality. Certain process optimizations and improvements, but also inefficiencies, can affect productivity. Another important factor according to [71] is the size of a project. The rationale behind this factor is that in bigger projects relatively less time is spent on development and more time is spent on communication between team members. This reduces the productivity of individual software developers. Technology support is considered as the next important factor. CASE tools and configuration management tools are presented as examples of tools that improve productivity. Finally, the impact of working environment is discussed. According to [71] a more comfortable work place is likely to increase the individual productivity of a software developer.

In [82], another set of factors affecting productivity was identified. Additionally, the authors classified each factor according to its “level of impact” on productivity as High, Medium, or Low. To the issues that have high impact on productivity the authors classified feature requirement completeness and stability. Stable and well defined requirements make it possible to avoid unnecessary rework or waste. Another factor with high impact on productivity is feature interaction complexity. The more complex the application is the lower the productivity is likely to be. The authors also mention staff experience and feature development environment (tools, etc.) as factors having large impact on productivity. The impact of these factors has already been discussed before in this section. As factors having medium impact on productivity feature hardware application novelty and change, and software architecture impact were classified. Developers are rarely very experienced in using new technology so usually some time is needed for them to master it and become productive (this issue is discussed in more detailed in Section 3.2). The authors of [82] also believe that certain architectural solutions can have impact on productivity. Finally, as issues having low impact on productivity, the following factors were classified: feature novelty and synergy with other features, feature program complexity, static and dynamic data impact, feature performance constraints, and work environment. An interesting factor in this list is the performance requirement issue, which indicates that some software quality related requirements can have impact on productivity.

The productivity improvement is also often discussed in the context of reducing the development time. Some examples of productivity improvement areas can be found in the work devoted to identification of lead-time reduction opportunities. For example, in [9] the authors discuss eleven techniques that result in the reduction of development time. They suggest using prototyping, improving customer specifications, using CASE tools, introducing concurrent development, improving quality of development to reduce rework and recoding, improving project management, having better testing strategies, reusing code, standardizing interfaces between modules, improving communication between team members, and finally hiring better people. These techniques, by reducing the development time, should also result in improved productivity.

3.2 Technology driven productivity improvement

Software development productivity is also affected by the kind of technology that is available to developers [9, 71, 82]. Therefore, technology change is a valid way of improving productivity. Technology can improve software development productivity in many ways. CASE tools can generate code and thus decrease the effort necessary to develop the software. Database systems can provide efficient ways of manipulating and storing data and remove the necessity of implementing these functionalities in an application. Integrated development environments (IDEs) can make coding and debugging processes faster and more productive.

In our studies we examine technologies that are meant to improve productivity in the development of applications with high availability requirements. In practice, to meet these requirements we must introduce redundancy to the system both at the hardware and the software level. Whenever one of the components (hardware of software) fails, the failover operation is performed and the redundant, backup component takes over [63]. Ideally, the backup component should have the same state as the primary one. Therefore, virtually every state updating operation performed by the primary component must be replicated to the backup one. This frequent replication creates problems with extensive communication between the primary and the backup components which affects the performance negatively [5, 8, 19, 30, 39, 81]. Therefore, implementing efficient replication is a rather complex task. In [35], the authors found that in one dual-computer cluster system 85% of the code was devoted to providing the availability. This value is application specific, but it gives an indication of how complex the task of the synchronization can be. Therefore, a lot of effort has been devoted to find technical solutions that remove as much of this burden from the developer as possible. Examples of such a solution is described in [54], where a CORBA implementation was extended with abilities to perform automatic object replication, migration and restart of objects that failed. Such a technology is able to take at least partial responsibility for providing availability. By using it the developers are able to build their applications using less work and thus improve the overall productivity in the project.

Unfortunately, even though technology change can bring productivity improvement, usually it takes time before the improvement is visible [33]. Normally, in a short term some decrease of productivity can actually be expected [33]. A part of our studies is devoted to quantifying the impact of introducing new technology on software development productivity and identifying the productivity problems in the situation of new technology adoption. According to [29] there are two major reasons why companies should be concerned about the impact of introducing new technology on productivity. Firstly, one of the major problems in a situation of technology adoption is usually the lack of knowledge about the technology and the need of overcoming the learning curve connected with it [33]. Therefore, poor introduction of new technology can "lead to a longer than anticipated (or budgeted for) payback time or even loss of investment due to non-use or ineffective use" [29]. Secondly, long period of decreased productivity can substantially decrease the competitiveness of the company [29].

Since lack of knowledge about the new technology in the company is the major problem when introducing new technologies, it is very important to provide the developers with a good source of information about the new technology. In [24] the authors stress the need of having this knowledge codified. Tacit knowledge (not available in written form) is considered as one of the most important obstacles for seamless introduction of new technology. Another important aspect is the motivation of the staff when it comes to accepting the new technology. In [29], the enthusiasm and willingness to accept a new technology are described as the key condition for overcoming productivity problems connected with technology introduction. Lack of experience and enthusiasm about the new technology are not the only problems that must be overcome. To other factors that negatively affect the productivity in the early development using the new technology belong [38]: lack of flexibility and timeliness, low quality, excessive distance between management and employees, unreliability of suppliers, poor labour relations, lack of skill, poor process design, insufficient capacity, loose capacity management, and poor communication between departments.

The negative impact of all issues mentioned above can usually be decreased by taking certain actions. For example the initial lack of knowledge can be compensated by training. Processes can be adjusted.

There are certain ways of increasing motivation as well. On the other hand, it is also natural to expect a productivity increase when the organization gets more familiar with the technology. Such a tendency has been recognized and researched. Examples of terms connected with increasing productivity over time are “learning curve” [72, 80] and “experience curve” [16]. Both are based on an observation that the cost of producing a unit of a product is decreasing with time. In [4] the possible mechanisms behind this phenomenon are identified. These include acquiring better, more suitable tools; adjusting production methods; product design changes; improved and more effective management; change of product volume and quality; developer learning; and finally incentive pays [4].

3.3 *Process driven productivity improvement*

Productivity can be affected by improvements in the development processes [71]. Therefore, there is an ongoing discussion regarding the impact of different processes and different software development practices on the productivity of software development. In [52], the authors noticed that there are two kinds of improvements in the process area. Some are complimentary with current development process, i.e., various practices are suggested that will make current development process better. Other approaches focus on finding another process model, i.e., finding a process model that would be more suitable for the development at a company than the currently used one. In our studies we looked at both these kinds of improvements in the process area.

Our process improvement suggestion was the introduction of a fault prediction model with the goal of making fault detection more efficient. Fault detection and removal activities account for about 40% of software development projects’ costs [33] so any improvement in the area of fault detection is likely to have a significant impact on project cost and consequently also on productivity. Normally, in software systems about 60%-80% of the faults can be found in about 20% of the code modules [11, 60]. At the same time about half of the code modules are usually defect free [11]. These characteristics indicate that that there is a potential for savings, if we manage to focus fault handling efforts on the portion of the code that actually contains faults. In order to achieve this productivity improvement we must, however, be able to identify code units that actually contain faults.

A popular method for identifying fault-prone code is by using a prediction model (e.g., [25, 60, 64-66, 83]). Fault prediction models that can be found in the literature are based on a great variety of different predictor variables, e.g., different code metrics (e.g., [46, 65, 83]) or variations of Chidamber and Kemerer (C&K) [21] object oriented metrics (e.g., [17, 25, 83]). There are also studies that take historical information about the code fault-proneness into account (e.g., [62, 64, 65]). There is also large variety in the methods that are used for fault prediction. These methods range from uni- and multivariate linear regression (e.g., [18, 20, 59, 61, 66, 83]), logistic regression (e.g., [17, 25, 28, 48]) and regression trees (e.g., [46, 47]) through neural networks (e.g., [49, 74]), and Bayesian Belief Nets [26], to statistical clustering techniques combined with expert estimations [84, 85].

The most common classification of prediction models is based on their output. Based on that prediction models can be classified as [49]:

- Classification models – these models classify code units as fault-prone or not, i.e., they predict if the code unit contains faults. Examples of such models can be found in [17, 25, 28, 48].
- Quality prediction models - these models attempt to quantify the quality of the code unit, e.g., by predicting the number of faults or fault-density of a particular code unit. Examples of such models can be found in [18, 20, 59, 66, 83].

Even though they predict slightly different things, both kinds of prediction models were found to be useful from the perspective of improving fault detection efficiency by focusing fault detection efforts only on code units that are most likely to contain faults.

Sometimes introducing improvements to currently existing processes might not be enough. It might be so that it is the process itself that is the major problem. For example, the traditional Waterfall model is considered to be ineffective when it comes to coping with rapidly changing requirements [6, 23, 34, 52, 77]. In the Waterfall model a change of requirements results in a need of repeating all development stages from the beginning [71], which is the source of waste and rework. Waste and rework result in productivity decrease, because they increase the effort without increasing the product size. Therefore, many new process models were suggested for markets in which requirements are changing rapidly. These models usually favour customer responsiveness over the control provided by the traditional Waterfall model. In such models the customers are more involved in the development process, changes are facilitated, and a part of a system or its prototype is released to the customers at much earlier stage of development than in the Waterfall model [52]. In this way the risks of waste or rework are reduced.

There is a lot of research in the area of comparison of the traditional Waterfall development process with other ways of developing software, e.g., [6, 23, 34, 43, 58, 77]. These studies largely focus on comparing the advantages and disadvantages of the Waterfall model compared to a proposed new way of developing software. For example, Dalcher *et al.* [23] present an experiment in which a number of teams developed similar systems using different processes. In this study, the teams used a traditional approach, incremental development, evolutionary development, and extreme programming. The results showed that, in fact, the use of modern approaches (i.e., incremental development, evolutionary development, and extreme programming) lowered lead-time and improved productivity.

4. Contributions in this thesis

In this section we present the major contributions in this thesis. We describe our contributions in the order of the research questions (see Section 2 for details regarding research questions). In Section 4.1 we present our results concerning the technology driven productivity improvement. In Section 4.1.1 we describe our findings regarding the new server platform introduction (i.e., research questions: **RQ1**, **RQ2**). Section 4.1.2 focuses on the results from our studies that aimed at finding a cost-efficient architecture by combining standard and fault-tolerant server platforms into one solution, which correspond to our research question **RQ3**. Section 4.2 describes our contributions in the area of process driven productivity improvement. In Section 4.2.1 we present our results in improving the efficiency of fault detection, i.e., research questions: **RQ4-RQ7**. In Section 4.2.2 our contributions in the field of process change evaluation are presented, i.e., research question **RQ8**.

4.1 Technology driven productivity improvement

4.1.1 Platform change

Our first research question (**RQ1**, see Section 2.1.1) concerned the impact of introducing new platform on productivity of software development in a short and in a long term perspective. To answer this question we performed two case studies at Ericsson. They are presented in *Paper I* and *Paper II.* In these studies we measured the impact of a fault-tolerant platform introduction on software development productivity.

In *Paper I* we investigated the impact that the introduction of a fault-tolerant platform had on software development productivity in a short term perspective. We compared early software development productivity on a fault-tolerant platform with the productivity in a project in which a standard UNIX platform was used. We discovered that the development was four times as productive on UNIX as on the new platform when it comes to delivering functionality, i.e., in the UNIX based project the same amount of work brought four times as much functionality. This difference was caused by two facts: in UNIX the code was written twice as fast and, on average, there was twice as much functionality delivered by a line of code compared to the development on the fault-tolerant platform.

In *Paper II* we present a case study performed at the same department at Ericsson three years later. We compared the productivity in two projects in which the fault-tolerant platform was used. One of them was an early project, one of the first projects in which the platform was used (the same as the one described in *Paper I*). The other was performed three years later. We discovered that in the subsequent project the functionality was delivered four times as fast as in the early one. On average, in the subsequent project there was almost three times as much functionality delivered by a single line of code and the code was written 40% faster.

Our second research question (**RQ2**, see Section 2.1.1) concerned the development productivity improvement on the specialized fault-tolerant platform. To address this question we identified issues that affect productivity negatively and we suggested a number of remedies to these issues. In *Paper I* we described reasons for low productivity in early software development on the fault-tolerant platform. We identified issues connected with staff competence, work characteristics, and platform characteristics and we assessed their impact on productivity. The staff competence level was considered as the one that most affects the low code delivery rate. Lack of libraries for typical purposes (e.g., communication protocols) contributed most to the low amount of functionality delivered by an average line of code. Therefore, as remedies to the productivity problems of the early development on the fault-tolerant platform we mainly suggested a number of competence development activities. We also recommended some process and platform improvements.

In *Paper II* we identified issues that affected productivity increase in the subsequent, more mature development on the new platform. The increase of code delivery rate was mostly caused by the increase of experience and knowledge about platform among the developers. The factor that contributed most to the large increase of the amount of functionality delivered by an average line of code was a large dose of code-reuse. Among the major productivity bottlenecks in the subsequent development on the new platform we identified certain platform related shortcomings, e.g., the lack of advanced programming tools available for it. Therefore, as far as the mature development is concerned, it seems that some platform improvements, mostly in terms of advanced programming tools, could contribute to further productivity increase.

4.1.2 Architecture change

As we have discovered before, the cost of development for the new platform can be significant, especially just after the platform is introduced (see Section 4.1.1). To minimize the impact of the low development productivity on the project cost we decided to look for a hybrid architecture that combines fault-tolerant and standard platforms in order to provide good non-functional requirements in a cost-efficient manner (**RQ3**, see Section 2.1.2). Since it is difficult to suggest and evaluate such an architecture in a general way, we chose an example application that is very likely to be implemented on the fault-tolerant platform. The example we chose was a Diameter Credit-Control Server [36], an application responsible for rating and accounting in prepaid services. The technical qualities required from the Credit-Control Server are availability, reliability and performance. To these qualities we added the implementation cost.

The study described in *Paper III* aimed at establishing the current state-of-the-art when it comes to implementing the Credit-Control Server on the standard UNIX platform. We performed this study to obtain a point of reference for the evaluation of our hybrid architecture. In the study we identified four candidate architectures. All of them presented some trade-off between availability, reliability and performance. In the study we also identified certain problems that could not be overcome using standard platforms. To overcome the limitations of standard platform implementations we suggested a new architecture of the Credit-Control Server. Our architecture is described in *Paper IV*. For the new architecture we suggested four different variants of the Credit-Control Server implementation. All variants offer significantly better availability and reliability compared to the standard platform implementations. The cost of implementation of our architecture is about 30% higher compared to the standard platform implementation. However, it offers the availability and the reliability comparable with the implementation on the fault-tolerant platform for about one-third of its price.

4.2 Process driven productivity improvement

4.2.1 Process improvement

Our next research question (**RQ4**, see Section 2.2.1 for details) concerned the fault detection efficiency improvement that can be expected from applying a fault prediction model. This question was addressed in *Paper V*, where we suggested and evaluated a number of fault prediction models. The study described in *Paper V* was based on data from three releases of two large software systems produced by Ericsson. Our prediction models were built using data from one release of one of the systems. They were then evaluated using the data describing the system on which they were built, the next release of the system on which they were built, and a completely different system. The fault prediction models built in this study were based on design and code metrics and predicted fault densities of individual classes. Their performance was quantified as the percentage of the theoretical maximum efficiency improvement over not using any model at all. We found that our models were able to provide, on average, 38% to 57% of the maximal theoretical improvement in fault detection efficiency. This means that using any of our models makes fault detection more efficient than not using any model at all. The difference in performance of our models, compared to not using any prediction model at all, was shown to be statistically significant. Our best prediction model made it possible to achieve 75% of the maximum possible improvement when applied to the next release of the system on which it was built. When applied to a completely different system it achieved 55% of the maximum improvement.

Fault prediction models described in *Paper V*, are available only after the system is implemented, because they are primarily based on code metrics. Since there is an added value in having these predictions earlier in the development process, we tried to find a method for early prediction of faults (**RQ5**, see Section 2.2.1 for details). In *Paper VI,* we suggest and evaluate methods for early fault density prediction in modified classes. Our predictions are based on information concerning the number of new and modified methods in the class. We evaluated our prediction methods on the data from three large telecommunication systems produced by Ericsson. We compared our predictions with the state-of-the-art prediction model available after the code is implemented. We found that our methods provide predictions that are of similar quality to the best predictions available after the system is implemented, but are available earlier in the development process.

Our next research question concerned the comparison of the accuracy of fault prediction made by statistical prediction models with fault prediction made by human experts (**RQ6**, see Section 2.2.1). In *Paper VII* we present a study in which we compared these accuracies empirically. We asked two groups of experts to perform predictions concerning two large telecommunication systems developed by Ericsson. Then we applied a statistical fault prediction model to these two systems. It turned out that in both systems our statistical fault prediction outperformed human estimations in two ways. First, it was more accurate. Second, it accounted for more code (human experts failed to estimate the fault-proneness of all code units in the system).

In all our studies described in this section we focused on modified code only. The reason was that in the systems under study a majority of the faults was usually found in the modified code. This does not imply that, in general, modified code is more fault-prone. Therefore, in our next research question we investigated this issue of "fault-proneness" of modified code (**RQ7**, see Section 2.2.1). In *Paper VIII,* we compared the fault-proneness of new and modified classes in four large telecommunication systems. We found that there is no statistically significant difference between new and modified classes when it comes to either number of faults per class or class fault-density. However, we found that, on average, the risk of introducing a fault when writing a line of code in a new class is significantly smaller compared to the risk connected with writing/modifying a line of code in an already existing class.

4.2.2 Process change

In our last research question we focused on evaluating the impact of a process change (**RQ8**, see Section 2.2.2 for details). In *Paper IX* we present a study in which we evaluated a new process called Streamline Development and its applicability for software development at Ericsson. The major difference Streamline Development would introduce, compared to the current development practices at Ericsson, would be a reduced project scope. By reducing the scope the projects will be less exposed to the risk of changing market demands. As we found in *Paper I* and *Paper II* changing market demands belong to the main productivity bottlenecks in software development projects. Therefore, by reducing the risk of changing market demands, Streamline Development should lead to improved development productivity.

In *Paper IX* we identify opportunities and challenges connected with changing from traditional software development to Streamline Development. By performing interviews with persons representing the roles that will be affected by changing the development process, we collected a number of opinions regarding introducing Streamline Development at Ericsson. These opinions were classified later using a modification of Force Field Analysis [44], i.e., they were classified as Pushing factors, Resisting factors, and Required changes. Pushing factors are things that would improve if a new process was introduced. Resisting factors are things that would deteriorate. Required changes are things that must be taken care of before introducing the new process. By balancing all those factors it was possible to perform an early evaluation of Streamline Development. An overall conclusion from this study was that that Streamline Development seems promising and that it has potential to achieve its goals.

5. Methodology

5.1 Industrial relevance

The lack of industrial relevance is considered as one of the major problems of software and computer engineering research. Researchers tend to study problems irrelevant to the industry, which makes the gap between the research and practice bigger and bigger [31]. Therefore, industrial relevance was one of our primary objectives. We tried to achieve this by looking for research problems and questions in industry, establishing close co-operation with industry, conducting the research in an industrial setting as well as discussing and validating the results and conclusions in discussions with our industrial partners. We used empirical methods; we observed, analysed and described real-life situations. We collected our data from real, large projects that ended as products available on the market. When performing estimations or interviews we reached people that have experience in professional software development.

The typical empirical research approaches are experiments, case studies and surveys [50]. The major difference between them concerns the scale and the level of formalism. The experiments are usually performed in a “laboratory” setting in which a controlled environment can be provided. In the experiments we can analyse the impact of specified factors on the output since we are able to control and eliminate the impact of other, possibly confounding factors. Because of that control, the experiments can be replicated, which is highly desirable in scientific studies. The laboratory setting, however, limits the scope of the experiments. The case studies usually lack the possibility of replication. They aim at “development of detailed, intensive knowledge about a single case, or a small number of related cases” [67]. Although sometimes hard to generalize, the case studies are useful as a way of in-depth analysis of some particular real-life phenomena. Surveys are easier to generalize since they usually involve very large sample. Their major objective is to detect some general trends in population. They, however, do not provide the same level of in-depth analysis as a case study does.

The majority of our studies are case studies. In *Paper I* and *Paper II* we analysed and compared three projects by performing interviews, sending questionnaires, organizing workshops, studying documentation and measuring the code characteristics. In *Papers V-VIII* we collected code and design measurements from a number of real projects. In *Paper IX* we applied our process evaluation method to evaluate a real development process and Ericsson. In *Paper III* and *Paper IV*, we introduced elements of experimentation. We evaluated a number of architectures using scenario-based assessment [15], simulation [15] and expert estimation [15]. The scenarios were created by the experts from industry. Simulation was done in the industrial setting, involving crucial elements of the real Credit-Control systems, real workload characteristics and real hardware. The estimations were performed by the experts from the industry and were based on analogies with the real systems that are available in the market.

We believe that by introducing a link to industry in every stage of every study we managed to assure industrial relevance in our studies.

5.2 Methods

Interviews and workshops. *Interview* is a very popular research method that can be used in almost every stage of a study. They can be used to collect data, they can help in data analysis and validation of the results [67]. There are three types of interviews possible [67]:

- *fully structured interview*, in which the questions, their order and wording are predetermined
- *semi-structured interviews*, in which the questions are predetermined, but their order and focus can be modified depending on the interviewee's interest
- *unstructured interviews*, in which only the area of interest is predetermined

According to [67] the flexibility of semi- and unstructured interviews makes them a perfect choice for exploratory work.

In *Paper I* we used interviews for exploratory work. We performed a number of semi-structured interviews to better understand the characteristics of early development on the fault-tolerant platform and to collect information about possible productivity bottlenecks. By using semi-structured interviews we clearly indicated the topic we are interested in and more or less guided the conversation, but we still left space for interviewee initiative. It proved to be a good idea – the interviewees mentioned some issues we did not take into account earlier. A similar strategy was used in *Paper II*. However, instead of a number of separate, semi-structured interviews we organized a workshop with a number of experts. We pin-pointed the general direction of the discussion, but we also adjusted the order and focus of the topics discussed to the interest of the participants. We found the workshop to be a very good method for conducting exploratory study because it provoked discussions and brainstorming as well as resulted in consensus decisions. In *Paper IX* we used interviews as our main data collection method. The interviews were also semi-structured. To each interview we invited a number of experts that shared the same role in the company. In this way we managed to combine the benefits of interviews and workshops – our questions provoked discussions, but the participants focused on representing the same perspective.

Throughout all studies we performed a number of *unstructured* interviews. In all our studies they were used in the analysis of the findings as well as in the validation. We organized meetings in which we discussed the correctness and the relevance of our findings and judgements with the industrial experts. In *Paper III* and *Paper IV* we performed unstructured interviews to verify our understanding of the technology as well as to obtain information for the architecture evaluation. In *Paper IX* unstructured interviews were mostly used to define study scope and to assure a common understanding of study goals.

Analytic Hierarchy Process (AHP). *AHP* [68] is “a method that enables quantification of subjective judgments” [75]. It makes it possible to assign priorities to the list of alternatives. In AHP pairwise comparisons are performed. Each two alternatives are compared against their relative importance. Based on such relative importance a total priority vector is calculated. In the priority vector, there is a weight assigned for each alternative. In this way the AHP outcome describes not only the order but also the distances between the alternatives, i.e. we can not only say that A is better than B but also that A is 3 times as good as B. AHP is a well established method for performing prioritisation in software engineering (see [7, 75] for an overview). We used AHP in *Paper I*. As input we had a list of issues that were identified in interviews as productivity bottlenecks. We asked each respondent to perform an AHP analysis on those issues to find which of the issues affect productivity most. AHP could potentially make it possible for us to filter the issues that had large impact on productivity from those which were (almost) insignificant. Since we had several respondents we got several subjective priority vectors. To get rid of the subjectivity and obtain a single priority vector we used an approach similar to [75] – we took the mean values. AHP is sometimes criticized as being time-consuming (when there are *n* alternatives, AHP requires performing $n*(n-1)/2$ comparisons). However, since we had only 9 alternatives, which required 36 comparisons, neither we nor our respondents consider it a major problem. Apart from the priority vector AHP also provides an inconsistency index – a value from 0 to 1. It describes to what extent the respondent's answers were consistent (0 - fully consistent, 1 - completely inconsistent). The rule is that the index should have a value below 0.1 to consider the answer consistent [75]. Our respondents had consistency problems. To overcome these problems we used AHP only as a supporting method. After performing AHP the respondents got their priority vectors and were asked to change them if they did not match their opinion. Only a few introduced some rather insignificant changes.

Expert judgement. Despite its obvious shortcomings, like bias, subjectivity and difficulty in repeating, expert judgement is an acceptable and widely practiced way of performing estimations [12, 42, 69]. Expert judgement is used in many software engineering related areas, from prediction [69] to assessment [15]. One of the major problems connected with the expert judgement is the lack of precision. Some researchers report huge discrepancy between the estimations done by different experts (e.g. [57]). To overcome the problem of subjectivity some methods that involve a group of experts were suggested. Example of such a method is Delphi, in which the researcher asks a number of experts for individual estimation and then iterates until some kind of consensus is reached [12]. Some promising results were reported by [57] in situations when group discussions were performed. We used expert estimations in many of our studies. In the ones described in *Paper I* and *Paper II* expert estimations were used to assess the difference in amount of functionality delivered by two products. The experts did this by discussing what kind of functionality is delivered in each product and comparing the complexity of the functionalities. In *Paper III* and *Paper IV* the experts estimated the characteristics of good, average and bad quality of the Credit Control Server implementation. In *Paper IV* we asked the experts to perform development cost estimations. In *Paper VII* the experts were asked to perform estimations regarding the most probable fault locations.

Architecture assessment methods. According to [15] there are the following architecture assessment methods: scenario-based, simulation based, mathematical model-based, and experience-based architecture assessment. In scenario-based assessment the typical scenarios (e.g. usage, change or hazard scenarios [15]) are defined. They are used for predicting the quality attributes. We used scenarios in *Paper III* and *Paper IV* to define the frequencies of events that have impact on the studied qualities. Simulation based assessment involves "high level implementation of the architecture" [15]. It makes it possible to assess the performance of an architecture by executing a typical workload and measuring the response time and the throughput [15]. This is the way in which we used simulations in the studies described in *Paper III* and *Paper IV*. The architecture assessment performed by us is rather simple. We did not use any advanced methods for quality attribute prediction. However, the purpose of our evaluation was to pin-point the strengths and weaknesses of the architectures discussed and we think the simplified examples presented it well enough.

Descriptive and inferential statistics. In our studies we use statistics to both describe and summarize the data (descriptive statistics) and to reason and draw conclusions about the data (inferential statistics). In our studies presented in *Papers I-VI,VIII* we use basic concepts of descriptive statistics (central tendency and dispersion measures, e.g., mean, median, and standard deviation) to summarize our datasets. In *Paper I* and in *Paper II* additionally we present some of our data in graphic form (histograms). In *Papers V and VII* we quantify the strength of relationship between different variables using correlation coefficient [3, 79]. Our prediction models from *Paper V* and *Paper VII* are built using univariate and multivariate linear regression [79]. In our studies we also test a number of hypotheses regarding our datasets and our results. In *Paper V* we use Wilcoxon Signed-Rank Test [79] to check if the improvements offered by our prediction models are statistically significant. In *Paper VIII* we use Mann-Whitney U test [79] to compare the fault-proneness of new and modified code units.

5.3 Validity

There are four types of validity: internal, external, construct, and conclusion validity [79].

The *internal* validity "concerns the causal effect, if the measured effect is due to changes caused by the researcher or due to some other unknown cause" [40]. Generally, we can say that internal validity threats concern the issues that could have impacted the phenomena we discuss and which we did not take into account in the study. As we see it, one issue that should be discussed in the context of internal validity of our studies is the risk that, in the studies described in *Paper I* and *Paper II*, the productivity could have been impacted by other than platform related issues. To minimize this threat we took a number of measures. One obvious step was to take quality into account. The productivity could have been impacted by different quality levels of the projects. We also deliberately selected projects done in the same department to assure that the staff involved in those projects was at least overlapping, which to some extent eliminated the risk of some external competence input affecting the productivity. Also, when performing the interviews concerning the productivity bottlenecks we always had some open questions, in which the interviewees could mention issues we did not ask about. Unfortunately, in overall it is much easier to assure the internal validity in controlled experiments than in case-studies. Therefore, we can not claim that we identified and that we took all possible threats to internal validity into account, since it is basically impossible. However, we did our best in identifying and documenting possible threats to internal validity. Another situation in which it may be interesting to discuss internal validity issues are our fault prediction studies (i.e., *Paper V-VII*). In their case we obviously neither can nor want to claim the causal relationship between our predictor variables and faults. The study is based on correlations, so there is a chance that there is some factor other than those taken into account in this study that demonstrates itself in both predictor variables selected by us and in faults. Therefore, it is possible to perform useful predictions even without the causal relationship between dependant and independent variables.

The *external validity* concerns the possibility of generalising the findings. Certain results of our study are very situation dependant, e.g., the values of productivity differences and the quantifications of the productivity bottlenecks' impact from *Paper I* and *Paper II*. Also our performance measurements from *Paper III* and *Paper IV* are hardware dependant, e.g. much slower network connection or faster memory would affect the actual results. Our fault prediction studies (*Paper V-VII*I) are all based on data from the same company, so projects could be considered to some extent, similar, which may explain why prediction models are transferable between these projects. Also our factors that impact the decision regarding the process change in *Paper IX* are very much dependant on the situation at Ericsson at the time the study was performed. However, we still believe that many general lessons can be learned from our studies. For example, when changing to the specialized technology with the unique programming model, certain competence related problems are quite probable. Such technologies, due to the limited number of users, may also suffer from a rather low support in terms of tools or off-the-shelf, ready to use components. What we would like to show is that there are certain problems connected with the introduction of a very specialized technology and that the total impact of these problems can be significant and should not be underestimated. Also our suggestion of gradual introduction of the specialized technology (*Paper IV*) shows that in some cases such a solution can provide a good trade-off between technical and economical requirements. In our studies we have also taken a number of measures to assure the external validity of our results. For example, to assure the external validity of our prediction models (*Paper V-VII*) we built and evaluated those using data from different systems. We tried to select as different systems as possible within one large company (e.g., systems developed by different development organizations in different countries). Finally, some of our major findings seem to be similar to findings reported by other researchers, e.g., findings concerning productivity bottlenecks from *Paper I* and *Paper II*, findings concerning fault predictors in *Papers V-VII*, and findings concerning the evaluation of Streamline Development in *Paper IX.* We believe that these similarities strengthen our results.

The *construct validity* ”reflects our ability to measure what we are interested in measuring” [40]. Simply speaking construct validity is about assuring that what we have measured is actually what we wanted to measure. It describes to what extent the quantities we have measured reflect the concepts we wanted to measure. The construct validity might be a problem in our case, since we try to measure abstract quantities, like software size, software functionality, or quality. As we have already discussed in Section 3.1.1 these problems will always occur, since there are no standardized metrics for many of the concepts we discuss. We tried to overcome that problem by explicitly describing what and how we measure. In some cases (e.g., software size measurement in *Papers I and II*) we applied multiple metrics (SLOC, number of classes) and we checked if they correlate. In other cases, we explicitly made certain assumptions (e.g., in *Paper V-VII* we assumed that fault-density is a measure of fault-proneness of code unit).

The *conclusion validity* concerns the correctness of conclusions we have made. When discussing conclusion validity we want to assess to what extent the conclusions we made are believable. The difference between internal and conclusion validity is that conclusion validity is mostly interested in checking if there is a correct relationship (e.g. statistically significant) between the input and the output [2]. Internal validity describes if the relation between input and output was actually caused by what we claim it caused. One possible threat to conclusion validity is the reliability of the measures [2]. In our studies we used estimations (*Paper I-IV*) extensively, which always can be affected by a number of issues, like e.g., subjectivity of assessment. However, as we have described in Section 5.2, we have tried to minimize their impact by introducing certain precautions, like group consensus or group discussion. Wherever possible, we have also checked the statistical significance of our findings.

6. Conclusions

Our primary research goal in this thesis is to investigate how the productivity of software development can be improved. We looked into technology driven and into process driven productivity improvements. A technology driven productivity improvement discussed in this thesis is the introduction of a new specialized technology. Process driven productivity improvements suggested and evaluated by us include using fault prediction models for improving fault detection efficiency and the change of the entire development process.

We investigated the impact of the new technology by performing case studies in which we measured the productivity in the organization that introduced the specialized, fault-tolerant platform. The measurement revealed a significant decrease of productivity when the new technology was introduced. We identified a factor of four in productivity decrease compared to development where standard technology is used. This factor of four was equally affected by low code delivery rate and high average amount of code necessary to deliver functionality. We found that both of these tend to improve with experience and maturity. In mature development on the fault-tolerant platform the productivity reached the level of the productivity on the standard platform. It was, however, mostly due to the large code reuse that reduced the amount of code necessary to deliver functionality.

To suggest improvements to the platform introduction process we investigated the reasons for the low productivity in the initial development on the new platform. We found that the main reason was the competence level when it comes to using the new technology. Therefore, as improvement methods, we mostly suggested a number of competence development activities. However, since it seems that some productivity decrease is unavoidable, we looked for a solution that could decrease the impact of low initial productivity on project’s cost. We showed that standard and fault-tolerant platforms can be combined into a single architecture that provides good technical qualities for a reasonable price. Such an architecture can be an interesting alternative when development using the specialized technology is still very expensive. We also identified productivity bottlenecks in the subsequent software development. We observed a significant decrease of the role of competence as a factor that affects productivity negatively. As the major productivity bottleneck in subsequent development on the new platform we found the shortage of programming tools available for the specialized platform. Therefore, our major productivity improvement suggestions for mature software development concerned introducing tool support for the software development on the specialized platform.

When looking for process driven productivity improvements we focused mostly on suggesting and evaluating methods for increasing the efficiency of fault detection. We suggested and evaluated a number of statistical fault prediction models. We showed that our models were able to provide, on average, 38% to 57% of the maximal theoretical improvement in fault detection efficiency. We also found that in industry fault predictions are commonly made by human experts. Therefore, to assess the practical value of our models, we compared their accuracy with the accuracy of predictions made by human experts. We found that statistical fault prediction models outperformed human estimations because they were more accurate and they accounted for more code. To enable more efficient resource allocation in software development projects we looked for methods that would enable fault prediction early in the software development process, i.e., before the system is implemented. We suggested a new method for such an early fault prediction. We found that our method provided predictions of similar quality to the best predictions available after the system is implemented.

To further investigate the reasons for fault proneness of code units we compared the fault proneness of new and modified code. We found no significant difference between new and modified classes when it comes to number of faults per class or class fault-density. However, we found that the risk of introducing a fault when writing a line of code in a new class is significantly smaller compared to the risk connected with writing/modifying a line of code in an already existing class.

We also looked for improvements that can be achieved by changing the development process. We performed an early evaluation of Streamline Development, which is a new process concept developed at Ericsson. One of the goals of this process is to improve productivity of software development. To evaluate Streamline Development we identified its advantages and drawbacks as well as issues that must be addressed before it can be introduced. Overall, we found that Streamline Development has potential to achieve its goals.

7. References

[1] *IEEE standard for software productivity metrics*, in *IEEE Std 1045-1992*. (1993). 2.

[2] *The Web Center for Social Research Methods,* *http://www.socialresearchmethods.net/*. (2005).

[3] A.D. Aczel and J. Sounderpandian, Complete business statistics, McGraw-Hill, Boston, Mass., (2006).

[4] P.S. Adler and K.B. Clark, Behind the Learning Curve: A Sketch of the Learning Process. *Management Science*, 37 (1991), 267-281.

[5] D. Barbara and H. Garcia-Molina, The case for controlled inconsistency in replicated data. *Proceedings of Workshop on the Management of Replicated Data*, (1990), 35-38.

[6] O. Benediktsson and D. Dalcher, Effort estimation in incremental software development. *Software, IEE Proceedings-*, 150 (2003), 351-358.

[7] P. Berander, Prioritization of Stakeholder Needs in Software Engineering. Understanding and Evaluation, Licentiate Thesis, Blekinge Institute of Technology, (2004).

[8] A. Bhide, Experiences with two high availability designs (replication techniques). *Second Workshop on the Management of Replicated Data*, (1992), 51-54.

[9] J.D. Blackburn and G.D. Scudder, Time-based software development. *Integrated Manufacturing Systems*, 7 (1996), 60-66.

[10] J.D. Blackburn, G.D. Scudder, and L.N. van Wassenhove, Improving Speed and Productivity of Software Development: A Global Survey of Software Developers. *IEEE Transactions on Software Engineering*, 22 (1996), 875-886.

[11] B. Boehm and V.R. Basili, Software Defect Reduction Top 10 List. *Computer*, 34 (2001), 135-137.

[12] B.W. Boehm, Software engineering economics, Prentice-Hall, Englewood Cliffs, N.J., (1981).

[13] H.S. Bok and Raman, Software engineering productivity measurement using function points: a case study. *Journal of Information Technology*, 15 (2000), 79-91.

[14] F. Bootsma, How to obtain accurate estimates in a real-time environment using full function points. *Proceedings of the 3rd IEEE Symposium on Application-Specific Systems and Software Engineering Technology*, (2000), 105-112.

[15] J. Bosch, Design and use of software architectures: adopting and evolving a product-line approach, Addison-Wesley, Harlow, (2000).

[16] Boston Consulting Group, Perspectives on Experience, Boston, (1972)

[17] L.C. Briand, J. Wust, J.W. Daly, and D.V. Porter, Exploring the relationship between design measures and software quality in object-

oriented systems. *The Journal of Systems and Software*, 51 (2000), 245-273.

[18] M. Cartwright and M. Shepperd, An empirical investigation of an object-oriented software system. *IEEE Transactions on Software Engineering*, 26 (2000), 786-796.

[19] Z. Chi and Z. Zheng, Trading replication consistency for performance and availability: an adaptive approach. *Proceedings of 23rd International Conference on Distributed Computing Systems*, (2003), 687-695.

[20] S.R. Chidamber, D.P. Darcy, and C.F. Kemerer, Managerial use of metrics for object-oriented software: an exploratory analysis. *IEEE Transactions on Software Engineering*, 24 (1998), 629-639.

[21] S.R. Chidamber and C.F. Kemerer, A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20 (1994), 476-494.

[22] R.T. Coupe and N.M. Onodu, An empirical evaluation of the impact of CASE on developer productivity and software quality. *Journal of Information Technology*, 11 (1996), 173-182.

[23] D. Dalcher, O. Benediktsson, and H. Thorbergsson, Development life cycle management: a multiproject experiment. *Fibres and Optical Passive Components, 2005. Proceedings of 2005 IEEE/LEOS Workshop on*, (2005), 289-296.

[24] A.C. Edmondson, A.B. Winslow, R.M.J. Bohmer, and G.P. Pisano, Learning How and Learning What: Effects of Tacit and Codified Knowledge on Performance Improvement Following Technology Adoption. *Decision Sciences*, 34 (2003), 197-223.

[25] K. El Emam, W.L. Melo, and J.C. Machado, The prediction of faulty classes using object-oriented design metrics. *The Journal of Systems and Software*, 56 (2001), 63-75.

[26] N. Fenton and M. Neil, A critique of software defect prediction models. *IEEE Transactions on Software Engineering*, 25 (1999), 675-689.

[27] N. Fenton and S.L. Pfleeger, Software metrics: a rigorous and practical approach, PWS, London; Boston, (1997).

[28] F. Fioravanti and P. Nesi, A study on fault-proneness detection of object-oriented systems. *Fifth European Conference on Software Maintenance and Reengineering*, (2001), 121-130.

[29] W. Fisher and S. Wesolkowski, How to determine who is impacted by the introduction of new technology into an organization. *Proceedings of the 1998 International Symposium on Technology and Society, 1998. ISTAS 98. Wiring the World: The Impact of Information Technology on Society.*, (1998), 116-122.

[30] H. Garcia-Molina and B. Kogan, Achieving High Availability in Distributed Databases. *IEEE Transactions on Software Engineering*, 14 (1988), 886-897.

[31] R.L. Glass, The software-research crisis. *IEEE Software*, 11 (1994), 42-48.

[32] R.L. Glass, The Realities of Software Technology Payoffs. *Communications of the ACM*, 42 (1999), 74-80.

[33] R.L. Glass, Frequently Forgotten Fundamental Facts about Software Engineering. *IEEE Software*, 18 (2001), 112.

[34] D.R. Graham, Incremental development and delivery for large software systems. *Software Prototyping and Evolutionary Development, IEE Colloquium on*, (1992), 2/1-2/9.

[35] D. Haggander, L. Lundberg, and J. Matton, Quality attribute conflicts - experiences from a large telecommunication application. *Proceedings of Seventh IEEE International Conference on Engineering of Complex Computer Systems*, (2001), 96-105.

[36] H. Hakala, L. Mattila, J.-P. Koskinen, M. Stura, and J. Loughney, *Diameter Credit-Control Application (internet draft - work in progress)*. (2004): http://www.ietf.org/internet-drafts/draft-ietf-aaa-diameter-cc-06.txt.

[37] P. Hantos and M. Gisbert, Identifying Software Productivity Improvement Approaches and Risks: Construction Industry Case Study. *IEEE Software*, 17 (2000), 48-56.

[38] J. Harvey, L.A. Lefebvre, and E. Lefebvre, Exploring the Relationship Between Productivity Problems and Technology Adoption in Small Manufacturing Firms. *IEEE Transactions on Engineering Management*, 39 (1992), 352-359.

[39] A. Hisgen, A. Birrell, T. Mann, M. Schroeder, and G. Swart, Availability and consistency tradeoffs in the Echo distributed file system. *Proc. of the Second Workshop on Workstation Operating Systems*, (1989), 49-54.

[40] M. Host, B. Regnell, and C. Wohlin, Using Students as Subjects-A Comparative Study of Students and Professionals in Lead-Time Impact Assessment. *Empirical Software Engineering*, 5 (2000), 201-214.

[41] M. Huggett and S. Ospina, Does productivity growth fall after the adoption of new technology? *Journal of Monetary Economics*, 48 (2001), 173-195.

[42] R.T. Hughes, Expert judgement as an estimating method. *Information and Software Technology*, 38 (1996), 67-76.

[43] B.D. Jensen, A software reliability engineering success story. AT&T's Definity PBX. *Software Reliability Engineering, 1995. Proceedings., Sixth International Symposium on*, (1995), 338-343.

[44] G. Johnson, K. Scholes, and R. Whittington, Exploring corporate strategy, Financial Times Prentice Hall, Harlow, (2005).

[45] C. Jones, How office space affects programming productivity. *Computer*, 28 (1995), 76-77.

[46] T.M. Khoshgoftaar, E.B. Allen, and J. Deng, Controlling overfitting in software quality models: experiments with regression trees and classification. *Proc. of The 17th International Software Metrics Symposium*, (2000), 190-198.

[47] T.M. Khoshgoftaar, E.B. Allen, and D. Jianyu, Using regression trees to classify fault-prone software modules. *IEEE Transactions on Reliability*, 51 (2002), 455-462.

[48] T.M. Khoshgoftaar, E.B. Allen, W.D. Jones, and J.P. Hudepohl, Accuracy of software quality models over multiple releases. *Annals of Software Engineering*, 9 (2000), 103-116.

[49] T.M. Khoshgoftaar and N. Seliya, Fault Prediction Modeling for Software Quality Estimation: Comparing Commonly Used Techniques. *Empirical Software Engineering*, 8 (2003), 255-283.

[50] B. Kitchenham, L. Pickard, and S.L. Pfleeger, Case studies for method and tool evaluation. *IEEE Software*, 12 (1995), 52-62.

[51] C. Low Graham and D.R. Jeffery, Function Points in the Estimation and Evaluation of the Software Process. *IEEE Transactions on Software Engineering*, 16 (1990), 64.

[52] A. MacCormack, C.F. Kemerer, M. Cusumano, and B. Crandall, Trade-offs between productivity and quality in selecting software development practices. *IEEE Software*, 20 (2003), 78-85.

[53] S.G. MacDonell, Comparative review of functional complexity assessment methods for effort estimation. *Software Engineering Journal*, 9 (1994), 107-116.

[54] S. Maffeis, Piranha: A CORBA tool for high availability. *Computer*, 30 (1997), 59-67.

[55] K.D. Maxwell, Collecting data for comparability: benchmarking software development productivity. *IEEE Software*, 18 (2001), 22-26.

[56] K.D. Maxwell, L. Van Wassenhove, and S. Dutta, Software development productivity of European space, military, and industrial applications. *IEEE Transactions on Software Engineering*, 22 (1996), 706-718.

[57] K.J. Moløkken-Østvold and M. Jørgensen, Software Effort Estimation: Unstructured Group Discussion as a Method to Reduce Individual Biases, *The 15th Annual Workshop of the Psychology of Programming Interest Group (PPIG 2003).* Keele University, UK, (2003), 285-296.

[58] J.r. Niels, Putting it all in the trunk: incremental software development in the FreeBSD open source project. *Information Systems Journal*, 11 (2001), 321-336.

[59] A.P. Nikora and J.C. Munson, Developing fault predictors for evolving software systems. *Proc. of The Ninth International Software Metrics Symposium*, (2003), 338-349.

[60] N. Ohlsson, A.C. Eriksson, and M. Helander, Early Risk-Management by Identification of Fault-prone Modules. *Empirical Software Engineering*, 2 (1997), 166-173.

[61] N. Ohlsson, M. Zhao, and M. Helander, Application of multivariate analysis for software fault prediction. *Software Quality Journal*, 7 (1998), 51-66.

[62] T.J. Ostrand, E.J. Weyuker, and R.M. Bell, Predicting the Location and Number of Faults in Large Software Systems. *IEEE Transactions on Software Engineering*, 31 (2005), 340-355.
[63] G.F. Pfister, In search of clusters, Prentice Hall, Upper Saddle River, NJ, (1998).
[64] M. Pighin and A. Marzona, An empirical analysis of fault persistence through software releases. *Proceedings of the International Symposium on Empirical Software Engineering*, (2003), 206-212.
[65] M. Pighin and A. Marzona, Reducing Corrective Maintenance Effort Considering Module's History. *Proc. of Ninth European Conference on Software Maintenance and Reengineering*, (2005), 232-235.
[66] Y. Ping, T. Systa, and H. Muller, Predicting fault-proneness using OO metrics. An industrial case study. *Proc. of The Sixth European Conference on Software Maintenance and Reengineering*, (2002), 99-107.
[67] C. Robson, Real world research: a resource for social scientists and practitioner-researchers, Blackwell Publishers, Oxford, UK; Madden, Mass., (2002).
[68] T.L. Saaty and L.G. Vargas, Models, methods, concepts & applications of the analytic hierarchy process, Kluwer Academic Publishers, Boston, (2001).
[69] M. Shepperd and M. Cartwright, Predicting with sparse data. *IEEE Transactions on Software Engineering*, 27 (2001), 987-998.
[70] H.M. Sneed, Measuring the performance of a software maintenance department. *1st Euromicro Conference on Software Maintenance and Reengineering*, (1997), 119-127.
[71] I. Sommerville, Software engineering, Addison-Wesley, Boston, Mass., (2004).
[72] G.J. Steven, The learning curve: From aircraft to spacecraft? *Management Accounting*, 77 (1999), 64-65.
[73] C. Stevenson, Software engineering productivity: a practical guide, Chapman & Hall, London, (1995).
[74] H. SungBack and K. Kapsu, Identifying fault-prone function blocks using the neural networks - an empirical study. *IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, 2 (1997), 790-793.
[75] M. Svahnberg, Supporting software architecture evolution: architecture selection and variability, Ph.D. Thesis, Blekinge Institute of Technology, (2003).
[76] C.R. Symons, Software sizing and estimating: Mk II FPA (function point analysis), Wiley, New York, (1991).
[77] P. Tran and R. Galka, On incremental delivery with functionality. *Computers and Communications, 1991. Conference Proceedings., Tenth Annual International Phoenix Conference on*, (1991), 369-375.

[78] M. van Genuchten, Why Is Software Late? An Empirical Study of Reasons for Delay in Software Development. *IEEE Transactions on Software Engineering*, 17 (1991), 582-591.

[79] C. Wohlin, P. Runeson, M. Höst, M. Ohlsson, B. Regnell, and A. Wesslen, Experimentation in software engineering: an introduction, Kluwer, Boston, (2000).

[80] T.P. Wright, Factors affecting the costs of airplanes. *Journal of Aeronautical Sciences*, 3 (1936), 122-128.

[81] H. Yu and A. Vahdat, Building replicated Internet services using TACT: a toolkit for tunable availability and consistency tradeoffs. *2nd International Workshop on Advanced Issues of E-Commerce and Web-Based Information Systems*, (2000), 75-84.

[82] W.D. Yu, D.P. Smith, and S.T. Huang, Software productivity measurements. *Proceedings of the 15th Annual International Computer Software and Applications Conference COMPSAC '91.*, (1991), 558-564.

[83] M. Zhao, C. Wohlin, N. Ohlsson, and M. Xie, A comparison between software design and code metrics for the prediction of software fault content. *Information and Software Technology*, 40 (1998), 801-809.

[84] S. Zhong, T.M. Khoshgoftaar, and N. Seliya, Analyzing software measurement data with clustering techniques. *IEEE Intelligent Systems*, 19 (2004), 20-27.

[85] S. Zhong, T.M. Khoshgoftaar, and N. Seliya, Unsupervised learning for expert-based software quality estimation. *Proceedings of the Eighth IEEE International Symposium on High Assurance Systems Engineering*, (2004), 149-155.

Paper I

Software Development Productivity on a New Platform - an Industrial Case Study

Piotr Tomaszewski, Lars Lundberg

Information and Software Technology (2005), Vol 47/4 pp. 257-269

Abstract

The high non-functional requirements on mobile telecommunication applications call for new solutions. An example of such a solution can be a software platform that provides high performance and availability. The introduction of such a platform may, however, affect the development productivity. In this study we present experiences from research carried out at Ericsson. The purpose of the research was productivity improvement and assessment when using the new platform. In this study we quantify and evaluate the current productivity level by comparing it with UNIX development. The comparison is based on two large, commercially available systems. We reveal a factor of four difference in productivity. Later we decompose the problem into two issues: code writing speed and average amount of code necessary to deliver a certain functionality. We assess the impact of both these issues. We describe the nature of the problem by identifying factors that affect productivity and estimating their importance. To the issues identified we suggest a number of remedies. The main methods used in the study are interviews and historical data research.

1. Introduction

Handling the rapid growth of the number of services and subscribers in telecommunication networks has become a very challenging engineering task. Apart from high performance and high capacity the systems that create the infrastructure for the mobile telecommunication network must provide high availability. No downtime is accepted since it results in huge losses. Different telecom system developers deal with high availability in different ways. There are hardware based solutions, which main purpose is to avoid a system crash, as well as software based solutions, that try to handle the situation after a system crash. As a result of the latter approach, a new server platform was introduced by Ericsson. The platform has features that facilitate development of systems with strong high availability requirements. The first experiences after the change of the platform revealed an increase of development time and cost. Both of them affect time-to-market, which is a crucial factor for economically successful software development.

In this paper we look at two large industrial projects at Ericsson. These projects concern the development of similar products. One project uses the new server platform and the other one uses a traditional UNIX development environment. By comparing time reports from the two projects and conducting interviews we were able to assess and compare the productivity for the two environments. A well know problem when dealing with productivity measures is the lack of metrics for measuring the size of the software. The most commonly used metric is the number of code lines [3, 4, 21, 30] We will discuss this and alternative metrics later in the paper. We also consider ways to measure the quality and complexity of the code, and not only the size. Furthermore, we identify some productivity bottlenecks; one category of these bottlenecks has to do with lack of experience. It is well known from many application domains that tacit knowledge, that has to be acquired by long term use and experience, is one source of initial productivity problems when introducing new technology [10] but as people get more used to the new technology these problems should go away.

This study was planned as one of the activities aiming at understanding and improving of the development productivity in the new environment. The initial analysis resulted in formulation of the following research questions:

- *How large is the productivity problem?* – when the study was started there were no hard proofs that productivity has actually decreased when the new platform was introduced. The knowledge about satisfactory productivity level and productivity level on the new platform would allow us to quantify the problem. To achieve that we must measure the productivity of software development on the introduced platform and, to obtain a point of reference, compare it with the productivity level in another project, in which the productivity was perceived as good. As discussed above, two projects were selected for comparison:

- Project A representing UNIX development. The productivity in that project was perceived as satisfactory. This project resulted in Product A.
- Project B representing the development on the introduced platform. It resulted in Product B.

Both systems are large (approximately 40 – 60 man years, 100-200 KSLOC), commercially available, high quality systems that are part of mobile telephone network.

- *Why does the problem occur?* – to solve the problem we must identify the issues that cause it. These issues would indicate areas in which there are opportunities for improvement.
- *What can be done about it?* – for the issues identified we suggested the remedies.

This paper is an improved and extended version of a previous conference paper [27]. In the current paper we present new and additional data and expand the discussion concerning quality aspects and the lessons that can be learned from our case study. We also include a significantly expanded discussion about related work.

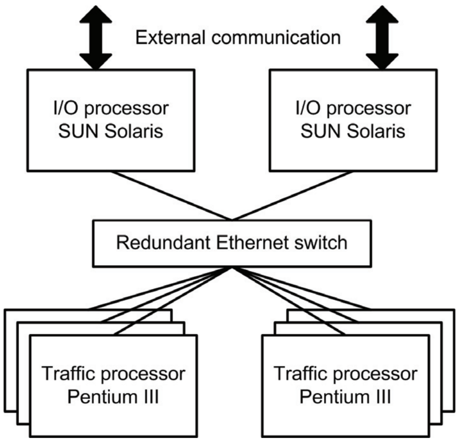
2. Presentation of the platform

The server platform introduced by Ericsson is usually used in real-time telecommunication applications. This type of applications is characterized by very strong non-functional requirements, like the need for scalability, high availability and efficiency. On the other hand market demand for lowering maintenance costs and the best price/performance ratio forces the use of standard hardware components. The new platform meets these requirements. The hardware platform, presented in Figure 1, comprises:

- A number of traffic processors that process pay-load. These are Intel Pentium III processors, and each of them has its own memory

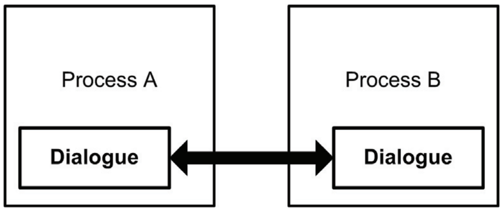
- Four I/O processors responsible for the external communication, maintenance and monitoring of the whole system. These are standard Sun machines running Solaris
- Two Ethernet switches and two separate interconnections via Ethernet networks

Figure 1. The hardware configuration of the platform



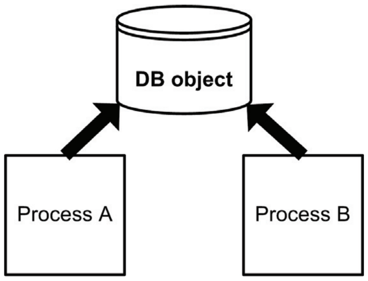
Although the platform offers standard interfaces (APIs) for Java and C++, the programming model is unique. The main execution unit is a process. There are two types of processes, static ones that are always running and dynamic ones that are created and destroyed on request. The inter-process communication is done by dialogue objects or globally accessible database objects. Dialogue objects are used for message passing communication (Figure 2). In the communicating processes two corresponding Dialogue type objects have to be created. They exchange messages using built-in mechanism provided by the platform.

Figure 2. Inter-process communication using dialogue objects



Database objects are the basic units of storage in the internal database. They can be accessed by any process running on the platform. They can therefore be used to implement a shared-memory communication model (Figure 3).

Figure 3. Inter-process communication using database objects



In order to assure an efficient load balance the programmer has a set of methods for specifying the allocation of database objects and processes to processor pools, i.e. sets of traffic processors on which database objects and processes may end up. The load balancing within a pool is done by the platform itself.

The platform facilitates programming of the highly available systems, i.e. every process or database object is automatically replicated on different machines in the cluster – a crash of one of them does not affect the correct operation of the whole system. Additionally the platform has built-in features that allow online upgrades of the applications that operate on the platform.

The two applications examined in our study co-operate within the same system that works in the service layer of the mobile telephony network. In each of them the high availability requirement was provided in a different way. In Product A high availability is assured by a backup server that takes over the work when the main server fails. Product B uses the new platform features to provide high availability.

Both applications have similar design structure. The following subsystems can be identified:

- *Platform*–software that extends functionalities provided by platform
- *Communication*–software responsible for handling of communication protocols

- *Functionalities*–software that contains actual business logic of the application

Both systems are written in C++. To minimize the impact of software reuse on the productivity measurement, only the first versions of both products were taken into account – both were written “from scratch”. Additionally it should be noticed that Product B was one of the first projects done on the new platform by the team of developers examined in the study. Before taking part in the project the project members underwent a training program about the new platform. The training program comprised of a one week long course. Additionally, the developers were provided with a web based tutorial that covered basic issues connected with programming on the new platform. They also produced a number of prototypes to gain practical, “hands-on” knowledge about the platform. According to the majority of the developers the quality of the introduction process was not satisfactory. They suggested that they would benefit significantly from longer and more advanced training.

3. Methods

In this Section we present methods used in the study. Each subsection in this Section has a corresponding subsection in Section 4 where the results are presented.

3.1 Productivity measurement

The first thing that must be done is establishing what productivity is and how it can be measured. The traditional productivity definition as a *ratio of output units produced per unit of input effort* [1] is not easily applicable to software development. The input effort is usually defined as the sum of all resources that were used to produce the output. In the software development the biggest part of whole production cost is the cost of work. Therefore, in the study, person hours were taken as the input unit of effort. It is much more difficult to select the metric for the unit of product. Two perspectives of measuring the size of the system can be identified:

- *Internal viewpoint* (developer’s perspective) – describes the amount of code that must be produced to complete the system
- *External viewpoint* (customer’s perspective) – refers to the amount of functionality provided by the system.

The internal point of view metrics usually measure the physical "length" of the code produced. Typical units of the internal size are number of source lines of code, number of classes or number of functions. Internal size measurements can be easily obtained by the use of automatic tools. The often mentioned weakness of measuring the system size using code lines is that result depends on the coding style - one programmer can write a statement in one line while other can consistently spread it among a number of lines. To check if such a situation took place the ratio of code lines per C++ statement was calculated for both projects. This metric should, to a certain extend, assure that coding style was similar in both projects.
The internal perspective may be confusing, since one platform may be "more productive" when it provides a certain functionality using "less code". The measurement from an internal perspective would not reveal this. However the measurement from external perspective is difficult in real time systems. This measurement should take into account not only the "amount" of functionality but also the complexity, which is very difficult to quantify. Therefore existing functional size metrics, like Function Points, are not recommended for real-time systems size measuring [26]. Instead of measuring we decided to estimate the ratio of functional size of both systems. Since expert judgement is considered as an acceptable way of performing estimations [6, 19, 24], we used it for comparing the functionality of both systems.

The following data concerning the sizes of both projects were collected:

- Number of person hours spent on each project (Hour) – only the development phase of the project was taken into account (design, implementation, testing). Person hours contain designers, testers and managers work hours.
- Number of code lines in each project (SLOC) - we counted only lines with C++ code, comments and blank lines were not counted.
- *SLOC/C++ statement* ratios in both projects
- Number of classes in each project (NoC)
- Ratio of amount of the functionality in both projects (FUNC) – an expert estimation. A total of 6 experts were interviewed for the estimation. All of them had knowledge concerning both projects. The results of the estimation were analysed by three other experts and a consensus was achieved.

3.2 Quality aspects

The main weakness of the size measurement methods is that they do not take any quality factors into account. The productivity can only be "*interpreted in context of overall quality of the product*" [1]. Software product must meet certain quality requirements (minimum acceptable requirements) before the productivity metric can be applied. Therefore, in the study, quality aspects were kept in mind when evaluating productivity. Big differences in any aspect of quality may possibly explain the difference in productivity – in that case lower productivity could be the price for higher quality.

In the study we have considered following quality factors that according to us can have impact on productivity:

- *design quality* – quality of application design and quality of code produced. Better design pays off in testing and maintenance phases and is more likely to be reused in other projects in future and therefore can be considered an added value.
- *final product quality* – qualities actually achieved in the final application. Example of such qualities may be non-functional requirements. High non-functional requirements (security, high availability) can be the important cost driver and therefore can explain relatively high development time.
- *quality of development process* – high quality of development process does not guarantee high quality of final product but makes achieving it more probable.

It is obvious that the lines of code are affected by a number of things. It is more difficult to produce well designed, structured and organized code. From the software metrics suggested by [9, 11] that are applicable for object oriented systems we selected those that were proven to have impact on quality of the system [2, 7, 8, 11]. Therefore, a number of metrics, describing different aspects of design quality, were applied:

- *McCabe Cyclomatic Complexity* (MCC) metric [11]. This metric measures number of linearly independents paths through the function. According to [13] "*Overly complex modules are more prone to error, are harder to understand, are harder to test, and are harder to modify.*"
- *Lack of Cohesion* (LC). It measures *"how closely the local methods are related to the local instance variables in the*

class" [11]. The idea is to measure to what extent the class is a single abstraction. In [8] Chidamber, Darcy and Kemerer proved that the Lack of Cohesion metric has impact on productivity. According to them the implementation of classes with high LC was difficult and time consuming. In the study we counted LC using method suggested by Graham [14, 17] which gives normalized values of LC (0%-100%). We considered normalized values more applicable for comparison purposes.

- *Coupling* (Coup) [9, 11], the metric measuring the number of classes the class is coupled to. This metric allows assessing the independence of the class. Coupling is widely recognized as important factor influencing productivity [8] and fault proneness [2, 7].
- *Depth of Inheritance Tree* (DIT) measures how deep in inheritance hierarchy the class is. According to [9] high DIT values resulting in higher complexity make prediction of class behaviour more difficult. In [2] Basili, Briand and Melo proved that there is relation between DIT and fault proneness of the class.
- *Number of Children* (NC) defined as "*number of immediate subclasses subordinated to a class in the class hierarchy*" [9]. In [9] authors claim that classes with huge amount of subclasses have potentially bigger impact on the whole design and therefore they require more testing (their errors are propagated).

Other quality aspects, like quality of the final product or quality of the development process are difficult to quantify. Therefore the opinions about both of them were collected during interviews. Twenty developers were interviewed in a semi-formal manner [22]; later an additional ten informal interviews were performed. When it comes to the quality of the final product the interviews mainly concerned comparison of non-functional requirements put on the systems. The development process quality discussions focused on the amount of quality assurance activities, like testing, inspections or the level of detail in project documentation.

3.3 Productivity bottlenecks

The next step after estimating the size of the productivity was the identification of issues that affect productivity. We were aiming at localization of productivity problems on the new platform. Therefore we focused on identifying the productivity bottlenecks only in Project B. Since productivity can be affected by factors of different nature the decision was made to take into account all the areas of productivity bottlenecks localizations, which are [15]:

- *People*–issues connected with competence level of people involved in the development process
- *Processes*–issues connected with work organization characteristics
- *Technology*–issues connected with technology used, in our case mainly different platform shortcomings

In order to identify the issues that affect the productivity we performed interviews with 20 developers directly involved in the development on the new platform. The interviews were semi-formal [22]. Apart from reporting productivity bottlenecks, the interviewees were asked for suggestions of improvements/remedies. Basing on data collected during the interviews we created a list of productivity bottlenecks.

The additional analysis was performed to estimate to what extend each of the identified issues affects the productivity. The method selected for that is called Analytic Hierarchical Process - AHP [23]. Each respondent did pair-wise comparisons between different issues and basing on those comparisons the final importance of the different issues was calculated [23]. The comparison was based on the question "Which alternative do you feel affects productivity more?". The AHP questionnaires were distributed among the same group of people that took part in the interviews. The AHP method made it possible to build the hierarchy of the importance for each respondent and assigning weights of importance to alternatives. An example of importance vector for 6 issues is presented in Table 1. Such a vector is an outcome of an AHP analysis performed by a single respondent.

Table 1. Example of the importance vector (N.B. importance figures always sum up to 100%)

Bottleneck	Importance
Issue 1 (e.g. voice conversation)	30%
Issue 2 (e.g. games)	10%
Issue 3 (e.g. SMS)	20%
Issue 4 (e.g. alarm)	15%
Issue 5 (e.g. calculator)	15%
Issue 6 (e.g. tunes)	10%

To ensure that the results really reflect the opinions of the respondents each of them was presented with an individual importance vector and was allowed to change/adjust it. The individual importance vectors were used to create the importance vector of the whole group. It was created by calculating an average importance weight for each issue.

The AHP analysis was performed by the same group of developers that took part in the interviews during which the productivity bottlenecks were identified.

4. Results

Each subsection in this Section presents results of application of the methods described in corresponding subsection in Section 3.

4.1 Productivity measurement

Due to the agreement with the industrial partner all measurement results will be presented either as [Project A / Project B] ratios or [Project A - Project B] differences. No results will be presented as absolute values.

The results of the size and the effort measurements taken on both projects are summarized in Table 2. The table presents relative values only.

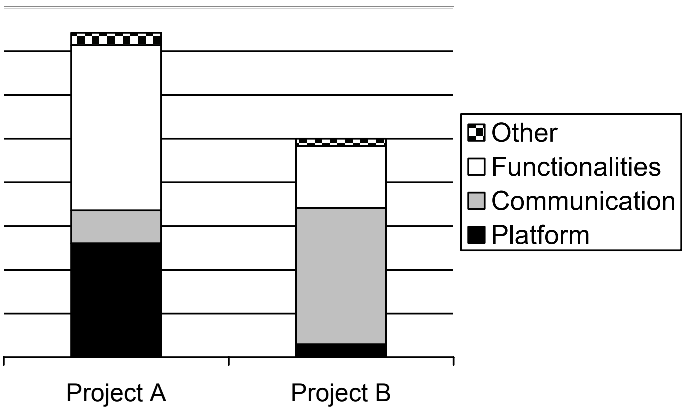
Table 2. Project size and effort ratios

Metric	Project A/Project B
Code lines (SLOC)	1.5
Number of classes (NoC)	1.5
Functional size (FUNC)	3.0
Person hours (Hour)	0.7

To check if similar coding style was used in both projects, the average number of code lines/C++ statement was calculated for both projects. It turned out to be similar. In Project A it was 2,22 lines/statement, while in Project B it was 2,42 lines/statement. Therefore we considered code lines comparable between both projects.

The significant difference (factor of 2) between the internal viewpoint measurement (code lines, classes) and the external viewpoint measurement (functionalities) suggests that UNIX is more successful in providing functionality – on average half of the code is required to provide a certain functionality. To explain that phenomena the structure of both projects was examined. The structure is presented on Figure 4. There is significant difference in the distribution of code in the subsystems. The large difference in amount of code in the Platform part can easily be explained. In Project A the own platform extension was developed on the top of the system, which was not the case in Project B. The subsystem responsible for the business logic (Functionalities) is about 3 times bigger in Project A which meets the expectations – according to the experts there is 3 times more functionality in Project A. In the unknown part there is a large difference in the amount of code for communication handling. We performed a number of interviews to explain the fact that more code is needed per average functionality. During the interviews we presented the diagram from Figure 4. Three possible explanations of the phenomena were mentioned:

Figure 4. Product structure



- In UNIX the support for communication is better – i.e. there are more third party libraries available or the system itself provides more
- In general it is more difficult to design systems on the new platform. The design is more complex, more code has to be written to complete a certain functionality
- The platform lacks certain tools (for example a good debugger) and therefore more code must be written to compensate for that (i.e. debug printouts that help in tracing faults)

From the information about the size and the effort the development productivity ratios were calculated. The results are presented in Table 3.

Table 3. Productivity ratios

	Project A / Project B
SLOC/PH	2.15
NoC / PH	2.15
FUNC/PH	4.30

4.2 Quality aspects

To compare the design quality in both systems a number of measurements (described in Section 3.2) were done. Each metric will be analyzed separately. Since all the metrics are done either on the function or on the class level (values are obtained either for each function or for each class) in order to compare two systems we will examine the distribution of the values in each of them. For each metric we will present mean, median, standard deviation and minimal and maximal values obtained. This way of describing measurements was presented in [2]. We will also present histograms describing in graphical form how many percent of the entities (classes, functions) in the system have certain value of the metric. Since it is sometimes difficult to assess if the obtained values are typical or not, where it is possible we will add a column where the corresponding values from the study described in [2] will be presented. Other examples of such values can be found in [7-9, 20]. We selected values from [2] for comparison purposes because measurements there were taken on relatively large amounts of C++ classes, which is similar to our study, and the same types of data were collected as in our study. Values from other studies mentioned [7-9, 20] will be used when discussing the findings.

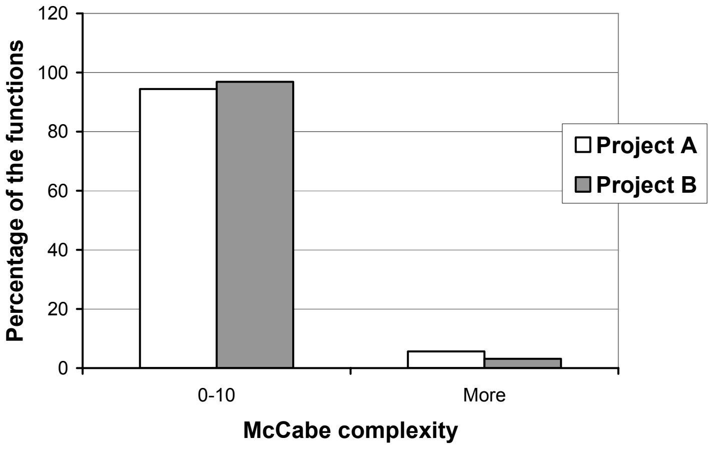
The first metric applied was McCabe Complexity. Results were obtained on function level and are presented in Table 4.

Table 4. McCabe complexity

	Project A	Project B
Mean	3.3	2.5
Median	1	1
Maximum	125	96
Minimum	0	1
Std. deviation	5.8	5.0

According to [11] McCabe Complexity of the function should not be higher then 10, otherwise the function is difficult to test. We examined the data from that perspective (Figure 5).

Figure 5. **McCabe Complexity - distribution**



In both projects the vast majority of the functions (95% in Project A and 97% in Project B) have complexity values within 0-10 range and therefore we consider both projects similar from that perspective.

The remaining measurements gave the results on the class level.

The second metric applied was the Lack of Cohesion. The results are summarized in Table 5.

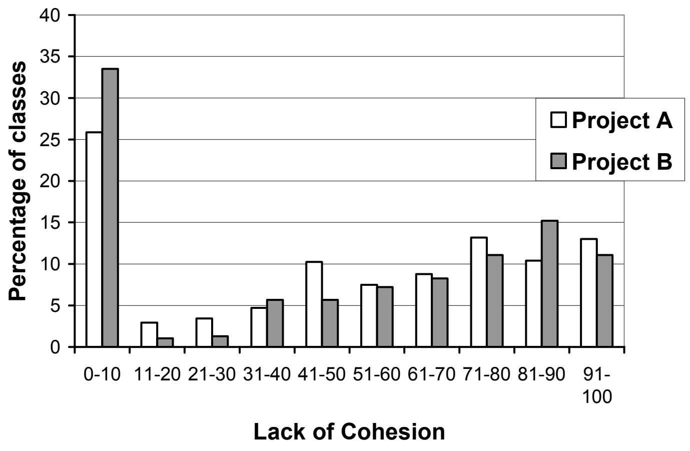
Table 5. **Lack of Cohesion**

	Project A	**Project B**
Mean	49.9	46.8
Median	57	57
Maximum	100	100
Minimum	0	0
Std. deviation	35.5	37.4

The distribution of LC values between classes of both systems is presented on Figure 6.

We can observe that trends in distribution are similar in both systems. Mean and median values are also similar. Therefore we consider both systems similar from a Lack of Cohesion viewpoint.

Figure 6. **Lack of Cohesion - distribution**



The next metric applied was Coupling. The results are summarized in Table 6.

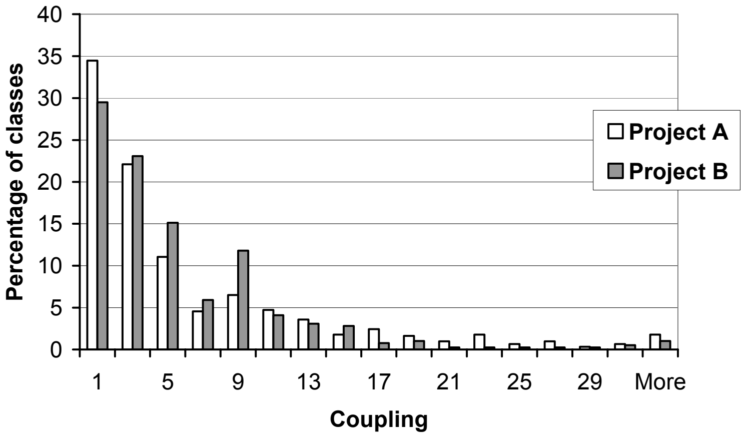
Table 6. **Coupling**

	Project A	Project B	*Ref.[2]*
Mean	6.0	5.1	*6.80*
Median	3	3	*5*
Maximum	82	35	*30*
Minimum	0	0	*0*
Std. dev.	8.6	5.7	*7.56*

In order to compare to what extend coupling values obtained in both projects are similar we looked for data describing coupling in typical projects. The mean coupling values reported in [2, 7-9, 20] are usually between 5 and 7. The distribution of coupling values in the project classes is presented on Figure 7.

We consider the distribution of coupling values similar for both projects. The values from Table 5 place both our projects among typical projects from coupling point of view. Therefore we consider them similar from that perspective.

Figure 7. Coupling – distribution



The application of the Depth of Inheritance Tree metric gave results presented in Table 7.

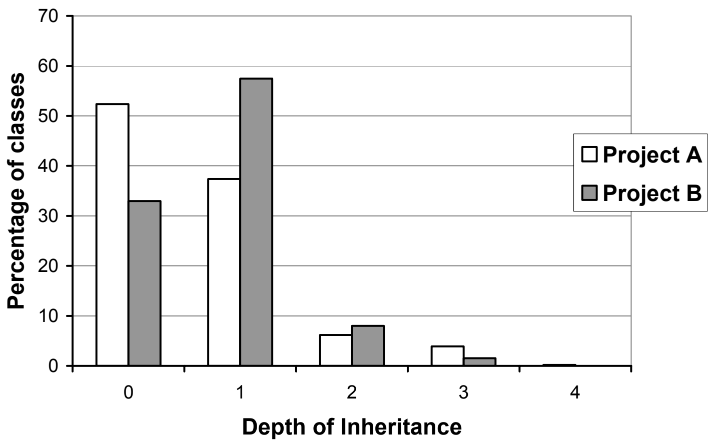
Table 7. Depth of Inheritance

	Project A	Project B	*Ref.[2]*
Mean	0,62	0,78	*1.32*
Median	0	1	*0*
Maximum	4	3	*9*
Minimum	0	0	*0*
Std. dev.	0.8	0.7	*1.99*

Comparing to [2, 9] the mean, median and standard deviation values obtained in the study are much lower. The distribution of DIT values is presented on Figure 8.

From Figure 8 it can be observed that the difference between projects is caused by about 20% of the classes that in Project B have depth 1 while in Project A the depth is 0. We consider that difference rather small especially because comparing to other studies the average DIT values in both examined projects are small.

Figure 8. **Depth of Inheritance - distribution**



The last metric applied was Number of Children (NoC). The results are summarized in Table 8.

Table 8. **Number of Children**

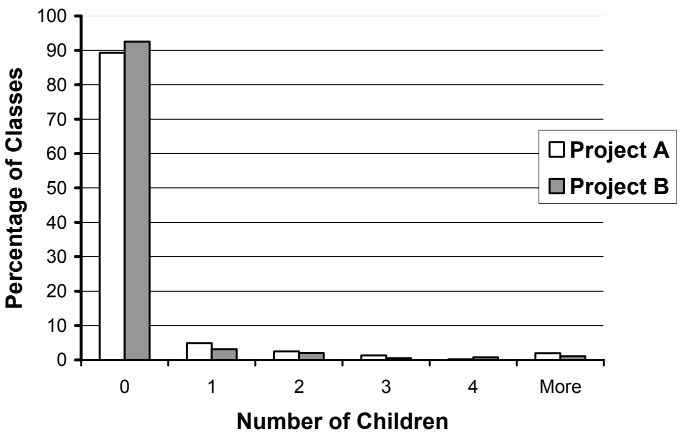
	Project A	**Project B**	***Ref.[2]***
Mean	0.35	0.19	*0.23*
Median	0	0	*0*
Maximum	24	9	*13*
Minimum	0	0	*0*
Std. dev.	1.8	0.9	*1.54*

Compared to the reference project we can observe that Project A has similar characteristics, while in Project B mean value is about twice as small. The median value is in both cases equal 0. The distribution of NoC values is presented on Figure 9.

As it can be seen on Figure 9 over 90% of classes in both projects has NoC equal to 0. The difference in mean value is caused mostly by 2% of classes that in Project A have 1 child while in Project B they have 0. Since the distribution is almost identical we consider both projects similar from NoC perspective.

After analyzing all the measurements taken we can not observe any major difference in design quality between the two projects. Therefore we consider the design quality similar in both projects.

Figure 9. Number of Children – distribution



The measurements presented above describe only design quality. Equally important for the productivity assessment are the quality aspects of the system developed (i.e. non-functional requirements) and the quality aspects of the development process (i.e. number of the quality assurance activities). The overall impression was that Project B was more ambitious in terms of the process quality aspects. Due to the relative novelty of the platform, and in order to minimize anticipated influence of the learning effect, much effort was put on quality assurance (inspections, testing) and documentation activities. The non-functional requirements put on the systems were similar, but during interviews the designers mentioned that they believed the non-functional characteristics achieved in the project were better in the system developed on the new platform.

Both systems met the requirements that were put on them.

4.3 Productivity bottlenecks

The nature of the productivity problems was identified during interviews. As a result of them the list of nine issues that negatively affect the productivity in the Project B was formulated:

- *Not enough experience sharing (i.e. seminars, meetings) and training activities* – this problem impacts the productivity in two ways:
    - The skills of the staff are not developed as quickly as it would be possible.

    - The "reuse" of ideas is smaller. Better exchange of the information would prevent the situation when two people invent a solution to the similar problem separately.

- *Staff competence level* – the novelty of the platform causes overhead connected with learning. This is important because, according to interviewees, the start up time on the new platform is relatively long comparing to e.g. UNIX.
- *Quality of the new platform's documentation* – Certain problems connected with the documentation's quality and availability were mentioned. According to interviewees a better structured and updated source of technical information on the new platform would positively affect the speed and quality of software development.
- *Runtime quality of the new platform* – low runtime quality of the platform makes development and testing more difficult. Relatively high amount of faults was classified as platform related, which means that they occurred due to the platform error. Therefore the scope of potential fault localizations is bigger compared to more mature platforms, which adds a lot of complexity to testing.
- *The platform interface (API) stability* – unexpected changes in the API in the different platform releases result in the need of "redoing" parts of the system to meet the new API specification.
- *Lack of target platform in the design phase* – Designers do not have access to a real cluster. Instead they are working with an emulator, which is not 100% compatible with the target platform. The faults caused by the incompatibility are discovered later, in testing phase, when the cost of correction is much higher.
- *Too much control in the development process* – due to the novelty of the platform there was a strong pressure on the quality assurance activities, like large amount of inspections or detailed documentation produced on quite low level. It resulted in heavy and costly development processes.
- *Unstable requirements* – the unstable requirements make it necessary to redesign parts of the system, sometimes in later stages of the project, which is extremely costly.
- *Too optimistic planning and too big scope of the projects* – big scope of projects combined with the novelty of the platform may result in long lead-time. Long lead-time projects are much more prone to change requests due to changes of market demands.

It is noticeable that the bottlenecks identified have different nature. This suggests that the productivity problem is complex and depends on many factors.

The interviewees were asked to prioritize the issues to show which, according to them, affects productivity most. Thanks to the use of the AHP method we were able to create individual importance vectors. For each interviewee we calculated the weights of importance that the interviewee assigned to the issues. Weights were normalized, so they always sum up to 100%. Based on the individual importance vectors the average vector was calculated (see Section 3.3). Table 9 presents the final ranking of the issues affecting the productivity of the software development on the new platform.

Table 9. Productivity bottlenecks prioritization

Bottleneck	Importance
Staff competence level	22%
Unstable requirements	16%
Not enough experience sharing and training activities	13%
Quality of platform's documentation	10%
Runtime quality of the platform	10%
Too much control in development process	9%
Too optimistic planning, too big scope of projects	8%
Lack of target platform in the design phase	7%
Platform interface stability (API)	5%

5. Discussion

5.1 Related work

In the literature the software productivity research has gone in two main directions [21]:
- productivity measurement
- identification of factors that affect productivity

Main point of concern of researchers dealing with measurements is the lack of a generally accepted metric for measuring software size. Often the simplest method of measuring the size is used, which is number of lines of code [3, 4, 21, 30]. Maxwell, Van Wassenhove and Dutta [21] performed a study that involved an evaluation of the lines-of-code productivity metric. They compared it with Process Productivity, the complex metric that includes management practices, level of programming language, skills and experience of the development team members and the complexity of the application type. After examining 99 projects the authors concluded that the simple lines-of-code metric was superior to the Process Productivity metric.

The second direction of the productivity research was identification of factors that affect the productivity. The continuously increasing cost of software development has made productivity improvement a very popular topic. It is usually seen as the way to decrease cost and improve delivery time. The fact that software projects are often late and over budget [3, 4, 28] makes the problem important. A number of research studies were done within that domain. Yu, Smith and Huang [30] created a universal framework for the improvement process. They identified three steps of the improvement process: measurement, analysis and improvement. The purpose of the first step was to find out where the project stands, the second step was devoted to identification of the factors affecting productivity and the third one was supposed to minimize their impact. The framework suggested corresponds well with our strategy. The authors presented the example of the study aiming into quantification and improvement of the productivity in a project from AT&T Bell Laboratories. One interesting finding in that project is that the issues affecting productivity are, to a large extent, similar to the ones described in our research. Among the issues that were ranked the highest belong the requirements stability and the staff experience, which is similar to the results obtained in our research.

Many researchers have observed, documented and tried to solve the productivity problem when adopting new technology [10, 12, 16, 18, 29]. Ever since the learning effect [25] was described most researchers agree that some of the initial productivity problems fade away with time due to growing experience and maturity of the organization. The question of how to make that time as short as possible remains unanswered. In [10] Edmondson, Winslow, Bohmer and Pisano stress the importance of having as much of the knowledge about new technology codified as possible. The more of the knowledge about new technology is tacit the bigger problem is to introduce it seamlessly. In the examined project we have experienced the situation where lack of easily accessible source of information about the technology affected the productivity. Fisher and Wesolkowski [12] present another view of the initial knowledge problem. It might be extremely difficult to increase the competence level of the staff it the staff does not want it. According to them one of the key points is the motivation and attitude issue – they quote the results of the study showing that only 15% of the population is enthusiastic when it comes to new technology adoption, 85% is more or less hesitant to it. Harvey, Lefebvre and Lefebvre [16] analyzed 100 companies to find out how the companies that successfully adopted new technologies differed from the others. One of their findings was the huge role of management in such process. The importance of management's role in facilitating the process of new technology adoption is also stressed by Vaneman and Triantis [29]. Among the bottlenecks we have identified there are issues concerning project planning and organization which proves that the role of management was recognized by our interviewees.

Other researchers also put a lot of effort in localizing issues affecting the productivity. Productivity issues are often referred to when discussing delays in software deliveries [3, 4, 28]. Although lead-time and productivity do not always correlate (i.e. adding a new developer may decrease the productivity but improve lead-time), the improvement of productivity is usually seen as a way to decrease the development time. Blackburn and Scudder [3] identified number of factors that reduce development time. The authors examined the data from 40 different projects. As the most promising technique of development-time reduction the Reuse of Code was considered. The second most promising technique was competence development, which is similar to the results obtained in our study. Moreover in the paper the authors report that "*managers are continually frustrated by changing requirements*" – which directly corresponds to "Unstable requirements", the issue which is second on our priority list.

A direct comparison of results obtained in the research studies presented above with results obtained in our study may seem to be inappropriate – these were usually surveys in which a large number of projects were examined, and therefore the results are on higher level of abstraction and to large extend their generalization would be justified. However the fact that conclusions concerning productivity bottlenecks seem to be similar may indicate that the problems identified are rather common for the situation when new technology is adopted.

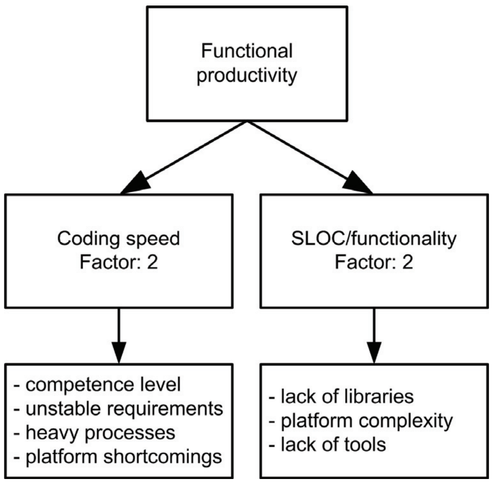
5.2 Productivity level

The results obtained in the study describe the current productivity level in the software development on the new platform. Compared to the UNIX platform a factor of four in the productivity measured from an external perspective was identified. It can be later decomposed into two issues:

- *code writing speed* - on the new platform the code is written slower, on average it takes twice as much time to deliver 1 line of code (internal perspective measurement measures the speed of delivering the code)
- *code line/functionality* - in the new platform the average number of code lines per functionality is bigger. Since it holds the remaining part of responsibility for the productivity level its impact may be counted as a factor of two. It is supported by measurements – on average we need twice as much code per functionality on the new platform.

The issues that affect code writing speed are presented in the Table 9. The ones that have impact on high SLOC/functionality ratio in the new platform are lack of third party libraries, missing tools (e.g. debuggers) and platform complexity. The distribution of responsibility for productivity level is summarized on Figure 10.

Figure 10. Productivity problem decomposition



5.3 Productivity improvement

The main reason for bottlenecks identification is to find out where the application of remedies would bring the best results. In our study it seems obvious, since three highest ranked issues hold over 50% of responsibility for the productivity level, and two of them are related to competence. It is not surprising – competence is usually a problem when new technology is introduced, especially a complex one with long start up time. The picture changes if localizations of bottlenecks, suggested by Hantos and Gisbert [15], are considered. Table 10 presents this.

A surprising finding is that platform related issues were rated quite low both individually and as a group. One possible explanation could be that the respondents focused on issues that are internal to the organization where the study was performed. Maybe they tried to focus on issues that directly depend on them. Platform quality issues do not belong to that group of issues while competence and work organization issues do. Another explanation would be more straightforward – platform quality is not the main problem. Additionally it should be noticed that the competence issues' importance is most prone to change over time. It should decrease with time when the developers will gain experience. However, it is still a valid issue when the platform is being introduced to a new organization.

Table 10. Bottleneck localizations importance

Localization	**Bottlenecks**	**Importance**
People	Staff competence level Not enough experience sharing and training activities	35%
Processes	Unstable requirements Too much control in development process Too optimistic planning, too big scope of projects Lack of target platform in design phase	40%
Technology	Quality of platform's documentation Runtime quality of the platform Platform interface stability (API)	25%

Table 10 clearly suggests that each group of the issues holds relatively large responsibility for current productivity level – no "silver bullet" solution to the problem can be expected. Therefore, remedies to all the productivity bottlenecks were presented. It is a subject to further research to suggest the order in which they should be applied. Their possible effectiveness, estimated in this study, is only one of the factors that should be taken into consideration when making that decision –

others are the cost of the remedy, risk connected with its introduction or the time after which the remedy application will pay off.

There are three ways to improve productivity [5]: work faster, work smarter and work avoidance. Faster work can be obtained by development of skills. Therefore, the following skill development activities were suggested:

- Good introduction process - The introduction process would familiarize the staff with the new technology and minimize the overhead connected with learning
- Continuous skills development processes, like an advanced course on programming on the introduced platform, seminars, meetings and technical discussions would give the developers a chance to share experiences and spread knowledge among team members.
- Better management of company knowledge
    - Set of patterns - set of easily applicable solutions to the common problems.
    - Better documentation of the problems encountered would help to avoid making the same mistakes in the future.

The second way of improving productivity, smarter work, suggests better work organization. The following possible remedies were suggested to the problems identified:

- Lack of target platform in the design phase
    - More automated functional tests run overnight would provide immediate feedback concerning the application's behaviour on the real platform. The faults would be detected earlier which would save a lot of time connected with e.g. fault localization.
    - Introduction of the target platform in the analysis/design phase would be an expensive solution, but would provide designers with immediate and precise feedback.
    - More prototyping in the early stages of the project – some problems that are encountered in the implementation phase would never occur if the ideas were tested on prototypes earlier in the design phase.
    - Shorter time between functionality development and testing would result in faster fault detection. It would be easier for designer to localize the problem in code that was recently produced than in code produced long time ago.
- Too optimistic planning, too big scope of projects and unstable requirements - a smaller scope of the projects would result in

more stable requirements. The main cause of requirements change is the change of market demands. If the project had shorter lead-time the probability of having change requests would be smaller and their impact would be minimized.

The last way of productivity improvement, namely "work avoidance", refers to the idea of acquiring the solution instead of developing it. One big issue in that topic is the current lack of third party components for typical purposes, like handling of the standard communication protocols. Other issues are the platform quality issues. The following improvements of the platform were suggested to the platform developer:

- Runtime quality of the platform – the focus should be put on providing quality to existing functionalities of the platform instead of developing new features.
- Platform programming interface (API) stability – a "road-map" describing which parts of the platform are subject to change would solve the problem. In that case, the designers would not be surprised by API changes.
- Quality of platform's documentation – Update documentation – features that are not documented can not be used, so there is no point in developing new features if their description is not added to the documentation.

6. Conclusions

The objective of the study was to examine the impact of the change of the platform on the software development from the development productivity point of view. To achieve that we at first quantified the productivity of the software development on the introduced platform, then identified the nature of the problems encountered and finally we suggested some productivity improvement methods. The quantification was done by comparison with an other project, in which the productivity was perceived as good.

The measurement from the functional perspective revealed the factor of four difference between the development productivity in the two projects. Examination of the product structure and measurement from the internal perspective allowed us to select two factors that result in the factor of four difference between productivity. These are the code writing speed, twice as small in the new platform, and the average amount of code necessary to provide certain functionality, about twice as big in the new platform.

In order to check to what extend the difference in productivity between the two projects could have been caused by the difference in quality we examined different aspects of quality. We found out that in terms of the design quality both projects are similar but in terms of process quality the project done on the new platform was more ambitious. In terms of non-functional requirements there was no significant difference in the requirements put on the system but the developers believed that the non-functional characteristics actually achieved in the project done on the new platform were better.

Later we identified factors that affect productivity and we estimated their importance. It turned out that the problem is complex - there are many factors of different origins that affect the current productivity level. The learning effect, caused by the introduction of the new platform, had the relatively highest impact on the code writing speed. However, other factors like the platform quality and the work organization also have a significant impact. Lack of third party libraries for the new platform and the platform complexity result in larger amount of code necessary to deliver functionality, compared to the Unix environment.

Due to the problem complexity the suggestion of one, “silver bullet” solution was impossible. Therefore we suggested a number of remedies to the issues identified. The individual importance of the issue the remedies address will be one of the factors taken into consideration when deciding the order in which the remedies will be applied.

We believe that some general lessons can be learned from our study. Problems, similar to the ones we have described, may be experienced every time a company decides to change platform or technology. If the change is from a standard, widely used environment, to one used mainly in specialized application domains, as in our case, it seems very likely that the bottlenecks we have identified may appear. The magnitude of their impact may however differ, due to their dependence on individual settings like the kind of technology introduced, experience of the staff, characteristics of projects done in the company and many others.

One of the main issues identified in our study, namely the learning effect, is always present when a new platform is introduced. If the platform has a unique programming model, the learning curve can be very steep. Appropriate training activities, although expensive, may bring significant savings on the project cost. This was clearly pointed by our interviewees who ranked the competence issues highest.

Therefore it seems to be extremely important that the developers are provided with a good source of information about the new platform. It not only minimizes the learning effort but also affects the coding speed even after the developers have gained a certain level of experience.

Due to the learning effect projects done using the new technology are prone to delays. In order to minimize the impact of limited experience on the quality of the product, often the amount of quality assurance activities is higher than normally, which makes the project even more delayed. In the systems like the ones we have examined it immediately results in unstable requirements, which according to our interviewees have significant impact on productivity. Therefore the scope of initial projects should be limited, if possible.

Another issue that may be a consequence of introduction of a very specialized platform is the lack of convenient add-ons that are available on standard, widely used platforms. The number of available tools or third party libraries is likely to be limited since the relatively small number of potential customers that would buy those makes their development questionable from economical perspective. Considering that among those there are debuggers, profilers or CASE tools as well as software libraries the impact of their absence should not be underestimated.

7. Acknowledgements

The authors would like to thank all the members of Ericsson's staff thanks to whose participation, patience and will to help the whole study was possible as well as Daniel Häggander and Johan Schubert from BTH for their help, contribution and valuable comments.

This work was partly funded by The Knowledge Foundation in Sweden under a research grant for the project "Blekinge - Engineering Software Qualities (BESQ)" (http://www.ipd.bth.se/besq).

8. References

[1] *IEEE standard for software productivity metrics*, in *IEEE Std 1045-1992*. (1993). 2.

[2] V.R. Basili and L.C. Briand, A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22 (1996), 751-762.

[3] J.D. Blackburn and G.D. Scudder, Time-based software development. *Integrated Manufacturing Systems*, 7 (1996), 60-66.

[4] J.D. Blackburn, G.D. Scudder, and L.N. van Wassenhove, Improving Speed and Productivity of Software Development: A Global Survey of Software Developers. *IEEE Transactions on Software Engineering*, 22 (1996), 875-886.

[5] B. Boehm, Managing Software Productivity and Reuse. *Computer*, 32 (1999), 111-114.

[6] B.W. Boehm, Software engineering economics, Prentice-Hall, Englewood Cliffs, N.J., (1981).

[7] L.C. Briand, J. Wust, S.V. Ikonomovski, and L. H., Investigating quality factors in object-oriented designs: an industrial case study. *Proceedings of the 1999 International Conference on Software Engineering*, (1999), 345-354.

[8] S.R. Chidamber, D.P. Darcy, and C.F. Kemerer, Managerial use of metrics for object-oriented software: an exploratory analysis. *IEEE Transactions on Software Engineering*, 24 (1998), 629-639.

[9] S.R. Chidamber and C.F. Kemerer, A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20 (1994), 476-494.

[10] A.C. Edmondson, A.B. Winslow, R.M.J. Bohmer, and G.P. Pisano, Learning How and Learning What: Effects of Tacit and Codified Knowledge on Performance Improvement Following Technology Adoption. *Decision Sciences*, 34 (2003), 197-223.

[11] N.E. Fenton and S.L. Pfleeger, Software metrics: a rigorous and practical approach, PWS, London; Boston, (1997).

[12] W. Fisher and S. Wesolkowski, How to determine who is impacted by the introduction of new technology into an organization. *Proceedings of the 1998 International Symposium on Technology and Society, 1998. ISTAS 98. Wiring the World: The Impact of Information Technology on Society.*, (1998), 116-122.

[13] S. Gascoyne, Productivity improvements in software testing with test automation. *Electronic Engineering*, 72 (2000), 65-67.

[14] I. Graham, Migrating to object technology, Addison-Wesley Pub. Co., Wokingham, England; Reading, Mass., (1995).

[15] P. Hantos and M. Gisbert, Identifying Software Productivity Improvement Approaches and Risks: Construction Industry Case Study. *IEEE Software*, 17 (2000), 48-56.

[16] J. Harvey, L.A. Lefebvre, and E. Lefebvre, Exploring the Relationship Between Productivity Problems and Technology Adoption in Small Manufacturing Firms. *IEEE Transactions on Engineering Management*, 39 (1992), 352-359.

[17] B. Henderson-Sellers, L.L. Constantine, and I.M. Graham, Coupling and cohesion (towards a valid metrics suite for object-oriented analysis and design). *Object Oriented Systems*, 3 (1996), 143-158.

[18] M. Huggett and S. Ospina, Does productivity growth fall after the adoption of new technology? *Journal of Monetary Economics*, 48 (2001), 173-195.

[19] R.T. Hughes, Expert judgement as an estimating method. *Information and Software Technology*, 38 (1996), 67-76.

[20] X. Li, Z. Liu, B. Pan, and D. Xing, A measurement tool for object oriented software and measurement experiments with it, *10th International Workshop New Approaches in Software Measurement*, Springer-Verlag, Berlin, Germany, (2001), 44-54.

[21] K.D. Maxwell, L. Van Wassenhove, and S. Dutta, Software development productivity of European space, military, and industrial applications. *IEEE Transactions on Software Engineering*, 22 (1996), 706-718.

[22] C. Robson, Real world research: a resource for social scientists and practitioner-researchers, Blackwell Publishers, Oxford, UK; Madden, Mass., (2002).

[23] T.L. Saaty and L.G. Vargas, Models, methods, concepts & applications of the analytic hierarchy process, Kluwer Academic Publishers, Boston, (2001).

[24] M. Shepperd and M. Cartwright, Predicting with sparse data. *IEEE Transactions on Software Engineering*, 27 (2001), 987-998.

[25] G.J. Steven, The learning curve: From aircraft to spacecraft? *Management Accounting*, 77 (1999), 64-65.

[26] C.R. Symons, Software sizing and estimating: Mk II FPA (function point analysis), Wiley, New York, (1991).

[27] P. Tomaszewski and L. Lundberg, Evaluating Productivity in Software Development for Telecommunication Applications, *The IASTED International Conference on Software Engineering*, IASTED, Innsbruck, Austria, (2004), 189-195.

[28] M. van Genuchten, Why Is Software Late? An Empirical Study of Reasons for Delay in Software Development. *IEEE Transactions on Software Engineering*, 17 (1991), 582-591.

[29] W.K. Vaneman and K. Trianfis, Planning for technology implementation: an SD(DEA) approach, *PICMET '01. Portland International Conference on Management of Engineering and Technology.*, PICMET - Portland State Univ, Portland, OR, USA, (2001), 375-383.

[30] W.D. Yu, D.P. Smith, and S.T. Huang, Software productivity measurements. *Proceedings of the 15th Annual International Computer Software and Applications Conference COMPSAC '91.*, (1991), 558-564.

Paper II

The Increase of Productivity over Time – an Industrial Case Study

Piotr Tomaszewski, Lars Lundberg

Information and Software Technology (2006), vol 48/9, pp. 915-927

Abstract

Introducing new and specialized technology is often seen as a way of meeting increasing non-functional requirements. An example of such a technology is a software platform that provides high performance and availability. The novelty of such a platform and lack of related experience and competence among the staff may affect initial development productivity. The competence problems should disappear with time. In this paper we present a study, which we conducted at Ericsson. The purpose of the study was to assess the impact of experience and maturity on productivity in software development on the specialized platform. We quantify the impact by comparing productivity of two projects. One represents an initial development stage while the other represents a subsequent and thus more matured development stage. Both projects resulted in large commercial products. We reveal a factor of four difference in productivity. The difference was caused by a higher code delivery rate and a lower number of code lines per functionality in the latter project. We assess the impact of both these issues on productivity and explain their nature. Based on our findings, we suggest a number of improvement suggestions and guidelines for the process of introducing a new technology.

1. Introduction

The increasing expectations concerning software systems require solutions that make it possible to deliver more and more sophisticated products. A well established way of overcoming technical limitations is to introduce new technology. An area in which a rapid increase of requirements can be observed is telecommunication. The increasing number of services and subscribers puts high quality requirements on the systems that provide the infrastructure for the mobile telephony network. A requirement that recently has become very important is high availability. The service should be available all the time – service outages result in significant financial losses of the network operator. One way to facilitate development of highly available systems is to introduce a specialized software platform. Such a platform can help achieving the desired availability level by using sophisticated built-in mechanisms for efficient data replication, failure recovery, and online updates.

Introducing a specialized platform, as any technology adoption process, is always connected with some cost. In the beginning, the new platform is unknown to the developers. Current development processes may not be well suited to the new way of developing software. Specialized platforms may also not offer as wide range of tools that support software development as standard platforms do. These issues may result in low productivity for initial software development on such a platform. In Paper I we have described such a situation. We have identified a factor of four difference between productivity on a standard (Unix) and specialized platform. Our study was performed just after the new technology was introduced. We identified a staff competence level, as well as organization and platform immaturity to be the main reasons for low productivity. In this paper, we present a follow-up study to the work presented in Paper I. In this study we analyze and quantify the change over time of the software development productivity on the specialized platform. The main methods used are historical data research and interviews with experts.

Our case study is based on two large industrial projects at Ericsson, in which the new specialized platform was used. From now on we call them Project A and Project B. To observe the impact of time on the development productivity we selected the projects such that there is a difference of approximately three years between the start of Project A and Project B. Project A was the first project, in which the new platform was introduced. Project B is a project that has recently been finished. Therefore, it represents the most recent development stage. Both projects are large (over 100 KSLOC, 20-50 person years) and both resulted in commercially available systems. The systems examined in our study co-operate within the same solution that operates in the service layer of the mobile telephony network. In Project A the entire development was done in C++; in Project B 75% of the code was written in Java, and 25% in C++. Analysis and design of both systems was performed using similar methods, which were typical methods for the organization, in which the system was developed.

The goal of this study is to investigate which initial problems experienced by the developers have been overcome and which remain. Such an analysis can not only help in better productivity predictions when a new technology is introduced, but also makes it possible to suggest improvements to the technology adoption process. When planning the study we formulated the following research questions:

- *What is the difference in productivity between the initial and the subsequent development?* To answer this question we compare the productivity on Project A and Project B.
- *What caused the difference?* We identify the factors that affected productivity in Project B and compare them with the factors that influenced productivity in Project A. We analyse which productivity bottlenecks disappeared in Project B and identify the underlying reasons. We also investigate what issues are currently hindering productivity of software development.
- *How can the productivity improvement be accelerated?* By analysing the problems that have disappeared with time we suggest improvements to the process of introducing a new technology.

2. Related work

The change of productivity, or more generally, the change of cost with time, is a well recognized and described economical phenomenon. In the literature it is presented under numerous terms, e.g.:

- *"economies of scale",* which states that the more units that are produced the lower is the cost of a single unit. This can be partially attributed to better experience and learning process of the producer
- *"learning curve"* – described by Wright [38]. It states that the unitary cost of production decreases with time

- *"experience curve"* – described by Henderson [9]. According to the experience curve theory the production cost is decreasing when experience is gained

A lot of work has been done on building models for predicting the productivity or cost changes with time. A good overview of the work related to the learning curve can be found in [2, 37, 40]. In this study we are, however, more interested in understanding why the effect presumably takes place and not in building a prediction model. In [2, 15] a list of possible sources of the learning/experience effect can be found. These are [2]: tools, methods, product design changes, management, volume change, quality, incentive pay and operator learning. Even though this list was not defined for software development it seems obvious that the issues presented there also affect software development productivity.

Many researchers have observed, documented and tried to solve the productivity problem when adopting a new technology [16, 18, 23, 25, 28, 36]. Most researchers agree that some of the initial productivity problems fade away with time due to growing experience and maturity of the organization. The question of how to make that time as short as possible remains unanswered. In [16] Edmondson *et al.* stress the importance of having the knowledge about the new technology codified. The more the knowledge about the new technology is tacit, the bigger the problem is to introduce it seamlessly. Fisher and Wesolkowski [18] present another view of the initial knowledge problem. It might be extremely difficult to increase the competence level of the staff if the staff does not want it. Harvey *et al.* [23] analysed 100 companies to find out how the companies that successfully adopted new technologies differed from the others. One of their findings was the important role of management. The importance of management's role in facilitating the process of new technology adoption is also stressed by Vaneman and Triantis [36].

The topic of productivity has also been discussed specifically in the context of software development. Productivity issues are often referred to when discussing delays in software deliveries [6, 7, 35]. Blackburn and Scudder [6] identified a number of factors that reduce the development time. Improvement of productivity is a way to decrease the development time. The authors examined data from 40 different projects. Reuse of code was the most promising technique for development-time reduction. The authors also discuss prototyping, the quality of requirement specification, the use of modern tools and management role as factors that directly impact development productivity.

3. Platform presentation

The specialized platform, which we evaluate in this study is used in real-time, high availability telecommunication applications. Developing such applications requires meeting a number of contradicting requirements. Achieving high availability is always connected with introducing redundancy into a system. Each vital piece of hardware or data must be replicated to assure a continuous operation of the system in case of failure. To ensure consistency between different data replicas an extensive updating communication is often required. This communication, however, usually affects the performance of the entire system negatively. This is a problem, because real-time systems used in telecommunications often face strict performance requirements. Therefore, a lot of work has been put into constructing platforms that make it easier to develop efficient high availability telecommunication systems.

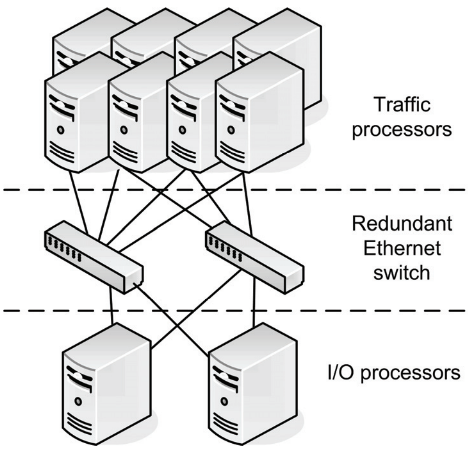
Typically, such platforms are either hardware or software based. Hardware based solutions, e.g., [4, 27] usually involve proprietary hardware (e.g., triplicated processors [27]) which makes them very expensive. Software based solutions overcome this problem by operating on standard or existing hardware. The platform we discuss in this study is an example of a software based solution.

A hardware configuration of the platform comprises (See figure 1)

- Up to 40 traffic processors that process pay-load
- Two I/O processors responsible for the external communication and maintenance
- Two Ethernet switches and two separate interconnections via Ethernet networks

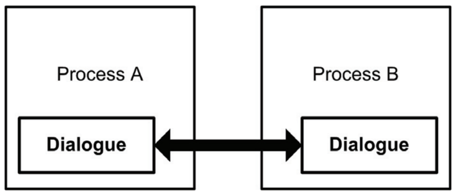
The platform offers standard interfaces (APIs) for Java and C++ but the programming model is unique. The main execution unit is a process. There are two types of processes, static ones that are always running and dynamic ones that are created and destroyed on demand. The platform provides an in-memory database. The basic units of storage are database objects. The inter-process communication is done by dialogue objects or globally accessible database objects. Dialogue objects are used for message passing communication (Figure 2). In the communicating processes two corresponding Dialogue type objects have to be created. They exchange messages using a built-in mechanism provided by the platform.

Figure 1. Specialized platform



Database objects can be accessed by any process running on the platform. Therefore, they can be used to implement a shared-memory communication model (see Figure 3).

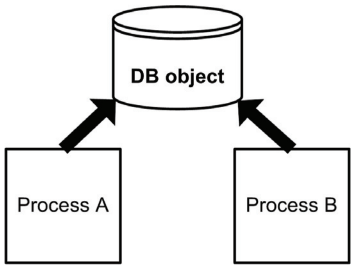
Figure 2. Inter-process communication using dialogue objects



To assure an efficient load balance, the programmer has a set of methods for allocating database objects and processes to processor pools, i.e. sets of traffic processors on which database objects and processes may operate. The load balancing within a pool is done by the platform itself.

To facilitate the programming of the highly available systems, every process or database object is automatically replicated on two different machines in the cluster – a failure of one of them does not affect the correct operation of the entire system. The platform also has built-in features that allow for online upgrades of the applications.

Figure 3. Inter-process communication using database objects.

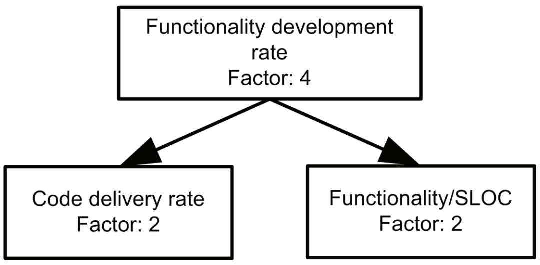


4. Productivity in the early software development

In the previous study (see Paper I), we have identified and analyzed the issues that affect the initial productivity of software development on the specialized platform. We have found that delivering equivalent functionality on the specialized platform takes four times longer compared to a standard Unix platform, i.e., the functionality development rate was four times as high on Unix as it was on the new platform. We have decomposed this factor of four into two factors (see Figure 4):

- the code was written twice as slow
- on average there was twice as much code per functionality.

Figure 4. Decomposition of productivity problem in early software development on the new platform



To explain the difference in code delivery rate we performed interviews with 20 designers. The interviews resulted in a list of productivity bottlenecks that affect code delivery rate. The importance of each individual issue was later quantified by the interviewees using the AHP [33] method. The final list is presented in Table 1.

Table 1. Issues affecting coding speed in early software development on the new platform – prioritization

Bottleneck	Importance
Staff competence level	22%
Unstable requirements	16%
Not enough experience sharing and training activities	13%
Quality of platform's documentation	10%
Runtime quality of the platform	10%
Too much control in development process	9%
Too optimistic planning, too big scope of projects	8%
Lack of target platform in the design phase	7%
Platform interface stability (API)	5%

Surprisingly for us, the platform quality issues were not ranked as the most important. The respondents generally rated issues connected with the individual competence level as affecting productivity the most. The project also suffered from the problem of unstable requirements, which, however, happened partially due to low productivity. The low productivity resulted in longer lead-time. Projects with long lead-time are usually more prone to change of market demands.

The second issue affecting low functionality development rate, the high amount of code per functionality, was explained by analyzing the project structure. We found that the issue affecting it most was the lack of libraries or components for typical purposes, like communication protocols. Such libraries are available for standard platforms but were not available for the specialized one.

5. Method

In this section we present the methods used in the study. Each subsection of this section has a corresponding subsection in Section 0, where the results are presented.

5.1 Productivity measurement

Traditionally, the productivity is defined as a *ratio of output units produced per unit of input effort* [1]. The input effort is usually defined as the sum of all resources that were used to produce the output. In software development, the biggest part of the production cost is the cost of work. Therefore, we selected person hours as the input effort metric.

Two perspectives of measuring the output product by system size can be identified:

- *Internal viewpoint* (developer's perspective) – describes the amount of code that must be produced to complete the system.
- *External viewpoint* (customer's perspective) – refers to the amount of functionality provided by the system.

The internal point of view metrics measure the actual length of the code. A typical unit of the internal size is number of source lines of code (SLOC) - e.g., [6, 7, 30, 39]. Therefore, the measurement of productivity from an internal point of view describes the code delivery rate.

An often mentioned weakness of measuring system size using SLOC is that the result depends on coding style - one programmer can write a statement in one line while another can spread it across a number of lines. To check this, the ratio of code lines per statement was calculated for both projects. This metric should, to a certain extent, assure that the coding style was similar in both projects.

The external viewpoint metrics measure the functionality. The measurement of productivity from an external perspective describes the rate of functionality development. In the study, instead of measuring, we decided to estimate the ratio of functional sizes of both systems. Since expert judgement is considered as an acceptable way of performing estimations [8, 26, 34], we used it for comparing the functionality of systems. Due to the fact that both systems under study co-operate within the same larger system and that they provide complementary functionality of similar complexity, it turned out to be very easy for our experts to reach consensus when it comes to comparison of amount of functionality provided by both systems.

The following data concerning the size of projects were collected:

- Number of person hours spent on each project (Hour) – only the development phase of the project was taken into account (design, implementation, testing). Person hours include designers, testers and managers work hours.
- Number of code lines in each project (SLOC) - we counted only lines with code, comments and blank lines were not counted.
- *SLOC/ statement* ratios in both projects
- The amount of functionality in Project B divided by the amount of functionality in Project A (FUNC) – an expert estimation. The estimation was made by six experts during a consensus meeting.

5.2 *Quality aspects*

In [12], Chambers noticed that a new technology is introduced to either improve product quality or manufacturing efficiency (productivity). Therefore, the productivity can only be "*interpreted in context of overall quality of the product*" [1]. Two products may deliver the same functionality but capabilities other than functional (e.g., performance, reliability, availability, modifiability, usability, etc.) can make one product better from some perspective. Such characteristics do not come for free and therefore differences in any aspect of quality may possibly explain the difference in productivity. Lower development productivity can be the price for higher product quality. Therefore, in this study, quality aspects were kept in mind when evaluating productivity.

We considered the following factors that may impact productivity:

- *design quality* – the quality of application design and the quality of code produced. Different aspects of design may have impact on the fault-proneness of the code [3, 10, 11], or the modifiability [22] of the system, or even the productivity [13].
- *final product quality* – the quality actually achieved in the final application. An example of such qualities may be non-functional requirements. Strict non-functional requirements (e.g., security, high availability, performance) can be the important cost driver and can therefore account for a high development cost.
- *quality of development process* – different quality assurance activities often account for a large part of the project budget (e.g., [11]). They are usually seen as an effective but expensive way of improving quality (e.g., [5, 32]). Therefore, we consider it important to capture the differences between processes in both projects; as such differences may explain the differences in productivity.

Our assessment of the design quality was based on metrics, for which there are indications (e.g., [3, 10, 13, 17]) that they may have impact on quality:

- *McCabe Cyclomatic Complexity (MCC) [31]*. This metric measures the number of linearly independents paths through the function. According to [19] "*Overly complex modules are more prone to error, are harder to understand, are harder to test, and are harder to modify*."
- *Lack of Cohesion (LC)* [14]. It measures *"how closely the local methods are related to the local instance variables in the class"* [17]. The idea is to measure to what extent the class is a single abstraction. In [13] Chidamber *et al.* showed that high values of the Lack of Cohesion metric have negative impact on

productivity. In this study we counted LC using a method suggested by Graham [20, 24] which gives normalized values of LC (0%-100%). We considered normalized values more applicable for comparison purposes.

- *Coupling (Coup)* [14], the metric measures the number of classes the class is coupled to. This metric makes it possible to assess the independence of the class. High levels of coupling are recognized as a factor negatively influencing productivity [13] and fault proneness [3, 10].
- *Depth of Inheritance Tree (DIT)* [14], measures how deep in the inheritance hierarchy the class is. According to [14] high DIT values result in higher complexity and make prediction of class behaviour more difficult. In [22] Harrison *at al.* found that systems with inheritance are more difficult to modify. In [3] Basili *at al.* showed that there is a relation between DIT and the fault proneness of the class. The same conclusion was reached by Cartwright and Shepperd in [11].
- *Number of Children* (NC) [14] is defined as "*the number of immediate subclasses subordinated to a class in the class hierarchy*" [14]. In [14], the authors claim that classes with a huge number of subclasses have potentially bigger impact on the whole design and therefore they require more testing (since their errors are propagated). In [11], Cartwright and Shepperd showed that there is a relation between inheritance and fault proneness – classes that were involved in inheritance had higher fault densities.

Other quality aspects, like the quality of the final product or the quality of the development process are difficult to quantify. Therefore, the opinions about both of them were collected during the workshop in which developers involved in both projects took part. The quality of the final product was assessed mainly by comparing the non-functional requirements put on the systems. The development process quality discussions focused on the amount of quality assurance activities, like testing, inspections, or the level of detail in project documentation.

5.3 Explaining productivity differences

To identify and explain the issues that could affect the change of productivity we organized a workshop to which we invited six experts from the organization where both systems were developed. Before the discussion we created a list of possible factors that could affect productivity. The list was created based on literature review ([2, 6, 7, 15] – see the "Related work" section) and our experiences from the previous study (see Paper I). We have identified the following factors:

- *Competence level* – this factor corresponds directly to the “operator learning” factor from [2]. The increase of competence due to experience or additional training could affect the productivity change. It can also be so that only the more experienced developers were involved in one of the projects.
- *Methods and tools* – these factors are taken directly from [2]. The introduction of new technologies, development methods, or tools can affect productivity.
- *Volume change* – a factor that also comes from [2]. The project complexity grows with size but, on the other hand, in big projects we can experience an “effect-of-scale” (see Section 0). An inappropriate scope of the project and size of the staff can affect the productivity negatively.
- *Quality* – a factor mentioned in [2]. As we have discussed in the “Quality aspects” section (see Section 5.2) different quality aspects can impact productivity. Heavy quality assurance or strong non-functional requirements are examples of quality issues affecting productivity.
- *Design* – a factor from [2]. Some fundamental differences in the system architecture can impact productivity.
- *Management* – the impact of management on productivity is a widely recognized issue [2, 36]
- *Chain effects* – as chain effects we understand issues that, although external to an organization, have an impact on its productivity. The project can benefit from improvements outside the project, e.g., the platform may improve or the input to the project (requirements) may be better defined and more stable.
- *Code reuse* – Reuse is considered an important factor influencing productivity [6].

Based on the factors presented above, the experts discussed the issues that affected productivity in Project B. By combining the new findings with the list of productivity bottlenecks from Project A (see Section 0) we created a list of issues that had impact on productivity in the software development on the specialized platform. We asked the experts to compare the impact on productivity of each issue in Project B with its impact on productivity in Project A. In this way we analysed the importance of the productivity bottlenecks in initial and subsequent software development.

Apart from explaining the productivity difference we were also interested in the experts’ opinions concerning the methods of overcoming both the initial and current productivity problems.

6. Results

Due to the agreement with our industrial partner the measurement results are presented as [Project B / Project A] ratios. No results concerning size or effort have been presented as absolute values.

6.1 Productivity measurements

The results of the size and the effort measurements taken on both projects are summarized in Table 2. The table presents relative values only.

Table 2. Project size and effort ratios

Metric	Project B/Project A
Code lines (SLOC)	0.34
Functional size (FUNC)	1
Person hours (Hour)	0.24

The measurements show that there is approximately 3 times as much code in Project A as in Project B. The experts estimated that both projects provide approximately the same amount of functionality. This means that in Project B the same amount of functionality was delivered using only 34% of the code. The development effort in Project A was about four times as high as in Project B.

To check if a similar coding style was used in both projects, the average number of code lines/language statement was calculated for both projects. It turned out to be similar. For Project A it was 2,42 lines/statement, while for Project B it was 2,46 lines/statement. Therefore, we considered code lines comparable between both projects.

From the information about size and effort the development productivity ratios were calculated. The results are presented in Table 3.

Table 3. Productivity ratios

Metric	Project B / Project A
SLOC/Hour	1.41
FUNC/Hour	4.16

The code delivery rate has increased by about 41% from Project A to Project B. In Project B, the functionality was delivered over four times as fast as in Project A.

6.2 Quality aspects

To compare the design quality in both systems a number of measurements (described in Section 0) were done. Each metric was analyzed separately. All the measurements were done either at the function or at the class level. For each metric we present mean, median, standard deviation and minimal and maximal values obtained. This way of describing measurements was used in [3]. We also present histograms describing the percentage of the entities (classes, functions) in the system that have a certain value of the metric. Since it is sometimes difficult to assess if the obtained values are typical or not, wherever possible we add a column of corresponding values from the study described in [3]. Other examples of such values can be found in [10, 13, 14, 29]. We selected values from [3] for comparison purposes because the measurements there were taken on a relatively large project, which is similar to our study.

The first metric applied was McCabe Complexity. The results were obtained on the function level and are presented in Table 4.

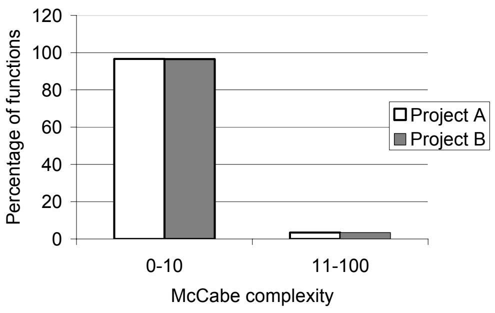
Table 4. McCabe complexity

	Project A	Project B
Mean	2.5	2.45
Median	1	1
Maximum	96	73
Minimum	1	1
Std. deviation	5.0	4.4

According to [31], the McCabe Complexity of the function should not be higher then 10, otherwise the function is difficult to test. We examined the data from that perspective (see Table 4).

In both projects the vast majority of the functions (97% in Project A and 97% in Project B) have complexity below 10 and therefore we consider both projects similar from that perspective. The remaining measurements gave results on the class level

Figure 5. McCabe Complexity - distribution



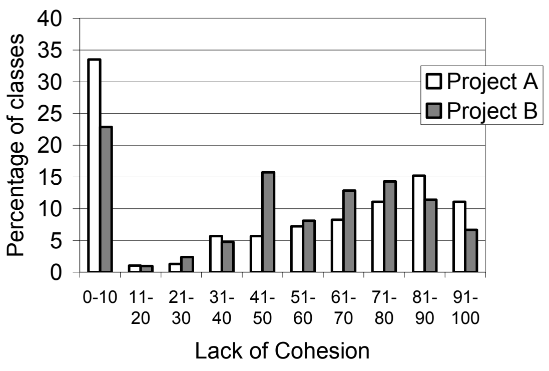
The second metric applied was Lack of Cohesion. The results are summarized in Table 5.

Table 5. Lack of Cohesion

	Project A	Project B
Mean	46.8	50.2
Median	57	56
Maximum	100	100
Minimum	0	0
Std. deviation	37.4	31.97

The distribution of LC values between classes of both systems is presented in Figure 6. We can observe that trends in distribution are similar in both systems. Mean and median values are also similar. Therefore we consider both systems similar from a Lack of Cohesion viewpoint.

Figure 6. Lack of Cohesion - distribution



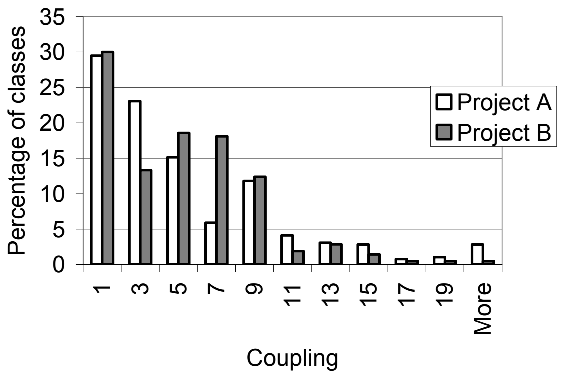
The next metric applied was Coupling. The results are summarized in Table 6.

Table 6. Coupling

	Project A	**Project B**	***Ref.[3]***
Mean	5.1	4.6	*6.80*
Median	3	5	*5*
Maximum	35	37	*30*
Minimum	0	0	*0*
Std. dev.	5.7	4.4	*7.56*

The distribution of Coupling values in the project classes is presented in Figure 7.

Figure 7. Coupling – distribution



We consider the distribution of coupling values similar for both projects.

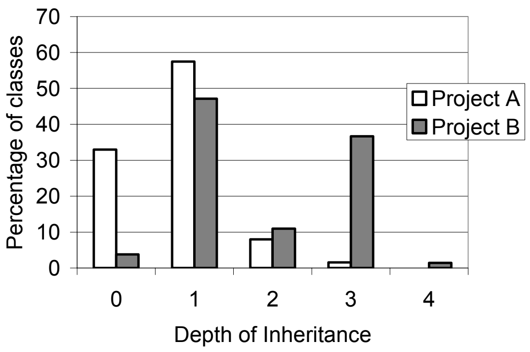
The application of the Depth of Inheritance Tree metric gave results presented in Table 7.

Table 7. Depth of Inheritance

	Project A	**Project B**	***Ref.[3]***
Mean	0,78	1.85	*1.32*
Median	1	1	*0*
Maximum	3	4	*9*
Minimum	0	0	*0*
Std. dev.	0.7	1	*1.99*

The distribution of DIT values is presented in Figure 8.

Figure 8. Depth of Inheritance - distribution



The mean values of DIT (see Table 7) indicate that in Project B there are more classes involved in inheritance structures than in Project A. From Figure 8 we can see that the percentages of classes with depth of inheritance equal 1 or 2 are similar in both projects. The difference in mean value is caused by about 30% of the classes that in Project B have inheritance depth equal 3, while in Project A their depth of inheritance is equal to 0. According to some studies ([3, 11, 22]) inheritance increases fault-proneness of the classes and makes the classes more difficult to modify (see Section 5.2 for details concerning these studies). Therefore, we further investigated the issue of higher average inheritance in Project B. According to our experts, the higher mean value of DIT in Project B can be explained by the use of Java for a large part of implementation in Project B. The experts say that it is more typical in Java than in C++ to inherit from some predefined classes. We investigated this and we found that in Project B all classes with DIT higher than 1 are indeed Java classes.

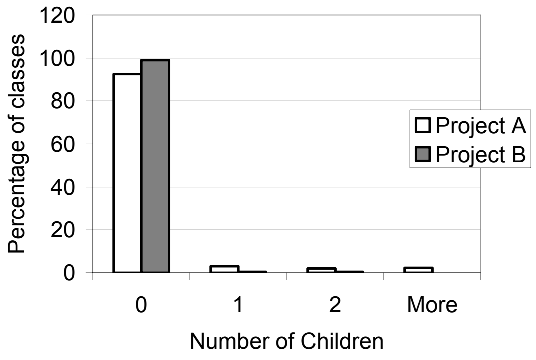
The last metric applied was Number of Children (NoC). The results are summarized in Table 8.

Table 8. Number of Children

	Project A	**Project B**	***Ref.[3]***
Mean	0.19	0.014	*0.23*
Median	0	0	*0*
Maximum	9	2	*13*
Minimum	0	0	*0*
Std. dev.	0.9	0.15	*1.54*

The distribution of NoC values is presented in Figure 9.

Figure 9. Number of Children – distribution



As can be seen in Figure 9 over 90% of classes in both projects have NoC equal to 0. Since the distribution is almost identical we consider both projects similar from NoC perspective.

After analyzing all the measurements we can not observe any major differences in design quality metrics between the two projects. Therefore, we consider the design quality similar in both projects.

The remaining two quality aspects, the product and the process quality, were assessed during the workshop with experts. According to them, the non-functional requirements put on both projects were rather similar. There was a difference, however, when it comes to the processes, especially the quality assurance processes. The experts shared the opinion that in Project B the processes were not as heavy as in Project A. The heavy processes in Project A were seen as a way to assure quality in the initial development on the new platform. In Project B, due to more experience with the platform, the amount of time spent on, e.g., inspections could be reduced to a more appropriate level. The experts agreed that lighter quality assurance processes contributed positively to productivity without affecting quality negatively. It can be considered an argument for a higher competence level of the staff in Project B – even though the quality assurance was not as extensive as in Project A, due to higher competence the quality achieved was similar.

6.3 *Explaining productivity differences*

To explain the difference in productivity we organized a workshop with experts. The experts discussed the issues that affected (both positively and negatively) the productivity in Project B. The discussion was based on the list of factors that influence productivity in software projects (see Section 0 for details regarding these factors). The findings are presented below:

- *Competence* – the competence level was perceived as a factor that contributed positively to productivity in Project B. According to experts, the experience and knowledge of the platform was significantly higher in Project B compared to Project A.
- *Design* – the design of both projects was similar.
- *Tools and methods* – one difference between both projects was that in Project B a large part of the code (approximately 75%) was developed in Java. According to the experts this should affect productivity positively. The reasons were:
    - more libraries and components were available for Java than for C++ in their application domain
    - Java designers used a modern Integrated Development Environment (IDE) that offered better tool support compared to the C++ environment used
- *Volume* – the scope of Project B was more appropriate and manageable but could have been better defined in the beginning
- *Quality* – the quality was discussed thoroughly in Section 0. The main difference between the projects was a more appropriate level of quality assurance activities in Project B, which should contribute positively to productivity.
- *Value chain effects* – here the experts focused mostly on two issues: the stability of the requirements and the platform quality. According to them the requirements were equally unstable in both projects, which affected productivity negatively. As far as platform quality was concerned, the experts agreed that it had improved significantly from the time when Project B was developed. However, they missed tools like debuggers and profilers. They would also appreciate more advanced database support.
- *Management* – the experts agree that the management work was more flexible in Project B, which contributed positively to productivity.
- *Code reuse* – in Project B there were components that were taken directly from one of the previous projects. We have

investigated that issue and we have measured the amount of code in the reused components. We found out that for each line of code written in Project B there was one line reused. The reused lines were not calculated in our measurements presented in Table 2 and therefore did not affect our code delivery rate measurement in Project B. They contributed only to the high Functionality/SLOC in Project B.

Further during the workshop, we presented the experts with the ranking of the issues that were identified as productivity bottlenecks in Project A. The identification and prioritization of issues that affected productivity in Project A was presented by us in Paper I (for a summary of findings from this study, including the list of identified productivity bottlenecks, see Section 0). We asked the experts to compare the impact of those issues on productivity in the initial and in the subsequent software development. We also asked them to assess the impact of the productivity bottlenecks that were identified in Project B. The results of the assessment are presented in Table 9.

Table 9. Impact of the bottlenecks on productivity in initial and subsequent development – a comparison

Productivity bottleneck	Impact in initial development	Impact in subsequent development
Staff competence level	large	not a bottleneck
Unstable requirements	large	large
Not enough experience sharing and training activities	large	not a bottleneck
Quality of platform's documentation	average	average
Runtime quality of the platform	average	not a bottleneck
Too much control in development process	average	not a bottleneck
Too optimistic planning, too big scope of projects	average	average
Lack of target platform in the design phase	small	not a bottleneck
Platform interface stability (API)	small	not a bottleneck
Lack of programming tools for the specialized platform	not a bottleneck	large

By comparing the impact in the initial and in the subsequent development we can see that from the issues that had high negative impact on productivity in the initial development only “Unstable requirements” is ranked equally high. The issues connected with the competence level and the platform shortcomings that were ranked high as bottlenecks in Project A received much less attention in Project B. However, there is a new issue, “Lack of tools”, which was not recognized as a problem in initial development but has a significant impact on the productivity in the subsequent one. The experts shared the opinion that the productivity would be higher if they had more advanced programming tools, like Integrated Development Environment (IDE) for C++, better debuggers, profilers, and more advanced database support.

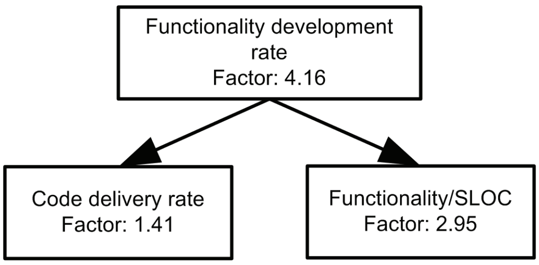
7. Discussion

7.1 Difference in productivity

By measuring productivity we have established the current level of the software development productivity on the new platform. We have identified a factor of four difference in the productivity measured from an external perspective between the initial and the subsequent software development on the new platform. This means that currently the organization is four times as efficient, when it comes to delivering functionality. We have decomposed this as a result of two factors (see Figure 10):

- *Code delivery rate*. In Project B there was approximately 41% more code delivered per unit of time.
- *Functionality/SLOC*. In Project A the average number of code lines per functionality was almost three times as high as in Project B.

Figure 10. Productivity difference – decomposition



The reasons for higher code delivery rate were investigated during the workshop. According to the experts, the increase in code delivery rate was an effect of (see Section 0):

- *Higher competence level* – better knowledge of the platform, and much more experience of the staff
- *Better tools* – the development environment (IDE for Java) used in a large part of the project, the improved stability of the platform
- *Better work organization* - lighter processes (partially due to higher competence), better and more flexible management

However, the largest part of the productivity improvement can be attributed to the high Functionality/SLOC value in Project B (almost a factor of three difference – see Figure 10). According to the experts, there were two major reasons for having so high Functionality/SLOC value in Project B:

- *Code reuse.* For each line written there was one line reused in Project B.
- *The use of Java as programming language*. Compared to C++, Java offered better support for the developers in terms of available libraries for typical functionalities (e.g., for graphical user interface implementation).

According to the experts, the use of Java was one of the factors that affected the high Functionality/SLOC value. We have quantified the impact of Java by performing an exercise in which we excluded the impact of code reuse on Functionality/SLOC value. We assumed that the remaining gain is attributed to the use of Java.

We know that in Project A there were 2.95 times as many lines to provide the same functionality as in Project B (see Figure 10). We also know that for each line written there was one line reused in Project B. The reused lines were not counted when measuring the project size. If we include the reused code in Project B, it would double its size and thus would decrease the difference in Functionality/SLOC between both projects by half – from 2.95 to 1.475. The experts think that this 1.475 overall increase was gained due to the use of Java. Java code accounted for only 75% of the code in Project B. The productivity gain, with respect to different programming languages, can be calculated by solving following equation (1):

$$0.25 \cdot Prod_{C++} + 0.75 \cdot Prod_{Java} = 1.475 \qquad (1)$$

We assume the same C++ productivity (in terms of Functionality/SLOC) in both projects, which means that $\text{Prod}_{C++} = 1$ (as it was in Project A). Therefore, $Prod_{Java} = 1.63$. This means that on average Java was able to deliver approximately 63% more functionality from a line of code. This is of course a very rough estimation, but it seems to support the opinions of our experts.

7.2 *Productivity improvement*

In the study, we investigated which of the initial productivity bottlenecks disappeared with time, as well as what issues are hindering productivity in current software development on the specialized platform. By analysing which of the issues tend to disappear with time we can suggest improvements to the process of introducing a platform that would make the learning time as short as possible. Identification of current productivity bottlenecks can help improving the productivity of mature software development, e.g., by suggesting certain platform improvements.

To find out the differences between initial and subsequent development we look at the bottlenecks identified in both projects through the general classification of productivity bottlenecks presented in [21]. In [21], Hantos and Gisbert divide productivity bottleneck origins into:

- *People* – issues connected with the competence level of people involved in the development process
- *Processes* – issues connected with the work organization characteristics
- *Technology* – issues connected with the technology used

In the initial development on the specialized platform the competence issues were ranked as the ones most affecting productivity. The platform issues that were mentioned mostly concerned the platform's quality and the API stability issues. The functionality of the platform and the tool support it offers were not questioned at that time.

Currently, it seems as if the focus has changed. The designers are much more confident since they managed to overcome the initial learning threshold. They can see certain shortcomings of the platform and they start to express their expectations concerning the platform. According to them, to improve their current productivity they need better, more modern tools.

Ironically, even though the platform is now seen as the major productivity bottleneck, it seems that it has improved significantly. According to our experts most of the issues reported as platform-related productivity bottlenecks in Project A are fixed now. The API is stable, there are no complaints regarding the runtime quality. Apparently, the platform producer focused on solving the problems instead of developing new features, which sounds like a reasonable thing to do. Currently, however, the focus should be put on making the platform more usable by providing better and more modern tools, like a modern Integrated Development Environment (IDE) with a debugger and a profiler. Also the platform documentation should be improved.

We can also notice that different work organization- and management-related productivity bottlenecks were ranked as less important now than before. Apparently, the organization adapted work processes to the new platform. For example, the level of quality assurance activities was adjusted - now it is as effective as before but less time consuming. It was possible due to higher staff competence –extensive quality assurance was not as necessary as in the beginning.

As we see it now, the quality shortcomings of the platform could have contributed to long initial learning of the developers. Even though the platform quality problems seem to have been overcome now, the specific programming model and problems with documentation can still make the learning process time consuming. Therefore, in order to make it shorter, the competence development activities, which we suggested in Paper I seem to be valid :

- *Good introduction process* - The introduction process would familiarize the staff with the new technology and minimize the overhead connected with learning.
- *Continuous skills development processe*s – e.g., an advanced course on programming for the platform, seminars, meetings and technical discussions would give the developers a chance to share experiences and spread knowledge among team members. It is especially important now, because there are a number of experienced developers that can share their knowledge and expertise with the new team members.
- *Better management of company knowledge*
    - Set of patterns - set of easily applicable solutions to the common problems.
    - Better documentation of the problems encountered would help to avoid making the same mistakes in the future.

The organization, in which we conducted the study, has learned a lot about introducing new platforms from Project A. They have adopted guidelines that suggest a limited scope and a limited number of staff when first projects on such new platforms are conducted. When a core group of specialists gain experience, projects can become bigger and new development team members can be added. In such a way, it is possible to build competence with a minimal impact on productivity.

Even though it may seem that the productivity has reached the level of Unix productivity (see Section 0) it is not entirely true. In Project B the productivity was gained mostly because of code reuse. If, instead of reusing, the code had to be developed, the productivity would decrease by half. That would make the Unix platform still twice as productive when it comes to delivering functionality. One problem is that code delivery rate is still 30% lower compared to the Unix platform. Therefore, we believe it is very important to focus on code delivery rate related improvements now, like the adaptation of modern programming tools (e.g., IDE) for the platform. Another issue is lack of libraries for typical functionalities (e.g., communication protocols), which are available for standard platforms (like Unix).

7.3 Lessons learned

We believe that some general lessons can be learned from our study. It seems very likely that any organization that decides to change a technology from a standard and well-known one to a very specialized one will face similar problems to those that we have identified in our study. The impact of individual issues may differ because they largely depend on the characteristics of the organization and the products that are developed. Also the degree of productivity decrease when a new technology is introduced depends very much on the individual settings. However, it seems reasonable to assume that some initial productivity decrease is difficult to avoid.

As a major issue impacting initial productivity we found the competence level among the staff. This issue is probably the most important when the introduced technology is very specialized. Such technologies, as opposed to standard ones, are usually not well-known and therefore require some time to be explored and mastered. If, like in our case, the technology has some unique characteristics (e.g., programming model), the learning curve can be very steep. Therefore, it is very important to provide good training opportunities, as well as good sources of information regarding the new technology to the staff. Such investments are likely to pay back in faster productivity increase.

Another issue that should be considered is the scope of initial projects done using the new technology. Initial projects are prone to delays, mostly because of low competence of the staff. Moreover, to compensate for low experience, the number of quality assurance activities is often increased, which makes projects even longer. In case of telecommunication systems, long projects are very prone to change requests, caused by changing market demands. This may delay projects even more. Therefore, the scope of initial projects, in which the new technology is used, should be small.

The new technology related competence of the staff will increase with experience. This is the time when the staff is most likely to start discovering different shortcomings of the new technology. If the technology is very specialized it might share many of the problems we have identified in this study. As such technologies are usually addressed to a limited number of customers; they may miss many add-ons typically available for standard, widely used platforms. Such add-ons may include debuggers, profilers, CASE tools, and other tools that largely facilitate software development. Therefore, before a new technology is introduced, it is recommended to investigate the need for such tools and their availability. This issue may not appear as a problem in the initial development but may become a serious productivity bottleneck when developers become more experienced.

8. Conclusions

The purpose of this study was to assess the impact of experience and maturity on the productivity in software development on a specialized, high availability platform. To achieve that, we have quantified the productivity and identified the productivity bottlenecks in initial and subsequent development on the platform. By analyzing the differences between them we have quantified the productivity change and described its sources. Finally, we have suggested improvement methods both for the process of introducing a platform and for the mature software development on the specialized platform.

The measurement of productivity from a functional perspective revealed a factor of four difference between initial and subsequent software development. We explain this by two factors. Firstly, in the subsequent development the code was delivered 41% faster. Secondly, in the initial project there was on average almost three times as much code per functionality as in the subsequent one.

The higher code delivery rate was achieved mostly because of a higher competence level, better tools and better work organization in the later project. The smaller amount of code necessary to deliver a functionality was a result of code reuse and the use of Java as a programming language. The code reuse increased the productivity in the project representing subsequent development by the factor of two. Java provided about 63% more functionality per code line compared to C++.

For both projects we investigated how the productivity was affected by different quality aspects. As far as design quality or non-functional requirements are concerned, we did not detect any major differences between the projects. The only difference concerned the quality of the software development process. In the initial development the quality assurance processes were rather heavy. Because of higher competence of the staff in the subsequent development they could be lighter and still equally effective.

The findings show that in initial development, the competence is the biggest problem. Therefore, we have suggested a number of competence development activities that can be applied when a new technology is introduced. These activities should accelerate the learning process and allow significant savings due to increased productivity. They are especially important in case of very specialized technologies with a steep learning curve, like the platform presented in this study.

When certain experience is gained, issues other than staff competence come into play. In the case presented, we found that additional improvements of the platform are necessary to further improve productivity of software development. The improvement suggestions mostly concern programming tools that should be available to the developers.

The lack of convenient tools can be seen as a consequence of introducing a very specialized platform. Such platforms, due to the limited number of potential users, are likely to lack tools that are available for standard, widely used platforms. Considering that among those there are debuggers, profilers or CASE tools the impact of their absence can be substantial.

The large role (a factor of two) of code reuse in productivity improvement proves that it is a very efficient way of decreasing a project’s cost. Therefore, developing code with its possible reuse in mind is a good investment that can bring significant savings in future projects.

As a concluding remark, we would like to point out certain limitations connected with productivity evaluations. A precise software development productivity assessment is a difficult, if not impossible task. As the main reason for this we see the lack of common understanding of what the actual product of the development is. It is difficult to quantify and compare the functionality of two software systems, but it is even more difficult to compare their quality. The number of different aspects of software quality mentioned in literature is large and still increasing. Not all quality aspects have assigned metrics, which enable their quantification. Many of the quality aspects are very subjective. Therefore, any case study, in which productivity is evaluated, must to a certain extent be based on qualitative descriptions and approximations. However, despite all these limitations, we can see a clear benefit from performing and reporting productivity evaluations. They increase our awareness of productivity as a factor that impacts the cost of project. They contribute to a general understanding of issues that influence software development productivity. They also let us learn from the experiences of others.

9. Acknowledgements

The authors would like to thank Ericsson staff whose active participation made this case study possible.

This work was partly funded by The Knowledge Foundation in Sweden under a research grant for the project "Blekinge - Engineering Software Qualities (BESQ)" (http://www.bth.se/besq).

10. References

[1] *IEEE standard for software productivity metrics*, in *IEEE Std 1045-1992*. (1993). 2.

[2] P.S. Adler and K.B. Clark, Behind the Learning Curve: A Sketch of the Learning Process. *Management Science*, 37 (1991), 267-281.

[3] V.R. Basili and L.C. Briand, A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22 (1996), 751-762.

[4] K.H. Benton B., Yamada H., A fault-tolerant implementation of the CTRON basic operating system. *Proceedings of the 11th TRON Project International Symposium*, (1994), 65-74.

[5] S. Biffl and M. Halling, Investigating the defect detection effectiveness and cost benefit of nominal inspection teams. *Software Engineering, IEEE Transactions on*, 29 (2003), 385-397.

[6] J.D. Blackburn and G.D. Scudder, Time-based software development. *Integrated Manufacturing Systems*, 7 (1996), 60-66.

[7] J.D. Blackburn, G.D. Scudder, and L.N. van Wassenhove, Improving Speed and Productivity of Software Development: A Global Survey of

Software Developers. *IEEE Transactions on Software Engineering*, 22 (1996), 875-886.

[8] B.W. Boehm, Software engineering economics, Prentice-Hall, Englewood Cliffs, N.J., (1981).

[9] Boston Consulting Group, Perspectives on Experience, Boston, (1972)

[10] L.C. Briand, J. Wust, S.V. Ikonomovski, and L. H., Investigating quality factors in object-oriented designs: an industrial case study. *Proc. of the 1999 Int'l Conf. on Software Eng.*, (1999), 345-354.

[11] M. Cartwright and M. Shepperd, An empirical investigation of an object-oriented software system. *IEEE Transactions on Software Engineering*, 26 (2000), 786-796.

[12] C. Chambers, Technological advancement, learning, and the adoption of new technology. *European Journal of Operational Research*, 152 (2004), 226.

[13] S.R. Chidamber, D.P. Darcy, and C.F. Kemerer, Managerial use of metrics for object-oriented software: an exploratory analysis. *IEEE Transactions on Software Engineering*, 24 (1998), 629-639.

[14] S.R. Chidamber and C.F. Kemerer, A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20 (1994), 476-494.

[15] R. Conway and A. Schultz, The Manufacturing Progress Function. *Journal of Industrial Engineering*, 10 (1959), 39-53.

[16] A.C. Edmondson, A.B. Winslow, R.M.J. Bohmer, and G.P. Pisano, Learning How and Learning What: Effects of Tacit and Codified Knowledge on Performance Improvement Following Technology Adoption. *Decision Sciences*, 34 (2003), 197-223.

[17] N. Fenton and S.L. Pfleeger, Software metrics: a rigorous and practical approach, PWS, London; Boston, (1997).

[18] W. Fisher and S. Wesolkowski, How to determine who is impacted by the introduction of new technology into an organization. *Proceedings of the 1998 International Symposium on Technology and Society, 1998. ISTAS 98. Wiring the World: The Impact of Information Technology on Society.*, (1998), 116-122.

[19] S. Gascoyne, Productivity improvements in software testing with test automation. *Electronic Engineering*, 72 (2000), 65-67.

[20] I. Graham, Migrating to object technology, Addison-Wesley Pub. Co., Wokingham, England; Reading, Mass., (1995).

[21] P. Hantos and M. Gisbert, Identifying Software Productivity Improvement Approaches and Risks: Construction Industry Case Study. *IEEE Software*, 17 (2000), 48-56.

[22] R. Harrison, S. Counsell, and R. Nithi, Experimental assessment of the effect of inheritance on the maintainability of object-oriented systems. *Journal of Systems and Software*, 52 (2000), 173-179.

[23] J. Harvey, L.A. Lefebvre, and E. Lefebvre, Exploring the Relationship Between Productivity Problems and Technology Adoption in Small

Manufacturing Firms. *IEEE Transactions on Engineering Management*, 39 (1992), 352-359.

[24] B. Henderson-Sellers, L.L. Constantine, and I.M. Graham, Coupling and cohesion (towards a valid metrics suite for object-oriented analysis and design). *Object Oriented Systems*, 3 (1996), 143-158.

[25] M. Huggett and S. Ospina, Does productivity growth fall after the adoption of new technology? *Journal of Monetary Economics*, 48 (2001), 173-195.

[26] R.T. Hughes, Expert judgement as an estimating method. *Information and Software Technology*, 38 (1996), 67-76.

[27] D. Jewett, Integrity S2: a fault-tolerant Unix platform. *21st International Symposium on Fault-Tolerant Computing*, (1991), 512-519.

[28] T.W. Legatski, J. Cresson, and A. Davey, Post-downscaling productivity losses: When projected gains turn to unexpected losses. *Competitiveness Review*, 10 (2000), 80.

[29] X. Li, Z. Liu, B. Pan, and D. Xing, A measurement tool for object oriented software and measurement experiments with it, *10th International Workshop New Approaches in Software Measurement*, Springer-Verlag, Berlin, Germany, (2001), 44-54.

[30] K.D. Maxwell, L. Van Wassenhove, and S. Dutta, Software development productivity of European space, military, and industrial applications. *IEEE Transactions on Software Engineering*, 22 (1996), 706-718.

[31] T.J. McCabe, A software complexity measure. *IEEE Transactions on Software Engineering*, SE-2 (1976), 308-320.

[32] G.W. Russell, Experience with Inspection in Ultralarge-Scale Developments. *IEEE Software*, 8 (1991), 25-32.

[33] T.L. Saaty and L.G. Vargas, Models, methods, concepts & applications of the analytic hierarchy process, Kluwer Academic Publishers, Boston, (2001).

[34] M. Shepperd and M. Cartwright, Predicting with sparse data. *IEEE Transactions on Software Engineering*, 27 (2001), 987-998.

[35] M. van Genuchten, Why Is Software Late? An Empirical Study of Reasons for Delay in Software Development. *IEEE Transactions on Software Engineering*, 17 (1991), 582-591.

[36] W.K. Vaneman and K. Trianfis, Planning for technology implementation: an SD(DEA) approach, *Int. Conf. on Management of Eng. and Technology.*, Portland, USA, (2001), 375-383.

[37] F.K. Wang and W. Lee, Learning curve analysis in total productive maintenance. *Omega*, 29 (2001), 491.

[38] T.P. Wright, Factors affecting the costs of airplanes. *Journal of Aeronautical Sciences*, 3 (1936), 122-128.

[39] W.D. Yu, D.P. Smith, and S.T. Huang, Software productivity measurements. *Proc. of the 15th Annual Int. Computer Software and Applications Conference COMPSAC.*, (1991), 558-564.

[40] W.I. Zangwill and P.B. Kantor, The learning curve: a new perspective. *International Transactions in Operational Research*, 7 (2000), 595.

Paper III

Evaluating Real-Time Credit-Control Server Architectures Implemented on a Standard Platform

Piotr Tomaszewski, Lars Lundberg, Jim Håkansson, Daniel Häggander

Proceedings of IADIS International Conference on Applied Computing, vol. 2, pp. 345-352, Algarve, Portugal, February 2005

Abstract

The high non-functional requirements put on payment systems result in increasing complexity. One way of providing the expected quality is usage of a specialized platform. However, development on such a platform can be very expensive. In this paper we examine what level of quality can be expected when the development is done on a standard platform. We suggest a number of architectures and evaluate them from availability, reliability and performance perspectives. We present suggestions for the system developers concerning the choice of an appropriate architecture when a certain combination of qualities is required. Finally we identify problems we cannot solve using standard platforms.

1. Introduction

The growing importance of online services places much attention on payment systems. These systems are designed to charge customers and store their information. Therefore they are crucial for any service provider – they assure collection of revenue. This role results in very high expectations concerning certain quality aspects of such systems.

One such aspect is availability. A major advantage of e-business is that it is available 24/7 and that should be true for a payment provider as well. Any service outage results in financial losses. In addition, a payment system must be reliable. A reliable payment system should be able to process each ‘charging request’ and store the charging data safely. The loss of charging information would result in a loss of payment record, which is the equivalent to losing money. Another aspect is performance. Performance is especially important in case of prepaid services, where a customer pays for a service in advance. Example of such a service can be prepaid telephony or video-on-demand. Each customer has an account and a service can be provided only as long as the assets on the account allow it. That requires real time credit control.

The fact that the requirements mentioned above are not orthogonal makes the development of such systems a very challenging task. Therefore much effort has been put into methods, tools and technologies that facilitate payment systems developing. One solution is to use a fault tolerant platform. Such platforms are able to provide high availability while maintaining good performance and reliability. They would seem to be an ultimate solution to the problem. However, as the previous research shows, the development on such a platform can be expensive. In Paper I we have identified a factor of four difference in productivity on a specialized, fault-tolerant platform compared to a standard UNIX platform.

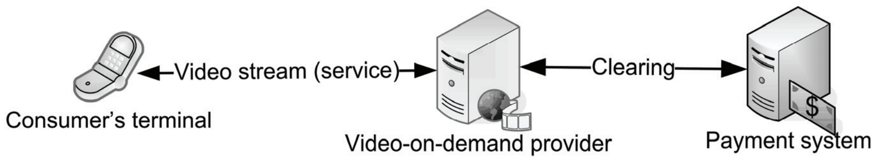
The high development cost may be a barrier that is impossible to cross. A competitive advantage gained, due to better quality characteristics of the system, may simply not be worth the money. Therefore our attention was again put on standard technologies. Even though they do not provide advanced, specialized features, their proven good productivity may be a great advantage.

The purpose of this study is to identify the service availability, performance and reliability characteristics, which we can expect when designing a payment system on a standard UNIX platform. The desired implementation should be compliant with one of the latest standards concerning real-time credit control, i.e. Diameter Credit Control [12]. In this study we suggest and evaluate a number of possible architectures of a payment system. The evaluation is done in an industrial setting. The testing was performed on a real payment system. The architectures are evaluated from the perspective of the three mentioned characteristics; availability, reliability and performance. The results of the study should help the developers of payment systems in determining the level of characteristics that they are able to provide using affordable technology and a method of how to do that.

2. Payment System Application

The role of the payment system will be presented in the example of a prepaid video-on-demand service. There are three parties involved in a service delivery (Figure 1), namely: payment provider – a system that we are going to analyze in this paper, video-on-demand provider and consumer that orders a video stream. According to Diameter Credit Control (DCC), the video-on-demand provider must implement Credit Control Client functionalities, while payment system provides Credit Control Server functionality. DCC specifies interaction between the Credit Control Client and the Credit Control Server.

Figure 1. Video-on-demand example



According to DCC after a consumer has requested a video stream, the video service provider contacts the payment system and requests permission to deliver the service to the customer for a specified period of time. The payment system calculates the cost of delivering the service for given time period and checks the account balance of the customer. If the balance is too low then it rejects the request, otherwise it reserves the money and accepts the request. The video service provider repeats the requests before each new time period.

In the study we consider a simple Credit-Control Client implementation that does not involve any resending of the requests. Therefore, if the video service provider does not receive an answer from the payment system within certain time limit, it aborts the service delivery (video transmission).

Rating (price calculation) operations performed by payment systems are often very complex. The calculation of service price depends on many parameters, such as; the type of service, the type of customer, the time of the day, the duration of service, historical information about customer, etc. Though it seems that the payment system mostly records data, our industrial experience shows that over 80% of all operations are read operations. This is caused by the need of reading all the parameters for price calculation. In the evaluation we assume equal distribution of the request throughout the operation time.

3. Method

In the study we identify a number of typical architectures that can be used to implement a Credit Control Server according to the DCC specification using standard UNIX platforms. The eight architectures are identified based on literature research and interviews with developers involved in a development of such systems. Each of the architecture proposals is evaluated. When evaluating availability we assess if the solution is able to survive a crash of one of its components without service stoppage. For the evaluation purposes we make a standard assumption that only one thing can break at the time. The availability definition is similar to the one suggested by [5, 11] and is described in terms of percentage of requests answered from requests that were sent by the client. In addition, we evaluate reliability of each of the architectures by estimating the percentage of successfully processed and stored requests. In evaluating performance of the system when processing its maximal load, we use two performance metrics; throughput and response time. The measurement of the response time under load is chosen because it is a real-time system that must provide required response time when being heavily loaded.

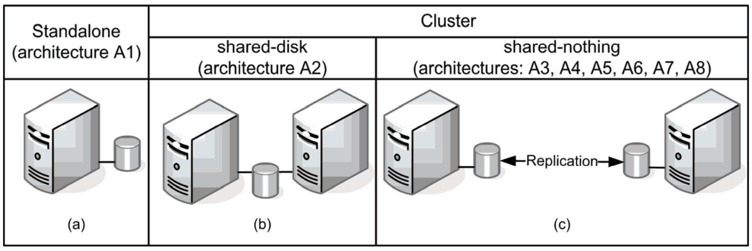
The evaluation is two-folded. We begin with a theoretical, qualitative evaluation of architecture characteristics. An estimation of the parameters allows us to eliminate four candidate architectures that are outperformed by other ones. By outperforming we mean that an architecture is better than another in one or more the aspects mentioned and not worse in the others. The estimation is based on data from the literature. For the remaining architectures, we perform a more thorough analysis that involves experimentation. The experimentation is carried out in an industrial setting on a real payment system and platform. Finally, we present the decisions and trade-offs that must be made by a payment system designer. We also provide suggestions as to which architecture to choose when a specific combination of qualities is required.

4. Architectures

In this section we present a number of possible payment server architectures. The simplest payment server architecture is a 'standalone' computer (Figure 2a) with a payment server application and a standard disk database. We will now refer to this architecture as A1. The architecture has an availability problem. When the database or the application or the whole computer fails the service is not available. This failure also results in loosing the requests that are currently processed, which is a reliability threat. Another reliability threat is a disk failure that may result in a loss of data from the database. However, normal practice would be to use a fail-safe technology, like RAID. Therefore we will not consider the disk a point of failure.

Both the reliability and the availability of A1 can be improved by introducing a dual-computer cluster and replication. We can either replicate a server only or a server and a database [3]. In case of the server replication both computers must have a shared disk (Figure 2b). Such an architecture will be referred to as A2. It improves availability of A1 by introducing a second contact point for the client. When the database is also replicated (Figure 2c) there must be a data replication mechanism between cluster nodes. Two possible data replication modes are available – asynchronous and synchronous.

Figure 2. Hardware configurations – overview


Standalone (architecture A1)
Cluster
shared-disk (architecture A2)
shared-nothing (architectures: A3, A4, A5, A6, A7, A8)
Replication
(a)
(b)
(c)

The asynchronous replication does not include replication time into a database update time. An update operation is committed on a local database only. The replication to the other machine is postponed - usually the change logs are bundled, which decreases network traffic. The shared-nothing architecture variant with an asynchronous replication will be referred to as A3. It provides better availability than A1. The reliability risk connected with a disk failure is minimised - data is stored in two places. However, the asynchronous replication has a problem of loosing the replication buffer - a number of requests that are waiting to be replicated. In case of a failure this data is lost and accounts on the other machine are not updated.

The “replication buffer loss” problem is minimised when a synchronous replication is used. The synchronous replication adds replication to a database update operation. An update transaction returns only if the change is committed in both databases. To increase availability it is usually possible to temporarily switch to asynchronous replication when one of the nodes fails. Otherwise the service would not be available during the failure. Synchronous replication improves the reliability of A3 but affects performance of update operations. Database reading is always done locally so replication does not affect its time. A variant of architecture with synchronous replication will be referred to as A4.

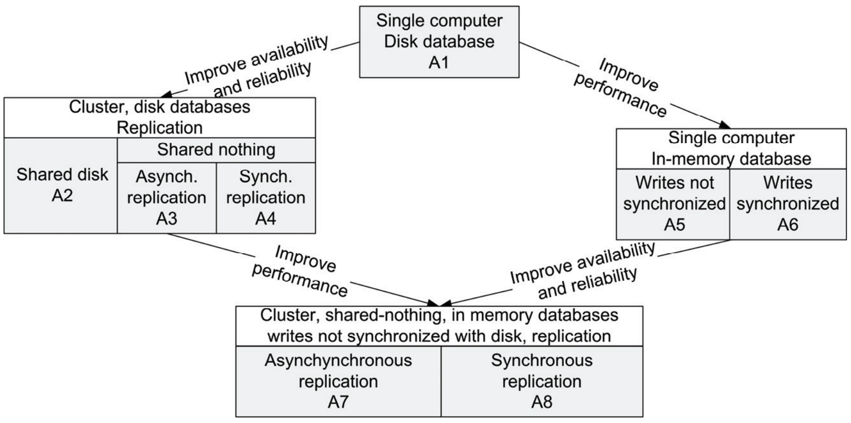
Architectures A2-A4 are aimed towards reliability and availability improvement of A1. In order to improve performance of A1 an in-memory database can be used. A large positive impact on performance of in memory databases is recognized by practitioners [6, 14, 18] and admitted by researchers [8]. The whole database is kept in memory which speeds up database operations by an order of magnitude [8]. Two main configurations of in-memory database are considered in this study. They are similar to two replication modes; however the replication is done between the memory and disk. In the first case all database writes are synchronized with a disk; in the second one they are deferred. From now on the solution without writes synchronized with disk will be referred to as A5, the one with writes synchronized is to be referred as A6.

Since in both cases the most often performed operation, database read, is done without involving a disk, both solutions offer much better performance compared to A1. The performance of A5 is better than the performance of A6. The reliability and availability of A6 is the same as in A1. The reliability of A5 is lower than of A1 due to that some of the data is, for certain amount of time, kept in memory only. We call that data portion a “disk write buffer”. That data portion would be lost in case of machine crash.

The last two variants we evaluate are combinations of A3 and A4 with A5. A7 consists of two computers that have in-memory databases and asynchronous replication between them. A8 has the same configuration with the exception of synchronous replication between machines. The characteristics they offer are exactly the same as their disk based counterparts apart from the significantly better performance.

The evolution of basic architecture can be followed in Figure 3. Because some of the architectures outperform others the outperformed ones will be excluded from this study. Architecture A1 is outperformed by A6, which offers better performance providing the same availability and reliability. The same situation is seen between architectures A2 and A4, which are outperformed performance wise by A8 and A3 which is outperformed by A7. For evaluation we have selected the following architectures: A5, A6, A7 and A8.

Figure 3. Architecture variants



5. Evaluation

5.1 Availability

The availability, described as percentage of answered requests, can be calculated using the expression (1):

$$\text{Avail} = \frac{Rq_{\text{year}} - Rq_{\text{noanswer}}}{Rq_{\text{year}}} \qquad \textbf{(1)}$$

Where Rq_{year} – number of requests per year, and $\text{Rq}_{\text{noanswer}}$ – number of requests unanswered per year.

Architectures A5 and A6 are realized using one computer only. If a computer or an application is out of service (due to failure or maintenance) a customer is not provided with a service. That means that the availability of A5 and A6 is limited to the time when everything works. Additionally there are always a number of requests that are processed concurrently in the system. These requests are also lost and unanswered in case of failure. Therefore for A5 and A6 the Rqnoanswer equals (2):

$$Rq_{\text{noanswer}} = N_{\text{Failure}} \cdot Rq_{\text{cur}} + (T_{\text{Repair}} + T_{\text{Maintain}}) \cdot Rq_{\text{sec}} \qquad \mathbf{(2)}$$

Where N_{Failure} is an average number of hardware and software failures per year, Rq_{cur} describes an average number of requests that are present in the system and are being processed at any point of time, Rq_{sec} equals an average number of requests per second, T_{Repair} is average time to repair software and hardware failures and T_{maintain} is an average maintenance time in a year. The availability is different in the case of A7 and A8. There are 2 machines involved and in case of a failure only the requests, which were sent to the node that crashed and not answered are lost. Therefore the Rq_{noanswer} of A7 and A8 equals (3):

$$Rq_{\text{noanswer}} = N_{\text{Failure}} \cdot Rq_{\text{cur}} \qquad \mathbf{(3)}$$

The availability of A7 and A8 is only affected by number of failures and number of concurrently processed requests. To exemplify the difference in availability between standalone and cluster implementations we have created three scenarios describing characteristics of three systems. The scenarios are based on interviews with practitioners. Scenario 1 corresponds to good, Scenario 2 to average and Scenario 3 to rather bad implementation of payment system. The scenarios are presented in Table 1. The amount of requests per year was calculated from Rq_{sec} – we assume an equal distribution of requests.

Table 1. Comparison of availability of cluster versus single machine implementation for 3 different scenarios

Characteristics		Scenario 1	Scenario 2	Scenario 3
$N_{Failure}$		10	50	100
Rq_{cur}		10	10	10
T_{repair}		600	1800	3600
$T_{maintain}$		1800	3600	7200
Rq_{sec}		100	100	100
Availability results	A5,A6	99.992	99.983	99.966
	A7,A8	99.99999683	99.99998415	99.9999683

In the literature [15], argues that a standalone server usually has 99% availability. This supports our own finding that high quality systems standalone systems (A5, A6) can reach 99,9%. Alternatively, it seems that it is possible to achieve almost 100% uptime for cluster. It is probably an optimistic assumption; however "five nines" seem to be in range, even though literature suggests four [15].

5.2 *Reliability*

Reliability is described as a percentage of successfully stored payment information - equation (4).

$$\text{Reliability} = \frac{Rq_{\text{year}} - Rq_{\text{lost}}}{Rq_{\text{year}}} \tag{4}$$

Where Rq_{lost} is amount of requests lost in the year. Two types of reliability threats that can result in data loss were identified in the study. The first one occurs when a request is lost and a customer does not get an answer. This threat concerns all architectures because of the lost of requests processed during a failure. In addition, A5 and A6 do not accept incoming requests during downtime, which is also considered a data loss. The second reliability threat concerns the "internal" data loss. A request is answered but the payment record is lost due to the loss of replication or disk write buffer. This risk concerns A5 and A7. In the architectures not involving deferred replication (A6, A8) $Rq_{\text{lost}} = Rq_{\text{noanswer}}$. For A5 the Rq_{loss} is following (5):

$$Rq_{loss} = Rq_{noanswer} + N_{Failure} \cdot Rq_{diskbuf} \tag{5}$$

Where Rq_{diskbuf} is an average amount of requests in the disk buffer. For A7 the Rq_{loss} is following (6):

$$Rq_{loss} = Rq_{noanswer} + N_{Failure} \cdot Rq_{repbuf} \tag{6}$$

Where Rq_{repbuf} is an average amount of requests kept in the replication buffer.

In Table 2 a reliability assessment for the examples from Table 1 is presented. Additionally the Rq_{diskbuf} and Rq_{repbuf} are specified. Remaining values (e.g. N_{Failure}) are the same as in Table 1.

When it comes to reliability the cluster based solutions clearly outperform the standalone implementations. However, it is worth noticing that none of the architectures achieves a 0% data loss.

Table 2. Comparison of reliability of architectures for 3 different scenarios

Characteristics		Scenario 1	Scenario 2	Scenario 3
$Rq_{diskbuf}$		10	10	10
Rq_{repbuf}		10	10	10
Reliability results	A5	99.99238645	99.98286070	99.96572140
	A6	99.99238648	99.98286086	99.96572171
	A7	99.99999680	99.99998399	99.99996797
	A8	99.99999683	99.99998415	99.99996829

5.3 Performance

The performance of the selected architecture variants was evaluated in an experiment. The experiment was performed using two identical computers connected with a network for evaluation of the cluster based solutions and one of the computers for the standalone implementations. The computers ran a payment system application that used an in-memory database. We used the database produced by Times Ten, which provides both synchronous and asynchronous replication and synchronous and asynchronous disk writes options.

For each of the architectures we measured throughput and response time. Throughput was defined as the maximum number of requests that can be processed in a unit of time. The amount of requests processed was increased until we reached the saturation point where increasing the number of requests did not increase throughput but response time only. The second characteristic was the response time on a loaded system – the response time of a system that reached its saturation point. The performance bottleneck in the system was the CPU. The results of the performance measurement can be found in Table 3. The values are normalized.

Table 3. Performance of the architecture variants

Characteristics	Architecture			
	A5	A6	A7	A8
Response time results	100	172	140	240
Throughput results	100	65	70	51

6. Related work

Finding an optimal balance between availability, performance and different data quality aspects is a well known problem (e.g. [4, 5, 7, 21]). In [4] the author presents two server architectures that meet a high availability requirement. One is based on a shared-disk and the other is based on a shared-nothing paradigm. They directly correspond to A2, A3 and A4 architectures from our study. In [13] the authors present a dual-computer with 'shared nothing architecture' as a solution for a fault-tolerant server. The paper provides an implementation proposal as well as theoretical assessment of availability. Similar solution can be also found in [10]. A single computer implementation is often ignored since it usually does not provide an adequate availability level.

The 'shared nothing architecture' must involve data replication between cluster nodes. Replication is also a well studied subject, although most studies concern many co-existing replicas of the data. An overview of replication techniques can be found in [9, 19, 20]. Practically three types of replication can be identified [19]: active, semi-active and passive. The idea behind active replication is that the requests are broadcasted to all server nodes. Such a solution requires a deterministic request processing in order to assure that all servers answer in the same way [19]. We find it impossible to assure in our case. A semi-active replication solves that problem. It is achieved by dividing replicas into a leader and followers. Every time there is a non-deterministic decision to be made, the leader must make sure that it is made correctly. Semi-active replication was given attention (e.g. [2, 16]) and it seems that there is potential in it. In [2], an architecture of a real-time server based on a variant of semi-active replication is presented. Although its implementation would result in a high communication overhead (we would have to consider all updates non-deterministic), the advantage of semi-active replication is lack of current request loss in case of failure. Therefore it should be further examined and can be seen as a future study.

The passive replication is exactly what we have suggested for cluster based architectures. Many researchers try to find optimisation of passive replication. In [7] the authors notice that not all users require the same database control privileges. Any access limitation decreases the synchronization effort. It is not applicable to our solution where both nodes should behave in exactly the same way. Another optimization is designing applications that can tolerate certain inconsistencies [1]. It is also difficult to apply in our case.

7. Conclusions

The purpose of the study was to evaluate the characteristics of a payment system that can be obtained given that a system is developed on a standard platform. The system characteristics evaluated in this study were availability, reliability and performance.

The initial analysis of some architecture variants resulted in the elimination of the candidate architectures that contained a disk based database. An in-memory database seems like the only reasonable solution in our case. However, such elimination is possible only if a database can fit into memory, which is the case in our application domain. It is not an unreasonable assumption for many other application domains. A discussion concerning that issue can be found in [8].

The results of the evaluation of the selected architecture variants indicated that there exists a trade-off that must be decided by a payment system designer. As each of the four identified solutions offers unique characteristics, there is no single "winner" that could be best determined. In determining which solution to use, this depends on the individual needs as to which should be selected. For example, if high availability is required, the choice is limited to two cluster based architectures. The availability they offer costs approximately 40% of the performance decrease in terms of response time and around 30% in terms of throughput compared to standalone implementations. They also require rather expensive cluster implementation. In [10] it was shown that in a dual-computer cluster implementation about 85% of the code was devoted to providing availability, i.e. mainly failover and replication implementation. This code would not be present in standalone implementations.

Our measurements indicate that that there is also a performance price to be paid for reliability. The "reliability oriented" solutions (A6 and A8) have about 70% higher response times and 30% lower throughput compared to their "performance oriented" counterparts (A5 and A7). An immediate, synchronous replication gives better reliability but affects performance. Therefore it is up to the designer to decide if some data loss, such as in case of failure, is an acceptable price to be paid for higher performance.

None of the architectures implemented on a standard platform solves the problem of loosing requests that are currently processed. It is actually a reliability and service availability threat – a customer does not receive an answer and, what is worse, he/she can not be sure if the operation was performed or not.

We are aware that there are more advanced methods of availability and reliability estimations (e.g. [17]). Our method is rather straightforward and simple. However, the purpose of our work was not to suggest a new availability or reliability assessment technique but to present and quantify the limitations of standard platforms when implementing a Credit Control server and similar payment systems. It seems that in order to overcome the problems identified (e.g. data loss) more sophisticated solutions, like middleware or clusterware, are necessary. Since, as we know (see Paper I), they require high development effort we see it as a future study to investigate how to introduce them in a cost efficient manner.

8. Acknowledgements

The authors would like to thank Simon Kågström, Jeanette Eriksson, Lars-Ola Damm and Lawrence Henesey for reviewing this paper. This work was partly funded by The Knowledge Foundation in Sweden under a research grant for the project "Blekinge - Engineering Software Qualities (BESQ)" (http://www.bth.se/besq)

9. References

[1] D. Barbara and H. Garcia-Molina, The case for controlled inconsistency in replicated data. *Proceedings of Workshop on the Management of Replicated Data*, (1990), 35-38.

[2] P.A. Barrett, Delta-4: an open architecture for dependable systems. *IEE Colloquium on Safety Critical Distributed Systems*, (1993), 2/1-2/7.

[3] P.A. Bernstein and E. Newcomer, Principles of transaction processing, Morgan Kaufmann Publishers, San Francisco, CA, US, (1997).

[4] A. Bhide, Experiences with two high availability designs (replication techniques). *Second Workshop on the Management of Replicated Data*, (1992), 51-54.

[5] Z. Chi and Z. Zheng, Trading replication consistency for performance and availability: an adaptive approach. *Proceedings of 23rd International Conference on Distributed Computing Systems*, (2003), 687-695.

[6] J. Cornetto, Databases dive into main memory. *InfoWorld*, 20 (1998), 33.

[7] H. Garcia-Molina and B. Kogan, Achieving High Availability in Distributed Databases. *IEEE Transactions on Software Engineering*, 14 (1988), 886-897.

[8] H. Garcia-Molina and K. Salem, Main memory database systems: an overview. *IEEE Transactions on Knowledge and Data Engineering*, 4 (1992), 509-516.

[9] J. Gray, P. Helland, P. O'Neil, and D. Shasha, The dangers of replication and a solution. *SIGMOD Record 1996 ACM SIGMOD*

International Conference on Management of Data, 4-6 June 1996, 25 (1996), 173-182.

[10] D. Haggander, L. Lundberg, and J. Matton, Quality attribute conflicts - experiences from a large telecommunication application. *Proceedings of Seventh IEEE International Conference on Engineering of Complex Computer Systems*, (2001), 96-105.

[11] Y. Haifeng and A. Vahdat, The costs and limits of availability for replicated services. *Operating Systems Review. 18th ACM Symposium on Operating Systems Principles (SOSP'01), 21-24 Oct. 2001*, 35 (2001), 29-42.

[12] H. Hakala, L. Mattila, J.-P. Koskinen, M. Stura, and J. Loughney, *Diameter Credit-Control Application (internet draft - work in progress)*. (2004): http://www.ietf.org/internet-drafts/draft-ietf-aaa-diameter-cc-06.txt.

[13] G. Hui, Z. Jingli, L. Yue, and Y. Shengsheng, Design of a dual-computer cluster system and availability evaluation. *2004 IEEE International Conference on Networking, Sensing and Control*, 1 (2004), 355-360.

[14] T. McElligott, An order of magnitude. *Telephony*, 238 (2000), 70.

[15] G.F. Pfister, In search of clusters, Prentice Hall, Upper Saddle River, NJ, (1998).

[16] P. Triantafillou, High availability is not enough (distributed systems). *Second Workshop on the Management of Replicated Data.*, (1992), 40-43.

[17] M. Walter, C. Trinitis, and W. Karl, OpenSESAME: an intuitive dependability modeling environment supporting inter-component dependencies. *Dependable Computing, 2001. Proceedings. 2001 Pacific Rim International Symposium on*, (2001), 76.

[18] C. Waltner, In-Memory Databases Aid Web Customization. *InformationWeek*, (2000), 80-83.

[19] Wiesmann and P.F. M., Schiper A., Kemme B., Alonso G., Understanding replication in databases and distributed systems. *20th International Conference on Distributed Computing Systems, 2000. Proceedings.*, (2000), 464-474.

[20] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso, Database replication techniques: a three parameter classification. *Proceedings of The 19th IEEE Symposium on Reliable Distributed Systems*, (2000), 206-215.

[21] H. Yu and A. Vahdat, Building replicated Internet services using TACT: a toolkit for tunable availability and consistency tradeoffs. *2nd International Workshop on Advanced Issues of E-Commerce and Web-Based Information Systems*, (2000), 75-84.

Paper IV

A Cost-Efficient Server Architecture for Real-Time Credit-Control

Piotr Tomaszewski, Lars Lundberg, Jim Håkansson, Daniel Häggander

Proceedings of the 10th IEEE International Conference on the Engineering of Complex Computer Systems (ICECCS 2005), pp. 166-175, Shanghai, China, June 2005

Abstract

The importance of mobile and electronic commerce results in much attention given to credit-control systems. There are high non-functional requirements on such systems, e.g. high availability and reliability. These requirements can be met by changing the architecture of the credit-control system. In this paper we suggest a new architecture and a number of alternative implementations of credit-control server. By quantifying the availability, reliability, performance and cost we enable the designers to make better trade-off decisions. We compare the new architecture with the current, "state-of-the-art" solution. Finally, we present suggestions for the system developers concerning the choice of an appropriate architecture implementation variant.

1. Introduction

The popularity of on-line services, like online video or internet shopping triggers a need of creating billing mechanisms. For revenue collection the service providers often use an online billing system.
Recently the prepaid services have become very popular. The characteristic of prepaid solutions is that the service is provided only as long as the customer has adequate assets on the account. This means a real-time credit-control – the service should not be delivered when the customer's account is too low.

Diameter Credit-Control (DCC) is a standard for developing billing systems for prepaid services. Currently, its specification draft is available from The Internet Engineering Task Force [6]. It is promoted by leading companies providing billing solutions.

The rapid growth of mobile commerce market [12] increases the expectations concerning billing systems. The requirements mostly refer to reliability, availability, performance and development cost. The increasing average monetary value of a request puts a lot of attention on reliability and availability. Any data loss or service outage becomes unacceptable.

The need of combining conflicting requirements makes development of billing systems a challenging task. One way of simplifying their development is to use a specialized, fault-tolerant platform. An example of the fault tolerant platform is presented in Section 5. Such a platform is able to provide required characteristics in terms of performance, availability and reliability. However, as our previous experiences show (see Paper I), the development cost on such a platform can be very high. We have identified a factor of four difference between the productivity on a specialized platform compared to a standard platform (UNIX).

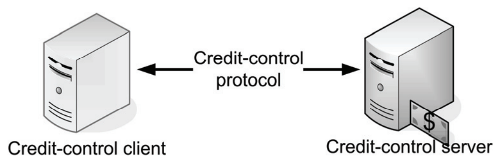
On the other hand, solutions based on standard platforms have problems providing the required availability, performance and reliability. In another study (see Paper III) we have suggested and evaluated possible credit-control server implementations on a standard platform. None of the identified variants provided 100% reliability and availability.

The purpose of this study is to check if, by combining a standard and a specialized platform, it is possible to implement, in an efficient manner, a credit-control server that meets high reliability and availability requirements. The server should be implemented according to DCC specification and evaluated from availability, performance, reliability and development effort perspectives. The result of the study should help the developers of credit-control and similar applications in deciding what technology they should use to meet, in a cost efficient way, the optimal balance between required characteristics.

2. Diameter Credit-Control

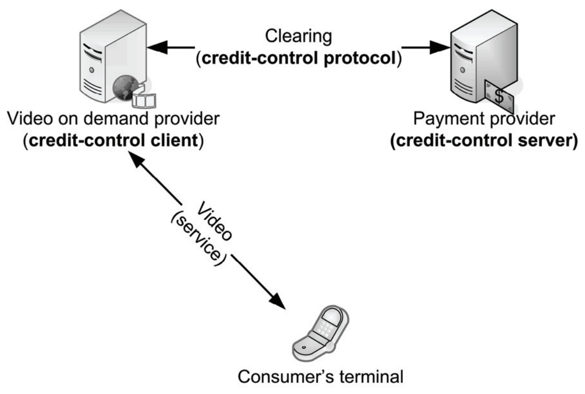
Diameter Credit-Control (DCC) defines the interaction between a credit-control client and a credit-control server. The credit-control server performs rating and accounting. Rating is an operation that maps technical units into monetary units, e.g., amount of goods or online video watching time to the amount of money the customer should pay. Accounting is an operation of recording and maintaining the information about the customer's activities. The overview of the architecture is presented in Figure 1.

Figure 1. Diameter credit-control



An example of a realization of DCC can be found in Figure 2 where the model of video-on-demand service is presented.

Figure 2. Video on demand service with DCC



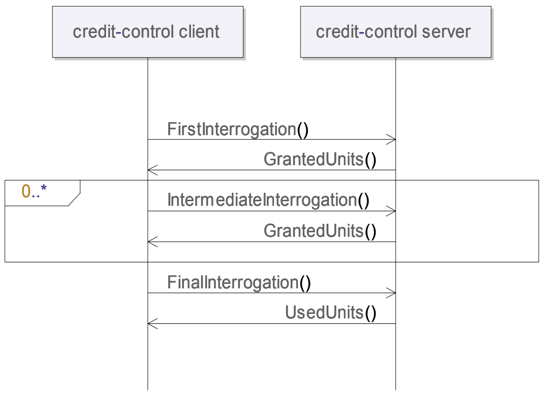
In DCC the video-on-demand provider is a credit-control client since it requests the credit-control server (payment provider) to debit the consumer. When the video-on-demand provider is asked for the next portion of the video it asks the credit-control server to reserve an appropriate amount of money on the consumer's account. If the reservation and service delivery succeeds the credit-control client asks the server to debit customer's account. Two types of services are offered by the credit-control server:

- *One Time Event* – in which one request is sent from a credit-control client to a credit-control server, e.g., an inquiry concerning an account balance.
- *Session Based credit-control* – which requires keeping session state on the server. It involves the exchange of multiple requests. Examples of such services are online video and prepaid phone calls.

Session Based credit-control is realized using interrogations (see Figure 3). Interrogation is a request that is sent to the credit-control server to get permission for continuing service delivery. The credit-control server reserves a quota that covers the estimated service cost for the next reservation period and charges the customer for the service delivered from the previous interrogation. In this way the service is divided into portions (e.g. number of minutes of movie or voice conversation). Each interrogation reserves assets for the next such portion and charges the consumer for the previous.

In this study we consider a simple credit-control client implementation that sends a request to the server only once. If no answer is given, the credit-control client assumes negative answer and stops service delivery. No request re-sent is performed by the client.

Figure 3. Interrogations



3. Related work

The development of systems that have highly prioritized availability and reliability requirements always results in the necessity of making trade-offs. In [7, 17, 18] the authors discuss the trade-off between availability and data consistency, in [9] the trade-offs between maintainability, performance and availability. Also the trade-off between performance and consistency has been recognized [18]. The problem with the required qualities is that they are dependant, e.g. availability usually involves data replication, which from a consistency perspective means a lot of state updating communication, highly undesired from performance perspective. Therefore there is no silver bullet solution in that matter and different suggestions are usually very application specific.

Both availability and performance are in the literature [2, 14] often discussed in the context of replication. According to [2] there are two main strategies when it comes to replication: "*replication of the server*" and "*replication of the server and the resource*". As server the stateless part of application is concerned. In our case it would be the rating component. The server operates on the resource, which, in our case, is the database.

The "*replication of the server*" means that both credit-control nodes share a single database. Such a solution was presented in [5] for the system similar to ours. As a benefit of such solution lack of overhead connected with replication and lack of consistency problems is presented. An example of "*replication of a server and resources*" is the "state-of-the-art" architecture described in this work. When it comes to this solution the main research effort was devoted to develop the efficient replication schemes (e.g. [4, 8, 16]). The replication schemes can provide balance between performance and consistency. However, there is no way of solving the problem of loss of data processed during failure without co-operation from the client.

As a way to solve this problem a specialized platform can be considered. Example of such solution is described in [13], where CORBA was extended with automatic object replication, migration and restart of objects that failed. Another example of that kind is [15] - it is an operating system that runs on a cluster of computers. It provides internal data replication and migration capabilities. The main problem with this kind of solutions, as we see it (see Paper I), is a low productivity of software development for such platforms.

4. Method

In this study we suggest a new credit-control server architecture and present number of its implementation variants. We evaluate and compare them with the “state-of-the-art” credit-control implementations. The evaluation is done from the reliability, availability, performance and development cost perspectives.

We define reliability as the probability that a request was successfully processed and stored. The availability is the probability that the request sent to the server is answered. The performance is measured as the maximum throughput of the system and the response time of the saturated system. The development cost is estimated and describes the effort of developing each of the implementation variants.

The reliability and availability are defined as functions of certain implementation and platform characteristics, e.g. downtime and amount of failures. To make the results tangible we present characteristics of three implementations and evaluate their availability and reliability for each of the new architecture implementation variants. For availability and reliability evaluations we make the standard assumption that only one component fails at the time.

For the purpose of performance measurement a prototype of the system is implemented. The evaluation is performed in an industrial setting – the prototype involves crucial (from performance perspective) parts of a real payment system. Therefore, according to the agreement with our industrial partner, the performance figures are normalized.

The development effort is estimated by experts involved in development of billing systems. The estimation concerned amount of functionalities that must be implemented and the effort connected with it. Based on that, we estimated the cost of the solutions.

This study begins with a presentation of the “state-of-the-art” architectures, which we have identified in a previous study (see Paper III). We present two credit-control server implementation variants based on a standard platform. We evaluate them from the reliability and availability perspectives as well as we measure their performance. We identify issues that affect reliability and availability of the “state-of-the-art” solutions.

As a solution to the availability and reliability problems we suggest a new architecture. It combines standard and fault-tolerant platform. We describe an example of a fault-tolerant platform. For the new architecture we present variants of a credit-control server implementation. The variants are evaluated and compared to the "state-of-the-art" solutions.

Finally we summarize findings and presented suggestions concerning the choice of the architecture when certain balance of requirements is expected.

5. "State-of-the-art" solution

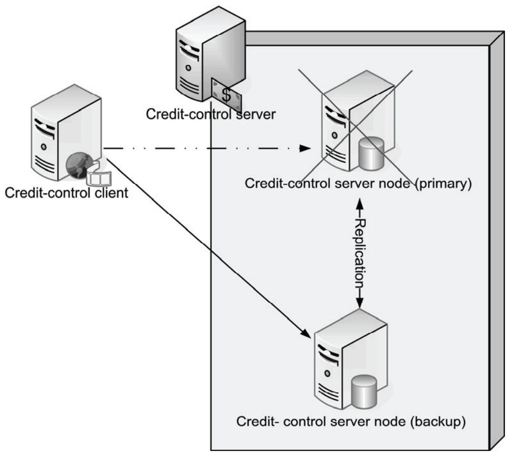
5.1 Presentation

In Paper III we have identified four "state-of-the-art" architecture variants of a credit-control server implementation on a standard platform. Two of them are implemented using a single machine and offer rather low availability and reliability levels. Their main advantage is low development cost. The remaining two architectures are based on a shared-nothing cluster [14]. They offer high availability and reliability. Since, in this study, we are interested mostly in availability and reliability, only the two cluster-based variants are taken into consideration.

Both variants are based on a hot backup server architecture [14]. It consists of two server nodes – a primary and a backup. When primary fails, a failover is performed. The secondary node becomes the primary (Figure 4) and takes over operation. Each of the nodes has the same configuration. Data storage is provided by a database. For performance reasons an in-memory database is used. Such a database, by keeping all the data in the RAM memory, speeds up the operations by an order of magnitude [3].

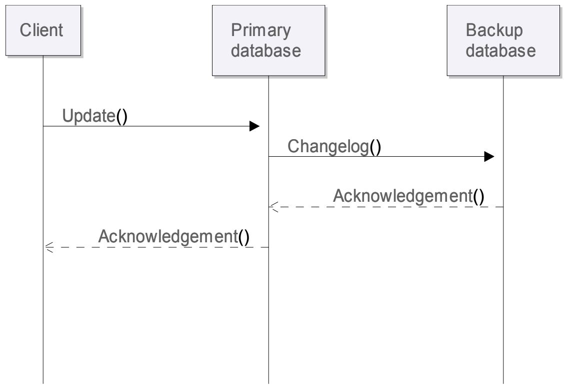
Since both nodes must operate on the same data there is a replication between the databases. The difference between the two cluster based architecture variants is that one uses synchronous and another one uses asynchronous replication. From now on we call the variants SYNC and ASYNC.

Figure 4. Failover scenario – when primary node fails the backup takes over the operation


Credit-control server
Credit-control client
Credit-control server node (primary)
Replication
Credit- control server node (backup)

In synchronous replication mode (Figure 5) the database transaction returns when the changes are committed in both databases. In asynchronous mode the changes are committed only in the primary server before the transaction returns (Figure 6). The replication of changes is deferred. That improves performance – not only the round trip to another node is not included in the transaction but also network traffic is decreased by sending a number of changes grouped together instead of sending them one by one.

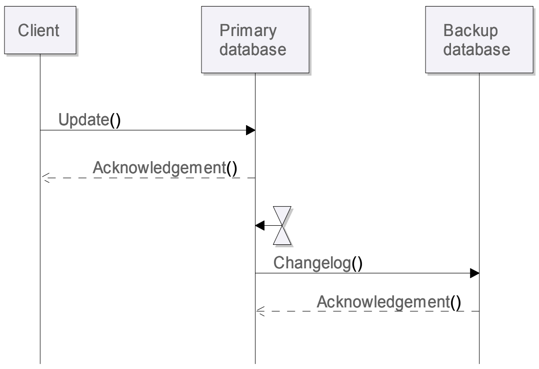
Figure 5. Synchronous replication



The drawback with the asynchronous replication is that the backup server's state is always "behind" the primary's one. A number of requests that were not replicated are lost when primary node fails. We refer to that amount of requests as to "replication buffer".

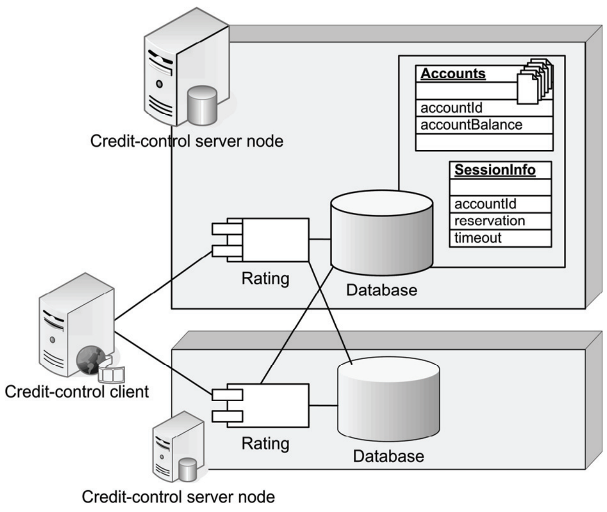
Internally the credit-control server node is divided into two logical parts (Figure 7) [5]. Rating is the stateless element responsible for rating. Database is the stateful part responsible for accounting. The main Database tables are Accounts (user accounts) and SessionInfo (on-going sessions information, e.g., reservations). Only the Accounts table is replicated.

Figure 6. **Asynchronous replication**



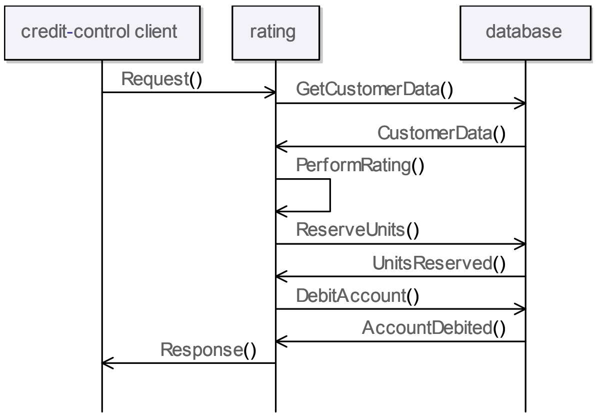
As can be noticed in Figure 7 the rating component can contact both databases. By default it uses the one that is on the same machine, but if it can not be contacted the rating performs the operation on the backup node's database. The interrogation handling is presented in Figure 8.

Figure 7. **"state-of-the-art" implementation of credit-control server**



The rating component gets a request from the client. It asks the database for information about account balance and reservations. Based on that, it performs rating. It decides how much money should be reserved and how much should be debited. Later it reserves money (update of SessionInfo table) and debits the actual account (update of Accounts table).

Figure 8. Interrogation processing



5.2 *Evaluation*

Planned downtime does not affect availability or reliability of cluster based solutions (the work can be migrated to the other node). The only availability and reliability threat is unplanned downtime (failure) that results in data loss. In Paper III we have identified two major data loss threats:

- *"current requests loss"* – the loss of currently processed requests. In the system there are always a number of concurrently processed requests, which are lost in case of failure. We will denote average number of requests in the system as Rq_{cur}
- *"replication buffer loss"* - loss of "replication buffer" in case of asynchronous replication. The average amount of requests in the replication buffer will be referred to as Rq_{buf}

The "*replication buffer loss*" affects only data reliability – the lost data was committed to the database which means that the customer got an answer. Therefore, the service availability of both solutions is the same and can be calculated as (see Paper III):

$$\text{Avail} = \frac{Rq_{\text{year}} - N_{\text{Fail}} \cdot Rq_{\text{cur}}}{Rq_{\text{year}}}$$

where Rq_{year} is amount of requests per year and N_{Fail} is number of application or platform failures per year. The reliability of replication modes is different. In SYNC it has the same value as availability:

$$\text{Reliability}_{\text{SYNC}} = \frac{Rq_{\text{year}} - N_{\text{Fail}} \cdot Rq_{\text{cur}}}{Rq_{\text{year}}}$$

ASYNC has lower reliability because the data loss is increased by the of "*replication buffer loss*":

$$\text{Reliability}_{\text{ASYNC}} = \frac{Rq_{\text{year}} - N_{\text{Fail}} \cdot (Rq_{\text{cur}} + Rq_{buf})}{Rq_{\text{year}}}$$

To estimate the availability and reliability level we can expect from SYNC and ASYNC implementation variants we created three different scenarios describing characteristics of three different system implementations. The results are presented in Table 1. The amount of requests per year was calculated based on a number of requests per second (Rq_{sec}).

Table 1. Reliability and availability evaluation

	Scenario		
	1	**2**	**3**
N_{Fail}	10	50	100
Rq_{cur}		10	
Rq_{buf}		10	
Rq_{sec}		100	
Availability	99.99999683	99.99998415	99.9999683
Reliability ASYNC	99.99999680	99.99998399	99.99996797
Reliability SYNC	99.99999683	99.99998415	99.99996829

The higher reliability of SYNC implementation variant has a performance price. In Paper III we have detected 30% throughput decrease and 70% response time increase in SYNC compared to ASYNC. The results of the performance measurement can be found in Table 2. The values were normalized – the ASYNC variant's results were used as a baseline and normalized to 100. The response time was measured when the system was saturated, i.e. when increasing load increased response time but not throughput.

Table 2. Performance evaluation results

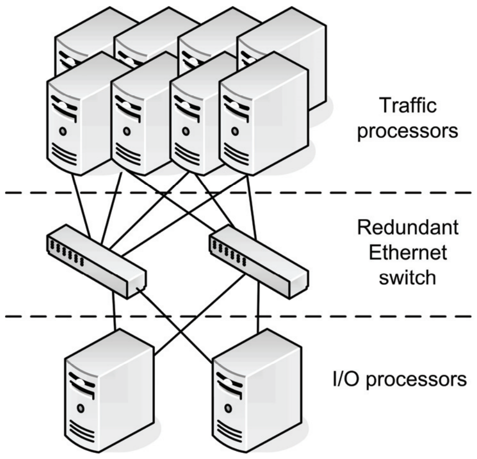
	ASYNC	**SYNC**
Response time (RT)	100	171
Throughput (T)	100	73

6. Fault-tolerant platform

Fault tolerance is defined as “the ability of a system or component to continue normal operation despite the presence of hardware or software faults” [10]. Typically fault tolerance is either hardware or software based. Hardware based solutions, e.g. [1, 11], involve proprietary hardware, which makes them very expensive. A software based solutions overcome this problem by operating on a standard hardware. An example of a software based solution is the fault-tolerant platform, which we have evaluated in (see Paper I). It is a system that consists of (Figure 9):

- Up to 40 traffic processors that process pay-load
- Two I/O processors responsible for the external communication and maintenance
- Two Ethernet switches and two separate interconnections via Ethernet networks

Figure 9. A fault-tolerant platform example



The platform offers standard interfaces (APIs) for Java and C++ but the programming model is unique. The execution unit is a process. There are static processes that are always running and dynamic ones that are created and terminated on demand. The platform provides an in-memory database. The basic storage units are database objects.

To assure an efficient load balance the programmer has a set of methods for allocating database objects and processes to processor pools, i.e. sets of traffic processors on which database objects and processes may operate. The load balancing within a pool is done by the platform itself.

To facilitate the programming of the highly available systems every process or database object is replicated on two machines in the cluster – a failure of one of them does not affect the correct operation of the whole system. The platform also has built-in features that allow online upgrades of the applications.

In Paper I we have investigated the productivity on such a platform. It turned out to be factor of four lower than on a standard platform. As the main cost drivers we have identified long learning and lack of tools and libraries that are available for standard platforms.

7. New architecture

7.1 Basic idea

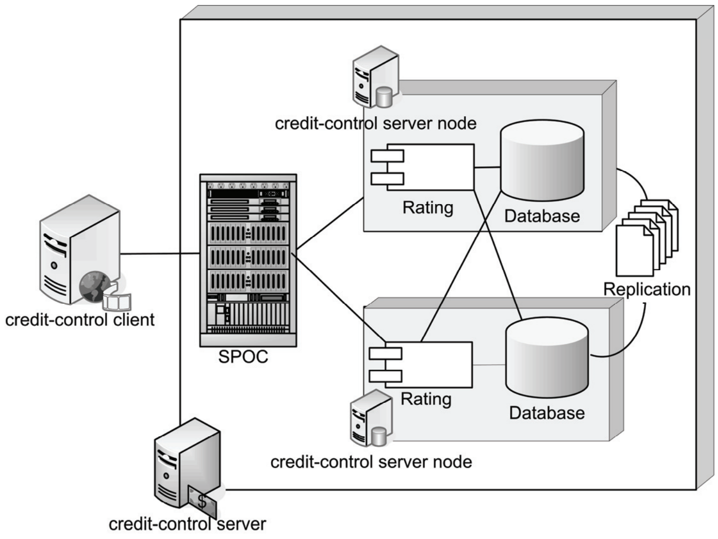
In Section 5 we have presented two “state-of-the-art” implementations of the credit-control server. None of them provided 100% availability and reliability. The best solution from reliability perspective is also the worst one from the performance point of view. Using the “state-of-the-art” architecture requires performance-reliability trade-offs. Such trade-offs can be avoided by implementing the credit-control server on a fault-tolerant platform, like the one described in Section 6. The price for high reliability and performance is significantly higher development cost.

Our idea is to combine standard and fault-tolerant platform into one architecture. By keeping as much as possible of the functionality on a standard platform we can decrease the development cost. The fault-tolerant platform can contribute to good availability and reliability. Such a mixture of a reasonable development cost and good availability and reliability is not provided by any of currently available solutions. Therefore we consider it interesting to investigate.

7.2 Architecture overview

In the new architecture to the “state-of-the-art” credit-control server (cluster with two computers) we have added a Single Point of Contact (SPOC). The new architecture is presented in Figure 10. The credit-control client always contacts a single designated machine. The SPOC is implemented on the fault-tolerant platform. Therefore, in the further discussion, SPOC is not considered a point of failure.

Figure 10. New architecture



The basic request processing is similar to the one on the "state-of-the-art" architecture. The SPOC acts as a proxy and passes requests from client to the credit-control server node and back. The additional requirement is that SPOC should provide enough processing power not to become a bottleneck of the system. We will estimate the processing power required from the SPOC.

7.3 Implementation variants

In Section 5.2 we have identified two sources of data loss in state-of-the-art solution, namely "*current request loss*" and "*replication buffer loss*". The introduction of SPOC gives us a chance to minimise or eliminate their impact.
One way of increasing reliability and availability is to introduce request re-sending on SPOC. In the "*request re-send"* method the SPOC re-sends the request if the response from the node is not received within a specific amount of time.

The main threat connected with re-sending of request is that it may be processed more than once, which may result in overcharging. Therefore we introduce mechanism on the credit-control server node that prevents it. Following changes are suggested:

- In the SessionInfo table we keep the id of the last request from the session together with the response that was sent to the customer
- When charging the rating component checks if the request was not answered. If it was, the response from the SessionInfo is sent to SPOC
- The SessionInfo table is replicated

There is a problem that re-sent request may reach the server while the original one is still processed. Since the answer to both requests may be different (e.g. due to tariff change) we must assure that the response to the client is consistent with the change in database. For that purpose we introduce additional serial number given by SPOC to each request. This number is included in the answer from the node. Only the response to the request with the last serial number is transmitted to the client. Also the replication conflict (conflicting changes of the same record) should be resolved by selecting the change caused by request with higher serial number.

The "*request re-send*" solves the problem of "*current request loss*". It does not, however, solve the *"replication buffer loss"* problem that is present when asynchronous replication is used. To address this we suggest introducing a "*request database*", which is an extension of "*request re-send*". The idea is to keep recent requests on SPOC as long as they are not replicated to both nodes. When primary node fails the SPOC re-sends the recent requests to backup node to make sure its state is updated. The amount of requests to re-send may be based either on time criterion or more complex solution may be suggested. One way is to mark each update with information on which node it was committed. The backup node, on regular bases, selects all requests committed on the primary one and informs SPOC that they are already replicated. SPOC removes them from the request database.

For further analysis we suggest three architecture implementation variants, presented in Table 3.

Table 3. The architecture implementation variants

Variant	Description
SPOC+SYNC	SPOC + request re-sending + synchronous replication
SPOC+ASYNC	SPOC + request re-sending + asynchronous replication
SPOC+RQDB	SPOC + request re-sending + asynchronous replication + request database

8. Evaluation

8.1 Availability and reliability

Both availability and reliability are affected by data loss. The new architecture variants ability to solve *"current requests loss"* and *"replication buffer loss"* are summarized in Table 4.

The availability of both "state-of-the-art" solutions (SYNC and ASYNC) was affected only by the *"current requests loss"* problem. This problem is solved in all variants of the new architecture. Therefore all 3 variants have 100% availability.

Table 4. Problems solved in the new architecture implementation variants

Variant	"current requests loss"	"replication buffer loss"
SPOC+SYNC	solved	N/A
SPOC+ASYNC	solved	unsolved
SPOC+RQDB	solved	solved

When a synchronous replication is used the reliability is affected only by the *"current requests loss"* problem. In asynchronous replication additionally the *"replication buffer loss"* affects it. This problem is not solved only in SPOC+ASYNC. Therefore the reliability of SPOC+SYNC and SPOC+RQDB will be 100%, while the reliability of SPOC+ASYNC is following:

$$\text{Reliability}_{\text{SPOC+ASYNC}} = \frac{Rq_{\text{year}} - N_{\text{Fail}} \cdot Rq_{buf}}{Rq_{\text{year}}}$$

In order to compare the reliability and availability of the "state-of-the-art" and the new architectures in a more tangible manner we perform an estimation of reliability and availability for the same three scenarios describing characteristics of three different system implementations as in the Table 1. The results are presented in Table 5. For the comparison the results of the "state-of-the-art" variants are included.

In the Table 5 it can be noticed that even though the reliability of SPOC+ASYNC is not 100, in practise it still improves the reliability of ASYNC.

Table 5. Reliability and availability – comparison of state-of-the-art and the new architecture

		Scenario		
		1	**2**	**3**
N_{Fail}		10	50	100
Rq_{cur}		10	10	10
Rq_{buf}		10	10	10
Rq_{sec}		100	100	100
Availability [%]	ASYNC SYNC	99.99999683	99.99998415	99.9999683
	SPOC+SYNC SPOC+ASYNC SPOC+RQDB	100	100	100
Reliability [%]	ASYNC	99.99999680	99.99998399	99.99996797
	SYNC	99.99999683	99.99998415	99.99996829
	SPOC+SYNC	100	100	100
	SPOC+ASYNC	99.99999683	99.99998415	99.9999683
	SPOC+RQDB	100	100	100

8.2 Performance

The performance, in terms of throughput and response time, depends on how powerful the SPOC machine is compared to the server node that processes requests. The throughput (T) of the entire system is as high as the lower of SPOC and the node throughputs:

$$T_{SYSTEM} = \min(T_{SPOC}, T_{NODE})$$

Because we do not want to introduce a bottleneck into the system, the SPOC has to provide a high enough throughput ($T_{SPOC}>T_{NODE}$). In this way the throughput of the new architecture variants is the same as of their counterparts without SPOC. To assess the processing power that is required from the SPOC we implemented the SPOC functionality on the cluster with asynchronous replication between databases (identical with ASYNC configuration) and we measured throughput and response time on it.

The results were compared with throughput and response time of SYNC and ASYNC credit-control server implementation variants. Under normal conditions (no failure) the data processing routines on the SPOC are the same for all three architecture variants (SPOC+ASYNC, SPOC+SYNC, SPOC+RQDB). The results of the throughput comparison are summarized in Table 6.

Table 6. The comparison of the SPOC and the node performance

	ASYNC	SYNC	SPOC
Response time (RT)	100	171	57
Throughput (T)	100	73	267

We measured the performance of the whole credit-control server that includes SPOC. The comparison can be found in Table 7.

Table 7. Comparison of performance of the credit-control server implementation variants

	ASYNC	SYNC	SPOC		
			ASYNC	ASYNC	ASYNC
Response time (RT)	100	171	169	240	169
Throughput (T)	100	73	100	73	100

Like before, the response time was measured when the system was saturated, i.e. when increasing load increased response time but not throughput.

9. Cost

The cost evaluation was based on three factors:

- the ratio of functionality between the SPOC and the credit-control server node
- the ratio of development effort between implementation of a functionality on a fault tolerant platform and on a standard platform
- the cost of additional features in the nodes that enable their co-operation with the SPOC

The ratio of functionality between SPOC and the credit-control node and the cost of additional features in the nodes were estimated by experts. The experts had knowledge about standard and fault-tolerant platform as well as experience in developing payment systems. According to them the amount of SPOC functionality in the SPOC+SYNC and SPOC+ASYNC corresponds to approximately 5% of the amount of the node functionality. The SPOC+DBRQ is more complex and its SPOC functionality corresponds to about 7% of the node functionality. The estimations were based on similarities between components that would have to be developed to implement the new architecture and already existing components implementing similar functionalities in other projects.

The ratio of a cost of functionality implementation between two platforms was investigated by us in Paper I. We found a factor of four difference between a functionality development cost.

The cost of additional features that must be added to the nodes was estimated as 5% in all three cases, compared to the “state-of-the-art” implementation.

The cost estimation is presented in Table 8. The values are normalized. It should be noticed that the cost of development on a new architecture is significantly smaller compared to the development of the whole credit-control system on the fault-tolerant platform which, according to our estimations, would cost about 400. This is about three times as much as the cost of development on our architecture.

Table 8. Development cost estimation

	ASYNC	SYNC	SPOC		
			ASYNC	SYNC	RQDB
Cost	100	100	125	125	133

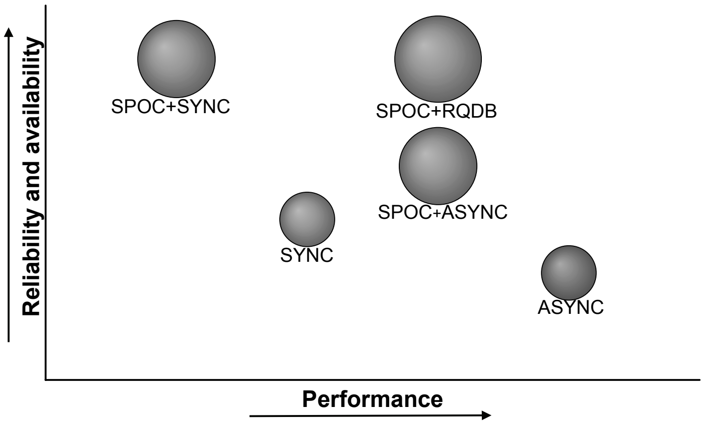
10. Discussion

The results obtained in our study look promising. All three architecture variants improve the reliability and availability of the standard, “state-of-the-art” solution. Two of them provide 100% reliability and availability. Their development cost is significantly smaller compared to the implementation on a fault-tolerant platform. According to our estimations the cost corresponds to about 30% of the cost of implementation on a fault-tolerant platform.

In the example presented we have also shown that, provided high enough throughput of the SPOC, the performance price for increasing availability and reliability may be reasonable in terms of response time and none in terms of throughput. Moreover we have shown that the high throughput of the SPOC can be provided for rather low price – the platform used for SPOC must provide only about 30% to 40% of processing power of the standard machine.

The three architectures offer different qualities. An overview of their characteristics is presented in Figure 11 (the “state-of-the-art” solutions are included for comparison). The size of the bubbles indicates the difference is cost between the architecture variants. Neither the distances between the bubbles nor their size ratios indicate the actual magnitude of the differences. The figure is only meant to outline the general differences between the architectures.

Figure 11. Characteristics of the architectures. The bubble size describes the development cost



The SPOC+SYNC architecture variant offers a very high availability and reliability and reasonable development cost. The price for that is the lowest performance of all architectures. The performance is better in SPOC+ASYNC variant but it does not provide equally high reliability level. High availability and reliability together with good performance is provided by SPOC+RQDB. This variant has, however, the highest cost of all.

When discussing the solution with our industrial partners they mentioned number of opportunities connected with introducing the SPOC. The quality pressure put on the credit-control node platform can be reduced – a higher number of node failures would not result in decreased availability or reliability (at least in SPOC+SYNC and SPOC+RQDB). It may also be possible to use a single SPOC for number of credit-control servers. Both these issues can decrease the cost of SPOC introduction, which make the new architecture even more interesting.

11. Conclusions

The objective of this study was to check if, by combining a standard and a specialized platform, it is possible to implement, in a cost-efficient manner, a credit-control server that meets high reliability and availability requirements. Our results show that, by using this approach, we can suggest an architecture that offers 100% availability and reliability for reasonably low price.

To achieve it we have analysed the current “state-of-the-art” of credit-control server – a cluster of two computers with database replication between them. We have described its reliability and availability problems. As the most important we have considered reducing the number of requests that are lost.

In order to solve the data loss problem in front of a two node cluster we have introduced a single point of contact application (SPOC) implemented on a fault-tolerant platform. For the new architecture we have designed three different credit-control server implementation variants. The variants have been evaluated from availability, reliability, performance and development cost perspectives. In the evaluation we have quantified the characteristics and compared them with the “state-of-the-art” solutions.

All variants provide 100% availability which is significantly better compared to the “state-of-the-art” solutions. Two variants provide 100% reliability. The reliability of the third one is not 100% but is better than the reliability of its “state-of-the-art” counterpart.

The performance of the new architecture depends on the performance of the SPOC. We have quantified the level of processing power required from the SPOC to maintain the same throughput as the one offered by the “state-of the-art” solutions. It turned out that the requirements concerning SPOC platform are significantly lower than the ones concerning the nodes implemented on a standard platform.

The evaluation of the development cost showed that, even though the new architecture variants cost about 30% more compared to the “state-of-the-art” implementations, they provide the same level of availability and reliability as the implementation on a fault-tolerant platform for about one-third of the price.

We have explicitly quantified availability, reliability, performance and development cost which allows the designers to make better trade-off decisions. Depending on the required quality levels and the resources available we suggest a concrete architectural solution.

12. Acknowledgments

This work was partly funded by The Knowledge Foundation in Sweden under a research grant for the project "Blekinge - Engineering Software Qualities (BESQ)" (http://www.bth.se/besq).

13. References:

[1] K.H. Benton B., Yamada H., A fault-tolerant implementation of the CTRON basic operating system. *Proceedings of the 11th TRON Project International Symposium*, (1994), 65-74.

[2] P.A. Bernstein and E. Newcomer, Principles of transaction processing, Morgan Kaufmann Publishers, San Francisco, CA, US, (1997).

[3] H. Garcia-Molina and K. Salem, Main memory database systems: an overview. *IEEE Transactions on Knowledge and Data Engineering*, 4 (1992), 509-516.

[4] R. Guerraoni and A. Schiper, Software-based replication for fault tolerance. *Computer*, 30 (1997), 68-75.

[5] D. Haggander, L. Lundberg, and J. Matton, Quality attribute conflicts - experiences from a large telecommunication application. *Proceedings of Seventh IEEE International Conference on Engineering of Complex Computer Systems*, (2001), 96-105.

[6] H. Hakala, L. Mattila, J.-P. Koskinen, M. Stura, and J. Loughney, *Diameter Credit-Control Application (internet draft - work in progress)*. (2004): http://www.ietf.org/internet-drafts/draft-ietf-aaa-diameter-cc-06.txt.

[7] A. Hisgen, A. Birrell, T. Mann, M. Schroeder, and G. Swart, Availability and consistency tradeoffs in the Echo distributed file system. *Proc. of the Second Workshop on Workstation Operating Systems*, (1989), 49-54.

[8] Y. Huang and P. Jalote, Availability analysis of the primary site approach for fault tolerance. *Proceedings of the Eighth Symposium on Reliable Distributed Systems*, (1989), 130-136.

[9] D. Häggander, Software design conflicts: maintainability versus performance and availability, Ph.D. Thesis, Blekinge Institute of Technology, (2001).

[10] IEEE Computer Society. Standards Coordinating Committee., IEEE standard computer dictionary: a compilation of IEEE standard computer glossaries, 610, Institute of Electrical and Electronics Engineers, New York, NY, USA, (1990).

[11] D. Jewett, Integrity S2: a fault-tolerant Unix platform. *21st International Symposium on Fault-Tolerant Computing*, (1991), 512-519.

[12] M. Lilge, Evolution of PrePaid Service towards a real-time payment system. *IEEE Intelligent Network Workshop*, (2001), 195-198.

[13] S. Maffeis, Piranha: A CORBA tool for high availability. *Computer*, 30 (1997), 59-67.

[14] G.F. Pfister, In search of clusters, Prentice Hall, Upper Saddle River, NJ, (1998).

[15] M. Toeroe, Performance simulation of the Jambala platform. *Proceedings of 35th Annual Simulation Symposium*, (2002), 190-197.

[16] P. Triantafillou and C. Neilson, Achieving Strong Consistency in a Distributed File System. *IEEE Transactions on Software Engineering*, 23 (1997), 35-56.

[17] H. Yu and A. Vahdat, Building replicated Internet services using TACT: a toolkit for tunable availability and consistency tradeoffs. *2nd International Workshop on Advanced Issues of E-Commerce and Web-Based Information Systems*, (2000), 75-84.

[18] C. Zhang and Z. Zhang, Trading replication consistency for performance and availability: an adaptive approach. *Proceedings of 23rd International Conference on Distributed Computing Systems*, (2003), 687-695.

Paper V

Improving Fault Detection in Modified Code - A Study from the Telecommunication Industry

Piotr Tomaszewski, Lars Lundberg, Håkan Grahn

To appear in Journal of Computer Science and Technology, special issue on Advances in Software Metrics and Software Processes (2006)

Abstract

Many software systems are developed in a number of consecutive releases. In each release not only new code is added but also existing code is often modified. In this study we show that the modified code can be an important source of faults. Faults are widely recognized as one of the major cost drivers in software projects. Therefore, we look for methods that improve the fault detection in the modified code. We propose and evaluate a number of prediction models that increase the efficiency of fault detection. To build and evaluate our models we use data collected from two large telecommunication systems produced by Ericsson. We evaluate the performance of our models by applying them both to a different release of the system than the one they are built on and to a different system. The performance of our models is compared to the performance of the theoretical best model, a simple model based on size, as well as to analyzing the code in a random order (not using any model). We find that the use of our models provides a significant improvement over not using any model at all and over using a simple model based on the class size. The gain offered by our models corresponds to 38% to 57% of the theoretical maximum gain.

1. Introduction

Finding and fixing faults is a very expensive activity in the software development process [38]. In large telecommunication systems fault detection activities can account for a significant part of the project budget, e.g., in [7] 45% of the project resources were devoted to testing and simulation. Therefore, an increase of the fault detection efficiency can potentially bring significant savings on project cost. A well-known fact concerning faults is that a majority of the faults can be found in a minority of the code (the Pareto principle [4, 12, 31]). Different sources report different numbers concerning the Pareto principle, ranging from 20-60 (60% of the faults can be found in 20% of the modules) to 10-80 (see [12] for a brief overview of the research concerning the Pareto principle). The Pareto principle shows that there is a potential for significant savings if we manage to focus our testing efforts on the most fault prone code units.

One way of helping testers to focus their efforts is to provide them with a fault prediction model. If we assume that the cost of finding faults in the class is proportional to the size of the class (like in [5, 6]) then, by selecting classes with the highest fault density, such a prediction model increases the *fault detection efficiency* (i.e., the number of faults found per the amount of code analyzed). In the long run, increasing the fault detection efficiency leads to higher quality of the products because testers focus on finding and removing faults in the classes that have the highest concentration of faults (fault density). As a result, they remove more faults within a given budget. Therefore, in this study we develop fault prediction models that predict fault density.

Fault prediction models are usually based on either different characteristics of the software that describe the structure of the code (e.g., design or code metrics [10, 41, 44]) or historical information about the code (e.g., [34, 35]). Our models are based on design and code metrics. We perform our analysis at the class level, i.e., our predictions concern the fault-proneness of individual classes and are based on the characteristics of those classes. We predict the fault density in two ways - by predicting the fault density itself and by predicting the number of faults in a class and dividing it by the size of this class.

Our models are built and evaluated using data from two different telecommunication systems developed by Ericsson. From now on we denote them as System A and System B. In this study we have used two releases of System A (from now on called System A1 and System A2) and one release of System B. These are the most current releases of both systems (the current release of System B and the two latest releases of System A). Both systems are large telecommunication systems. Their sizes are about 800 classes (500 KLOC) and about 1000 classes (600 KLOC) for System A and System B, respectively. Both systems operate in the service layer of mobile phone network. As they are mission-critical for the customers, they undergo an extensive testing before they are released.

Both systems are mature systems that have been present in the market for several years. Over that period a number of releases of each system have been produced. Each new release usually introduces a significant amount of new functionality. Typically, new functionality is introduced by modifying existing classes and/or implementing new classes. In System A1 the modification of classes from the previous release accounted for 65% of the code written in the current release (35% of the new code was introduced as new classes). In System A2 37% of the code was introduced as a modification of previous classes, and in System B 44% of the code was introduced as a modification of the classes from the previous release. A well-known fact is that a modification of already existing code is an important source of faults [34, 37]. This is supported by our data. Faults found in the modified code accounted for 86%, 62%, and 78% of all faults found in System A1, System A2, and System B, respectively. It can be noticed that in all three systems the modified code was significantly more fault-prone compared to the new code.

In this study we build and evaluate models that predict faults specifically in modified code, which is different from most studies in the area that do not distinguish between new and modified code (see Related Work section). One reason for focusing on the modified code is that, as we have shown, the modified code is an important source of faults. Focusing on the modified code also gives us an opportunity to include not only usual metrics that describe the structure of the final product (e.g., size, complexity) but also metrics that describe the characteristics of the modification (e.g., the number of new and modified lines of code in the class). Also many studies in the fault prediction domain predict faults at the component or module level [15, 17, 21-24, 31, 32, 35]. As we have shown in [40], the class level prediction, which we suggest in this paper, is of higher precision and therefore is likely to bring higher improvements.

We arbitrarily select System A1 as the system on which we build our models. The models are later evaluated by applying them to System A2 and System B. In this way we check if our models are stable across different releases of the same system as well as across different systems. We show that the models increase the efficiency of fault detection in similar way in all three systems.

The rest of the paper is structured as follows: in Section 2 we present the work that has been done by others in the area of fault prediction. Section 3 describes the methods we have used for model building and evaluation. Section 4 presents the results we have obtained. In Section 5 we discuss our findings and different validity issues. In the last section (Section 6) we present the most important conclusions from our study.

2. Related work

Fault prediction models that predict the number of faults or the fault density are very common in literature (e.g., [7, 8, 30, 33, 43, 44]). The most typical methods for building prediction models are different variants of linear regression (e.g., [7, 8, 30, 32, 43, 44]). Other methods include, e.g., negative binomial regression [33]. Usually the construction of prediction model starts with selecting *independent variables* (variables that are used to predict the *dependant variable* - faults). The most common candidates are different code metrics (e.g., [21, 35, 44]) or variations of Chidamber and Kemerer (C&K) [9] object oriented metrics (e.g., [5, 10, 44]). There are also studies that take historical information about the code fault-proneness into account (e.g., [33-35]). The initial set of independent variables is often large (e.g., over 200 metrics in [14]). A common assumption is that models based on a large number of variables are less robust and have lower practical value (more metrics have to be collected) [7, 11]. Therefore, the first step of model building usually involves a reduction of the number of metrics. A commonly used method for the dataset reduction is a correlation analysis ([7, 10, 44]). It is usually used to detect highly correlated metrics. Highly correlated metrics can, to a large extent, measure the same thing (e.g., the number of code lines and the number of statements are usually highly correlated because both measure size). Including them into the model causes a risk for multicolinearity [7]. Multicolinearity is especially risky when regression models are built. It leads to "*unstable coefficients, misleading statistical tests, and unexpected coefficient signs*"[11]. Correlation analysis is also used for selecting independent variables to predict faults (e.g., [31, 44]). Only those metrics that are correlated with faults are good fault predictors.

Below, we present studies in which fault prediction models were built. For each study we describe the set of metrics used, the metric selection criteria, and the results obtained. In the cases of prediction models built using linear regression we also quote R^2 values. R^2 is a "goodness-of-fit" measure that describes how well the model fits the data it was built on. It describes the proportion of variability of variable predicted by the model [20]. Therefore, it has values between 0 and 1 [27]. The closer R^2 is to 1 the better is the prediction model. For details concerning the calculation of R^2 see Section 3.3.

In [44], Zhao *et al.* compare the applicability of design and code metrics to predict the number of faults. The analyzed system is one release of a large telecommunication system. The authors do not say if the code analyzed is new or modified. The design metrics collected are mostly different SDL related metrics (the number of SDL diagrams, the number of task symbols in SDL descriptions, etc.). The code metrics included the number of lines of code, the number of variables, the number of signals, and the number of if statements. The initial selection of metrics is based on the correlation analysis. To build the models the authors use the stepwise regression, which additionally eliminates the metrics that are not good as fault predictors. The authors conclude that both code and design metrics are applicable and give good results. However, in this study, the best fault prediction is obtained when both types of metrics are included in the same model. R^2 values obtained in this study are 0.63 for the design metrics model, 0.558 for the code metrics model, and 0.68 for the model based on design and code metrics.

The applicability of object-oriented metrics for predicting the number of faults is evaluated by Yu *et al.*[43]. The analyzed system consists of new classes only. The set of metrics used is largely based on C&K metrics [9]. The authors evaluate univariate and multivariate models. The best univariate model is based on the Number of Methods per Class metric ($R^2 = 0.423$). The results of univariate regression are used to select metrics for multivariate regression. For the metric to be selected, the univariate regression model based on it has to be significant (t-test) as well as it has to account for a large proportion of variability of the predicted value. However, in practice, the authors only reject the variables from the insignificant models. Finally, six different metrics are included in the proposed regression model, i.e., Number of Methods per Class, Coupling, Response for Class, Lack of Cohesion, Depth of Inheritance, and Number of Children. The R^2 statistic of this model is 0.597. The authors also show the model based on all ten metrics they collected. This model has the R^2 value equal to 0.603.

Cartwright and Shepperd [7] present a study in which they predict faults in object oriented system. The metric suite they use consists of some of the object-oriented C&K metrics (Depth of Inheritance, and Number of Children), some code metrics, and some metrics that are characteristic for the development method employed (Shlaer-Mellor). The authors obtain very high prediction accuracy. Their best univariate linear model is based on the number of events in the class and has $R^2 = 0.876$. The authors show that the accuracy of the model can be increased by adding a variable indicating if the class inherits from some other class ($R^2 = 0.897$.).

Unlike our study, the studies described above do not focus specifically on modified code. However, they are very good examples of how fault prediction models are build as well as what kind of data fit can be expected from them.

There are also studies that attempt to predict faults in modified systems. Nagappan and Ball [29] evaluated the applicability of relative code churn measures to predict the fault densities of software units. As relative code churn measures they understand the amount of code change normalized by the size of the code unit the change was introduced to. Their study was based on the code churn between Windows Server 2003 and Windows Server 2003 Service Pack 1. The authors concluded that the relative code churn measure could be used as predictor of a system's fault density. Their best model achieved a data fit (R^2) of 0.821. Munson and Elbaum [28] analyzed a large software system and they also noticed that relative measures are very good predictors of the fault-proneness of modified code. The metric they evaluated was the relative complexity of modified modules. They showed that this metric was highly correlated with the fault density. Selby [37] reached a similar conclusion. He observed that the number of faults in a modified class tends to increase with the size of the modification of the class.

There are also other studies that attempt to assess the applicability of different metrics to predict faults. In most cases these are studies in which classification models were built, i.e., models that predict if there are faults in the module, not how many faults there are. From our perspective such studies are interesting, since they give an indication of metrics that are good predictors of fault-proneness. For example, El Emam *et al.* [10] observes an impact of inheritance and coupling on the fault-proneness of the class. The relation between inheritance, coupling, and probability of finding faults in the class was also identified by Briand *et al.* [5]. In [17], Gunes Koru and Tian evaluate

the applicability of complexity measures to predict faults. They concluded that there is a relation between complexity and faults, but it is not linear and therefore complexity measures are not likely to be good fault predictors when used in linear prediction models, like our ones.
When it comes to the evaluation, most classification models are evaluated against the percentage of correctly classified classes. Briand et al. [5] noticed that such an evaluation may have a low practical value. Even though the model may point to a minority of classes, these classes can potentially account for a majority of the code. The prediction models used for estimating the number of faults are usually evaluated against their "goodness of fit" to the data they were built on, i.e., using R^2 statistic. Therefore, as we see it, there is a lack of studies evaluating prediction models from the perspective of gain, in terms of cost reduction, that can be expected from applying them.

3. Methods

3.1 Metrics suite

In this study we base our prediction models on the metrics that describe the structure of the system, i.e., on code and design metrics. All metrics that we collect are summarized in Table 1. All our measurements are done at the class level. The design metrics are mostly metrics that belong to the classic set of object oriented metrics suggested by Chidamber and Kemerer (C&K metrics) [9]. The code metrics are different size metrics (e.g., the number of statements), metrics describing McCabe cyclomatic complexity (Maximum Cyclomatic Complexity) as well as metrics describing the size of modification (Change Size – the number of new and modified lines of code in the final system as compared to the previous release of the system). For each class we collect information about the number of faults that were found in the class as well as calculate the fault density.

All product measurements mentioned in this study can be obtained automatically from the code using software tools. In this study we used the Understand C++ [1] application to obtain all the design and code metrics (apart from the ChgSize quantification) from the systems' code. The ChgSize was quantified using the LOCC [39] application. The information about faults was extracted from an internal Ericsson fault reporting system. We understand that it would be highly desirable [25, 26] to reveal some information about the raw data we collected. However, since these data are highly confidential, due to our agreement with Ericsson we are not allowed to do that.

Table 1. Metrics collected in the study

Name	Variable	Description
Independent variables		
Coup	Coupling	Number of classes the class is coupled to [9, 13]
NoC	Number of Children	Number of immediate subclasses [9]
Base	Number of Base Classes	Number of immediate base classes [9]
WMC	Weighted Methods per Class	Number of methods defined locally in the class [9]
RFC	Response for Class	Number of methods in the class including inherited ones[4, 9, 31]
DIT	Depth of Inheritance Tree	Maximal depth of the class in the inheritance tree[9, 12]
LCOM	Lack of Cohesion	*"how closely the local methods are related to the local instance variables in the class"* [13]. In the study LCOM was calculated as suggested by Graham [16, 18]
Stmt	Number of statements	Number of statements in the code
StmtExe	Number of executable statements	Number of executable statements in the code
StmtDecl	Number of declarative statements	Number of declarative statements in the code
Comment	Number of comments lines	Number of lines containing comments
MaxCyc	Maximum Cyclomatic Complexity	The highest McCabe complexity of a function from the class
ChgSize	Change Size	Number of new and modified LOC (from previous release)
CtC	Ratio Comment to Code	Ratio of comment lines to code lines
Dependent variables		
Faults	Number of faults	Number of faults found in the class
FaultDensity	Fault density	Fault density of the class

3.2 *Model building*

We assume that the cost of performing fault detection is directly proportional to the size of the class. Therefore, our prediction models should identify the classes with the highest fault densities. Fault detection in such classes is the most efficient because it requires the least amount of code to be analysed to find a fault. Class analysis according to the model means that fault detection activities are performed on the classes in the order of their decreasing fault density predicted by the model. As we see it, the fault density can be predicted in two ways:

- by predicting the fault density (Faults/Stmt) – the fault density is predicted by the model.
- by predicting the number of faults (Faults) and dividing the predicted number of faults by the real class size (Stmt) – Faults are predicted by the model, while size (Stmt) is measured.

In our study we evaluate both approaches. Even though they seem to predict the same thing, the prediction accuracy, given our set of metrics and our method of building models (regression), may be different for both of them. Linear regression, which we use for building models, attempts to predict the dependent variable as linear combination of independent variables. It may turn out that, e.g., linear combination of our metrics predicts fault density much more accurately than it predicts the number of faults.

We evaluate six prediction models, three predicting the fault-density and three predicting the number of faults. The models are built using:

- single metric – a model based on the single best fault (fault-density) predictor
- selected metrics – a model based on a set of the best fault (fault-density) predictors
- all metrics – a model based on all metrics collected

To find the single and the selected metrics we use the simplest method, which is the correlation analysis. Since it turned out that our data were not normally distributed we use Spearman correlation co-efficient, which is not dependent on normality assumption [42]. As selected metrics, we choose those that are correlated to the independent metrics, i.e., with correlation coefficient values not close to 0. In the case of our dataset it turned out that the lowest correlation among the metrics from the selected metrics model was 0.29. Additionally, the correlations of our selected metrics with the dependent variables have to be significant at a 0.05 level (a standard significance level describing 5% risk of rejecting a correct hypothesis). In this way we eliminate the metrics that, due to a low correlation with the number of faults and the fault-density, can not be considered useful for building prediction models.

Our univariate models are built using linear regression. The multivariate models are built using stepwise multivariate linear regression. The univariate linear regression estimates the value of the dependant variable (i.e., the number of faults or the fault-density) as a function of one of the independent variables (i.e., code and design metrics) [36]:

$$f(x) = a + b_1x_1 \tag{1}$$

Multivariate linear regression estimates the value of the dependant variable (i.e., the number of faults or the fault-density) using linear combination of independent variables (i.e., code and design metrics) [36]:

$$f(x) = a + b_1x_1+b_2x_2+b_3x_3+.....+b_kx_k \tag{2}$$

Stepwise regression is one of the methods that attempt to build a model on the minimal set of variables that explain the variance of the dependant variable. Other methods of that kind are forward (backward) regression. In these methods variables are added (removed) to the model until adding (removing) the next one does not give any benefit (does not change model's ability to predict the dependant variable) [27]. We select stepwise regression because, compared to the forward regression, it additionally excludes variables that do not contribute to the model anymore [27]. Therefore, by using stepwise regression we hope to get models based on minimal sets of variables. Stepwise regression is used on the previously defined sets of metrics (All, Selected) and the final models are built on subsets of the previously defined sets of metrics, i.e., building a model on all metrics does not mean that all metrics are used in the final model but that all metrics are used as an input to the stepwise regression.

For each model we calculate a coefficient of determination, R^2, which is the standard measure of model's "goodness-of-fit". R^2 measures the strength of the correlation between the actual and the predicted number of faults. The R^2 equation is presented below (3):

$$R^2 = 1 - \frac{\sum_{i=1}^{n} (y_i - \hat{y}_i)^2}{\sum_{i=1}^{n} (y_i - \bar{y})^2} \quad \textbf{(3)}$$

where: y_i – the actual number of faults (or the actual fault density), $\hat{y}_i$ - the predicted number of faults (or the predicted fault density), $\bar{y}$ - the average number of faults (or the average fault density).

The practical meaning of R^2 is that it describes the proportion (percentage) of variability of the predicted variable accounted by the model [20]. The higher the R^2 value is, the better the prediction model fits the data it is built on. R^2 values range from 0 to 1 [27], where 1 means the perfect model that accounts for all variability of the predicted variable (perfect prediction). R^2 equal to 0 indicates that the model is useless as a prediction model. We include R^2 for two reasons. First, it enables comparisons between our models. Second, it makes it possible to compare our results with the results obtained by other researchers, who usually quote R^2 values obtained for their models (see Section 2 for examples).

Similarly to [44], we also evaluate the significance of the entire prediction model using the F-test [27]. We select a 0.05 significance level, i.e., if the significance of the F-test has a value below 0.05 then the prediction model is significant.

All statistical operations connected with model building (i.e., correlation, stepwise regression and calculation of statistics connected with it) were performed using the statistical software package SPSS [2].

3.3 Model evaluation

To evaluate our models we need some objective measurement of the accuracy of our models. We want to know what advantage can be expected from using our models as compared to not using any model at all. We also want to know how far our models are from the theoretical best model. A good prediction model must also give good results when it is applied to the data other than the one it was built on.

To measure the objective "goodness" of our models we introduce three reference models:

- *Random model* – this model describes a completely random search for faults. The results obtained by this model are, on average, the results we could expect when no model is used and the order in which the classes are analyzed is random.
- *Best model* – this is a theoretical model that makes the right choices about which classes to analyze. In this model the classes are selected according to their actual fault density. According to our criteria, it is impossible to do better than that.
- *Size model* – a common (mis)conception [11] is that bigger classes tend to have more faults and higher fault densities. Therefore, we introduce a model in which the classes were analyzed based on their size (bigger classes are analyzed earlier).

A comparison of our models with the Random model gives us an indication if following our model is better than not following any model at all. The Best model gives us an indication of how good a model can get, and how far we are from being perfect. The Size model might often be encountered in real life situations because of its simplicity, as well as because many models suggested in literature actually tend to correlate with size [11]. By including this model we can evaluate it against our criteria of efficiency improvement as well as compare our models with it.

To check if our models are good prediction models, i.e., if they can be successfully applied to different projects, we build our models based on data from one of the projects only (System A1) and we apply them to:

- Project A1, on which the models are built
- Project A2, which is a different (next) release of Project A1
- Project B, which is a completely different project

By comparing how well our models work in Project A1 and Project A2 we get an indication if they are stable across different releases of the same system. By comparing how well the models work in Project A and Project B we get an indication if they are stable across different systems. The stability is required because a prediction model is normally used to predict faults in projects/releases other than those it was built on.

In order to compare the models' performance both within and between systems we use three complementary comparison methods for assessing model "goodness". Generally, the "goodness" of the model is measured by the amount of code necessary to analyze in order to detect a certain number of faults, i.e., a model is better if by following it we are able to detect more faults by analyzing the same amount of code compared to another model.

Our first comparison method is a diagram plotting the percentage of faults detected against the percentage of code that has to be analyzed to detect them. On every diagram we include our reference models (Random, Size, and Best model). By comparing how well our models do in relation to the Random model and to the Best model we are able to assess how good the models are and compare their performance in different systems.

The second method attempts to perform a quantification of the model's "goodness". Our *Gain* metric quantifies the ratio of an improvement offered by our model over the Random model to the theoretical maximum improvement possible. The calculation steps for the *Gain* metric are presented in Figure 1. Eq.1 in Figure 1 presents the way in which the Gain metric is calculated. In Eq.1 *IOR* stands for *Improvement Over Random*. The IOR_{Model} measure quantifies the overall improvement over the Random model that is offered by *Model*. On our diagrams, on which we plot the percentage of faults detected against the percentage of code that has to be analyzed to detect them, such an improvement over the Random model corresponds to the size of area between the Random model and *Model* (see Figure 2).

Figure 1. Calculation of the *Gain* metric. *Model(i)* is the percentage of faults found if analyzing the *i*-th class according to the *Model*, *Random*(*Code(Model,i))* is the expected percentage of faults detected if analyzing the same amount of code as in case of *Model(i)* but not following any model at all, *n* is the number of classes. The details regarding calculation steps can be found in Section 3.3.

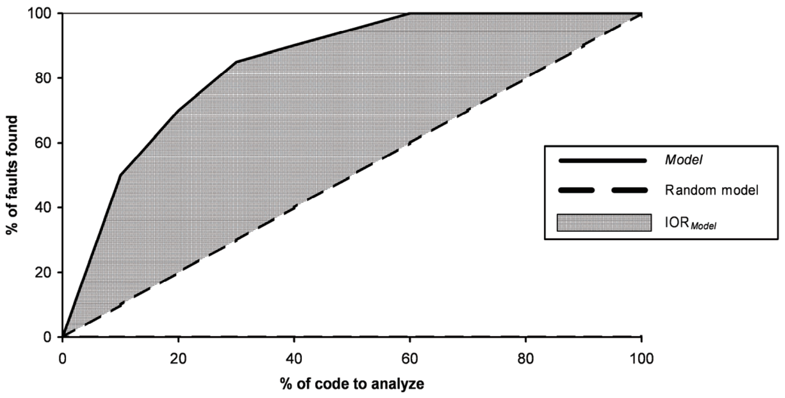
Eq.1: $$Gain_{OurModel} = \frac{IOR_{OurModel}}{IOR_{Best}}$$

Eq.2: $$IOR_{Model} = \sum_{i=1}^{n} \left(\frac{(DTR(Model,i) + DTR(Model,i-1)) * (Code(Model,i) - Code(Model,i-1))}{2} \right)$$

Eq.3: $$DTR(Model,i) = Model(i) - Random(Code(Model,i))$$

To calculate *IOR* we divide the area between the Random model and *Model* into a number of parallelograms equal to the number of classes in the system, and we sum their areas (see Eq. 2 in Figure 1). In Eq. 2, *n* is the number of classes in the system, *Code(Model,i)* is the percentage of code that must have been covered when analyzing the *i*-th class according to the *Model*. It must be remembered that in order to analyze the *i*-th class according to *Model* we must have analyzed all the classes with predicted fault densities larger than the predicted fault density of the *i*-th class, which means that *Code(Model,i)* consists not only of the size of the *i*-th class but also the sum of all sizes of classes with predicted fault densities larger than the predicted fault density of the *i*-th class. *DTR* stands for *Distance To Random*. Figure 1, Eq.3 presents the way DTR is calculated. In Eq.3, *Model(i)* is the percentage of faults detected when analyzing the *i*-th class according to the *Model*. *Random(Code(Model,i))* is the expected percentage of faults that we would detect using the Random model when analyzing the same amount of code as when analyzing the *i*-th class according to the *Model*.

Figure 2. ***Improvement Over Random (IOR)*** **for a *Model* is defined as the size of area between the *Model* and the *Random model* (checkered area on the figure)**



The *Gain* metric gives a normalized value between -1 and 1, where 1 describes the Best model and -1 describes the worst possible model. It is so, because it is impossible to do better than the Best model and it is also impossible to do worse than the worst possible model, in which all the classes are selected according to their increasing actual fault density. Therefore, $IOR_{Worst} = -IOR_{Best}$, which explains the -1 value. The Random model in this scale gets value 0, which means that all models with *Gain* lower that 0 are worse than the Random model and all those with gain over 0 are an improvement over the Random model. The *Gain* metric quantifies only the average gain from using the model. As every average, it might be missing some important details. Therefore, we use it together with previously described diagrams presenting the gain from using the model for different percentages of the code. They give more insight in how the models actually perform.

Our final, third method of assessing model "goodness" is by checking the statistical significance of the difference between the performance of a model and the performance of Random model. The analysis of the graphs described before can give some conclusions regarding the model "goodness" but based on them it is hard to say to what extent the improvement over the Random model is statistically significant. Therefore, for each model, we perform the statistical analysis is which we test the following null hypothesis:

H_0: the expected mean distance between tested model and Random model equals zero

where as distance we understand Distance To Random (DTR), defined before. Statistical tests appropriate for testing this hypothesis according to [42] are paired t-test and its non-parametric alternative Wilcoxon test (Wilcoxon Signed-Rank Test). Since our data were not normally distributed we applied the non-parametric test, i.e., Wilcoxon test.

The Wilcoxon test is performed in the following way [42]: first for every data point the distance between the Random model and the examined model is calculated. The distances are basically DTRs, as we defined them before. Absolute values of DTRs are ranked, and the sums of positive ranks (T^+) and negative ranks (T^-) are calculated. As the test statistic T of the Wilcoxon test the smaller of these two values is used, i.e., $T=\min(T^+,T^-)$. This value can be compared against tabularized values for desired significance level. For large samples it can be approximated by a normal random variable as described in [3]. SPSS, the statistical package used by us, reports the significance level for each test. Therefore, we do not need to pre-select the desired significance level for our test – we base our analysis on the highest confidence with which we can reject null hypothesis. i.e., if SPSS reports the significance of 0.05 it means that with 95% confidence we can reject the null hypothesis that our model's performance does not differ from the performance of the Random model.

4. Results

4.1 Model building

As described in Section 3.2 our models are built using the data from System A1. We begin the model building with a correlation analysis. The results of the correlation analysis are presented in Table 2. The main purpose of the correlation analysis is to identify metrics that are the best single predictors of the number of faults and the fault-density (we look for single metrics with the highest correlations with the number of faults and the fault density). From Table 2 it can be noticed that ChgSize is the best predictor for both values (correlation values in bold in Table 2). Therefore, we select this metric to build the prediction models for the number of faults and the fault-density based on one metric.

Table 2. The correlation analysis (Spearman correlation co-efficient) of the metrics collected from System A1. The correlations with a grey background are NOT significant at 0.05 significance level. The correlation of the best individual predictor of fault number and fault density is in bold.

	Base	Coup	NOC	WMC	RFC	Com ment	Stmt	Stmt Decl	Stmt Exe	Max Cyc	DIT	LCO M	CtC	Chg Size
Base	1													
Coup	0.35	1												
NOC	-0.13	0.06	1											
WMC	0.23	0.76	0.18	1										
RFC	0.63	0.68	0.07	0.82	1									
Comment	0.21	0.73	0.05	0.80	0.68	1								
Stmt	0.15	0.67	0.09	0.74	0.62	0.84	1							
StmtDecl	0.01	0.57	0.08	0.61	0.44	0.73	0.86	1						
StmtExe	0.25	0.72	0.10	0.79	0.70	0.83	0.92	0.64	1					
MaxCyc	0.21	0.65	0.09	0.65	0.56	0.73	0.83	0.51	0.93	1				
DIT	0.97	0.35	-0.13	0.22	0.61	0.20	0.11	-0.01	0.22	0.12	1			
LCOM	-0.08	0.3	0.18	0.37	0.15	0.32	0.20	0.38	0.11	0.07	-0.08	1		
CtC	0.18	-0.01	-0.06	-0.02	0.07	0.12	-0.36	-0.34	-0.24	-0.26	0.21	0.06	1	
ChgSize	0.07	0.48	0.02	0.47	0.40	0.59	0.68	0.7	0.50	0.42	0.04	0.28	-0.25	1
Faults	0.00	0.43	0.02	0.47	0.41	0.54	0.52	0.48	0.48	0.38	-0.01	0.15	-0.1	**0.6**
FaultDensity	-0.03	0.35	0.01	0.39	0.32	0.46	0.42	0.4	0.36	0.29	-0.04	0.15	-0.05	**0.53**

The second reason for performing the correlation analysis is to eliminate the metrics that can not be considered useful for building our prediction models (see Section 3.2 for details). The remaining metrics are used to build the model based on “selected metrics”. It turned out that we removed the same metrics for the model that predicts the number of faults and the model that predicts the fault-density. We have decided not to use the following metrics in “selected metrics” models:

- Base, NOC, CtC, DIT – due to their low correlation with the faults and with the fault density and due to low significance of the correlation
- LCOM – due to the low correlation with the faults and with the fault-density
- Comment – due to a unsure meaning of this metric and its correlation to size

We find the high positive correlation between Comment and Faults quite surprising. There are some possible explanations of that phenomenon, like considering the number of comments as a measure of human perceived complexity. We exclude this metric from selected metrics, because it is difficult to assure that the "commenting style" is maintained between the projects (there are no explicit guidelines concerning this in either of the analyzed projects). Therefore, it is difficult to say if prediction models based on Comments would be stable also in other products/other releases of the same product.

In the study we build six different prediction models based on the data from System A1. Their names, independent variables and outputs are summarized in Table 3. The models are built using stepwise regression. The significance of each model's coefficient is checked using the t-test. The hypothesis tested is that the coefficient could have value 0, which would imply a lack of relationship between the independent and dependant variables (and therefore would make the original model the best, but not a meaningful mathematical relation between both variables) [36]. It turned out that all coefficients were significant at the 0.05 level. The significance of the entire model is tested using the F-test. The goodness-of-fit of each model is assessed using the R^2 statistic. The actual models are presented in Table 4.

Table 3. **A summary of our models. Single metric: ChgSize. Selected metrics: Coup, WMC, RFC, Stmt, StmtDecl, StmtExe, MaxCyc, ChgSize.**

Name	Based on	Predicts
AllNumber	All metrics	Number of faults
SelectedNumber	Selected metrics	Number of faults
SingleNumber	Single metric	Number of faults
AllDensity	All metrics	Fault density
SelectedDensity	Selected metrics	Fault density
SingleDensity	Single metric	Fault density

4.2 Model evaluation

As it can be noticed in Table 4 all our models are significant according to the F-test. By looking at the R^2 values we can see that the goodness-of-fit is better for the models predicting the number of faults compared to those predicting fault-densities. Apparently, given our set of metrics, it is easier to predict the number of faults than the fault-density. As we expected (see Section 3.2) the models based on all metrics (AllNumber and AllDensity) have a better fit compared to their counterparts based on a limited number of metrics (see the R^2 values in Table 4). They may, however, suffer from the multicolinearity problem – e.g., according to the AllNumber model the number of faults increases with Comments and decreases with StmtExe, which is difficult to explain since both StmtExe and Comments are positively correlated with the number of faults (see Table 2).

Table 4. Prediction models obtained using stepwise regression based on data from System A1. R^2 describes goodness-of-fit (values closer to 1 indicate better fit), Sig. is significance level of F-test. F, R^2, and Sig. quoted for AllNumber, SelectedNumber, and SingleNumber concern the models that predict the number of faults. These models are divided by *Stmt* in Equation section in order to provide the fault density prediction.

Model	Equation	R^2	F	Sig.
AllNumber	FaultDensity = (0.004 * Comment + 0.003 * ChgSize - 0.677 * Base + 0.276 * Ctc - 0.003 * StmtDecl - 0.005 * LCOM + 0.010 * RFC - 0.001*StmtExe + 0.089)/Stmt	0.752	75.944	0.0
SelectedNumber	FaultDensity = (0.004*ChgSize+0.001*StmtExe-0.002*StmtDecl + 0.008)/Stmt	0.585	96.412	0.0
SingleNumber	FaultDensity = 0.005*ChgSize/Stmt	0.550	252.84	0.0
AllDensity	FaultDensity = 54.040*CtC + 0.115*ChgSize - 42.431*DIT - 0.096*Stmt + 3.036*Coup + 0.670*RFC - 19.685	0.479	30.997	0.0
SelectedDensity	FaultDensity=0.184*ChgSize-0.138*Stmt + 2.429*Coup + 1.057*WMC + 2.748	0.280	19.815	0.0
SingleDensity	FaultDensity=0.081*ChgSize + 12.739	0.058	12.742	0.0

Our main model evaluation is performed from the perspective of the fault detection efficiency improvement that they offer. We use each model as an indicator of the order in which the classes should be analyzed. For the models that predict the fault-density (AllDensity, SelectedDensity, SingleDensity) we order the classes according to the output of the model, so that we analyze classes with the highest predicted fault-density first. In the models that predict the number of faults (AllNumber, SelectedNumber, SingleNumber) the predicted number of faults is divided by the class size (Stmt). This partially predicted density measure is used to select the classes for analysis.

Our evaluation consists of three steps. First, for each model we plot a graph in which the percentage of faults detected is mapped to the percentage of code that has to be analyzed to detect them. The model is considered better than the other one if, by following it, we are able to detect more faults by analyzing the same amount of code. Later we calculate the *Gain* for each of our models. For details concerning the *Gain* metric see Section 3.3. Finally, we check if the differences between our models and the Random model are statistically significant.

To benchmark our models we include three reference models in the evaluation. The reference models are presented in Figure 3. By comparing the Best and the Random model in Figure 3 we can see that there is a large room for improvement that can be filled using a fault prediction model. For example, if we inspect 20% of code randomly, on average we would find 20% of faults. However, by inspecting the most fault prone 20% of code we can find 60%, 80%, or even almost 100% of faults for System A1, System A2, and System B, respectively (see Figure 3). That is three, four, and five times as much as by inspecting the code randomly. Therefore, a model that tells us which part of the code to analyze first can potentially result in cost savings and increased quality of software. In all future figures we include the Best, Random, and Size models (always in dashed line) in order to provide reference points for evaluating our models.

The second conclusion from analyzing Figure 3 is that the Size model does not help very much when it comes to increasing the efficiency of fault detection. In fact, it is either about as good as the Random model (System A1, and System B) or even worse than the Random model in the case of System A2. Neither of our cases supports the theory that the size affects fault density and that the Size model can be used to predict fault density.

When evaluating our models we start with evaluating the fault detection efficiency improvement gained by using the models that predict the number of faults (AllNumber, SelectedNumber, SingleNumber). The results are presented in Figure 4. As it can be noticed all three models present an improvement over both the Random model as well as the simple Size model. This holds true not only for System A1, on which the models are built, but this is also very clear for System A2 and System B. Our models seem to be stable and to work similarly well in all systems.

Figure 3 The reference models. The Best model describes a perfect model that makes only the right choices and selects classes in the order of their decreasing fault density. Random model represents a case in which code for inspection is picked randomly. The Size model is a model in which the biggest classes are analyzed first (see Section 3.3 for details concerning reference models).

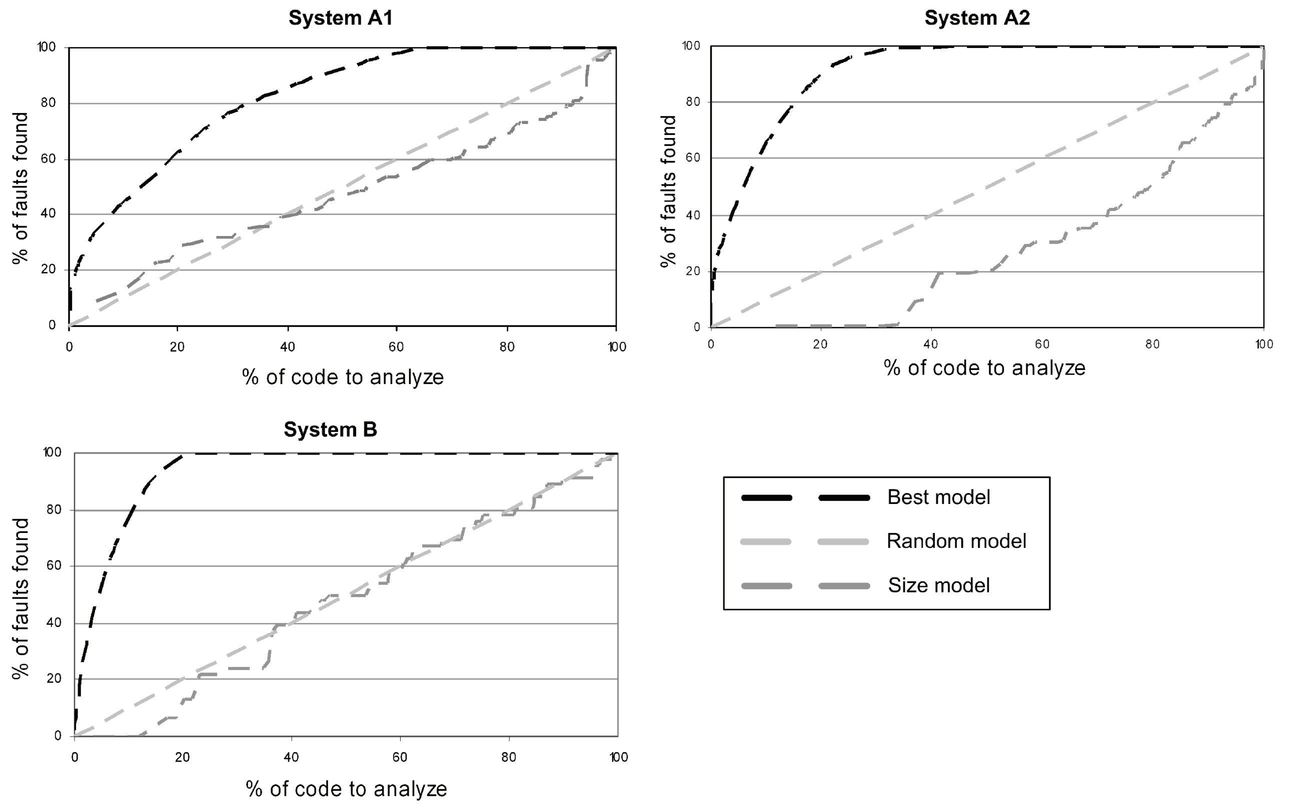
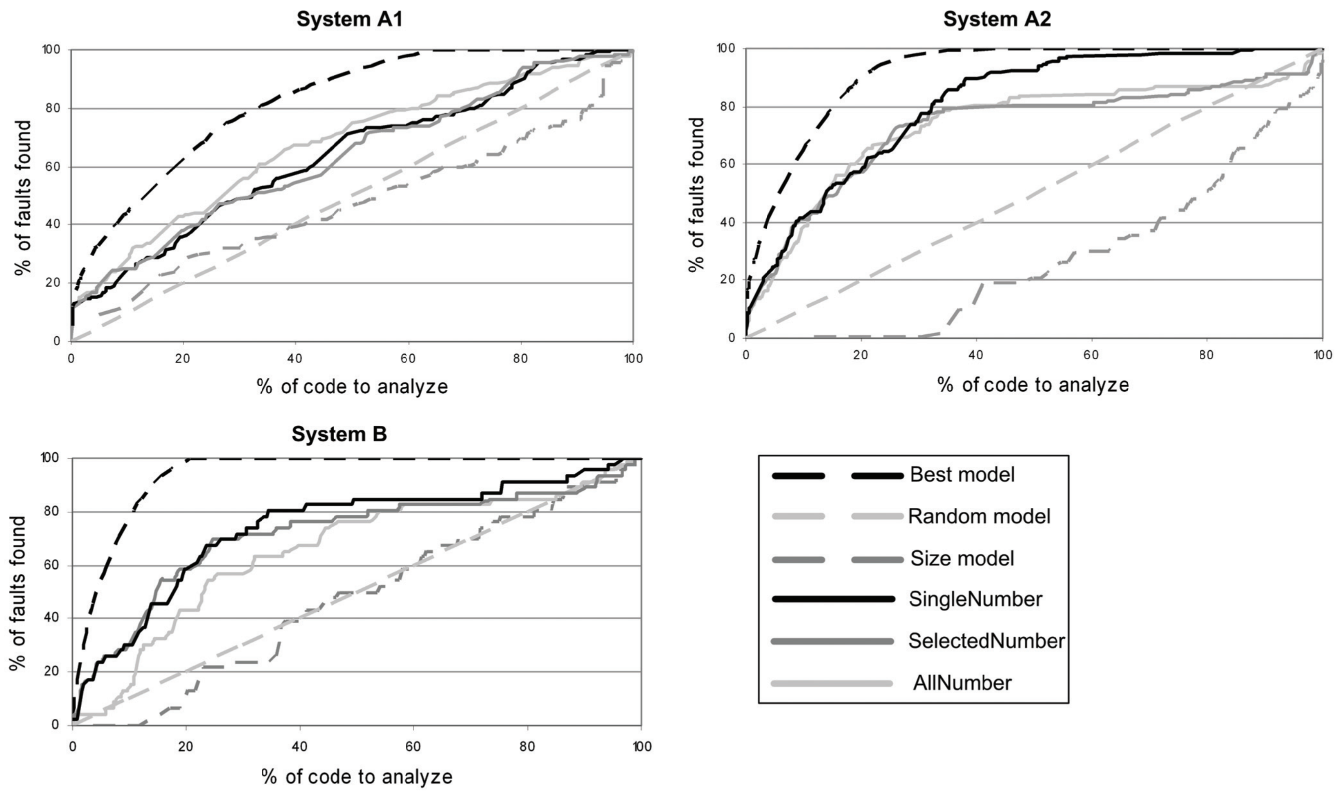


Figure 4 **The efficiency comparison of the models that predict the number of faults. The models were built on the data from System A1. Three reference models (Best, Random, and Size) are introduced to provide a baseline for comparison.**



The evaluation of the models' performance when applied to System A2 and System B indicates which models are the most promising ones as prediction models, i.e., which models are the best for predicting faults in projects other than the project they are built on. It seems that the least complex model (SingleNumber) works best. For example, both in System A1 and System B the model makes it possible to detect over 80% of faults by examining only 40% of the code. It is, on average, twice as many faults as we would detect when inspecting the code randomly and only about 10%-15% less than the possible maximum described by the Best model.

After evaluating the models that predict the number of faults we perform an evaluation of the models that predict fault density. The performance of the SingleDensity, SelectedDensity, and AllDensity models is presented in Figure 5. As in the case of models that predict the number of faults, the models that predict the fault density in most cases have a clear advantage over the Random model. Although the gain from using them is slightly but noticeably lower compared to the models that predict the number of faults (compare with Figure 4) the density prediction models are still an improvement over not using any model at all (i.e., using Random model).

When it comes to evaluating the stability, the SingleDensity and SelectedDensity models are stable in providing improvement over the Random model in all three systems. The AllDensity model works well in System A1 and System A2, but it does not in System B. It is probably an example of overfitting – the model is very much based on the unique characteristics of System A, which are present in two subsequent releases of the same system A (A1 and A2) but are not present in System B.

As a next step, we calculate the *Gain* for our models (see Section 3.3 for details concerning the *Gain* metric). The results are presented in Table 5. In Table 5 we quantify the gain for each model when applied to each system as well as provide an average gain for each model. The numbers from Table 5 support our findings from the analysis of the diagrams. From the models predicting the number of faults, SingleNumber dominates over the other two models. The SelectedDensity model is the best model from the models that predict fault density. The SingleNumber model seems to be the best model of all, since it provides, on average, 57% percent of the maximum gain possible. SelectedNumber comes second, providing on average 49% of the maximum efficiency gain. Once again the poor performance of the Size model is confirmed – in all cases it is actually worse than the Random model.

Figure 5 The efficiency comparison of the models that predict the fault density. The models were built on the data from System A1. Three reference models (Best, Random, and Size) are introduced to provide a baseline for comparison.

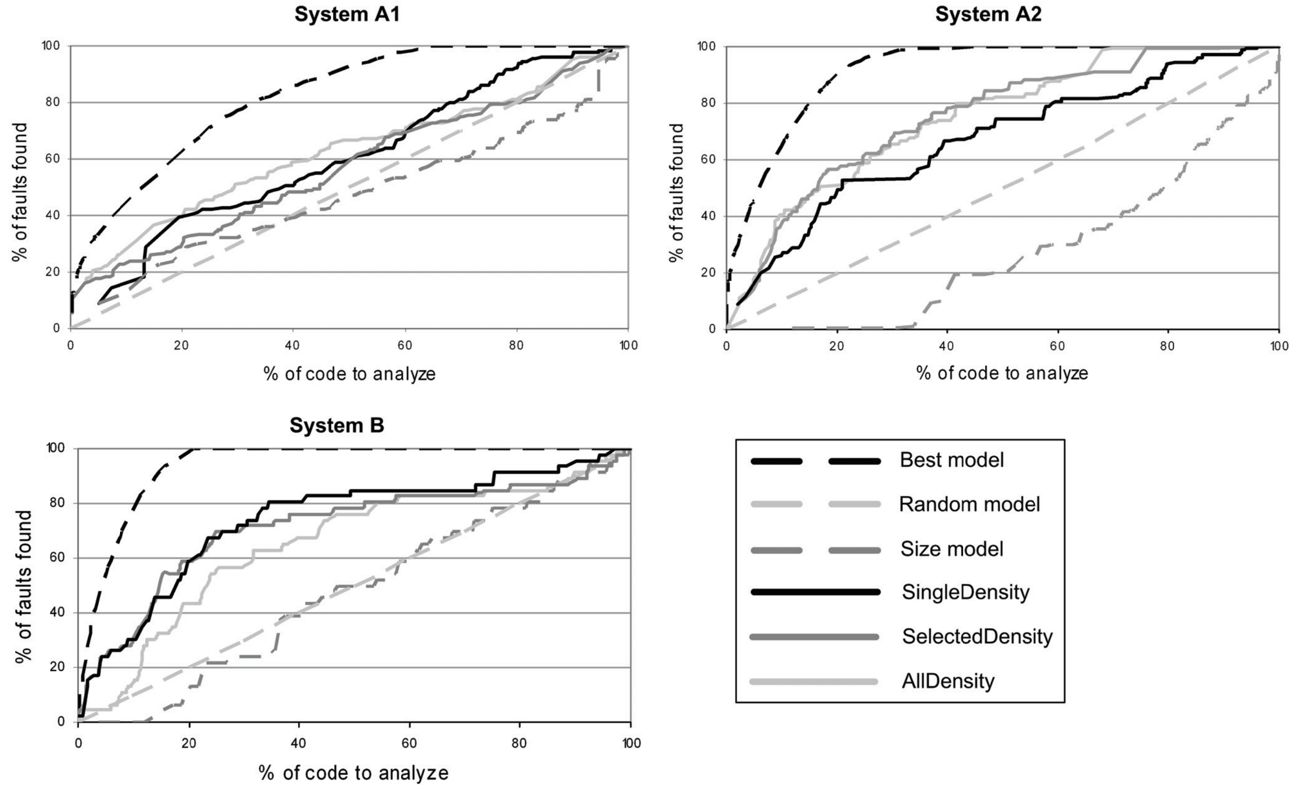


Table 5. The quantification of the Gain from using the model. Gain measures the improvement of the efficiency over the random model as a percentage of the improvement offered by the Best model. The closer the values are to 1 the better the model is. Values larger than 0 indicate that the model offers an improvement over Random model. For details concerning the Gain metric see Section 3.3.

	Single Number	Selected Number	All Number	Single Density	Selected Density	All Density	Size
System A1	0,42	0,43	0,52	0,32	0,22	0,38	-0,09
System A2	0,75	0,55	0,57	0,41	0,61	0,60	-0,54
System B	0,55	0,49	0,35	0,47	0,51	0,16	-0,06
Average gain	**0,57**	**0,49**	**0,48**	**0,40**	**0,45**	**0,38**	**-0,23**

Finally, we check if the differences between our models and the Random model are statistically significant. The hypothesis about equality of models was tested using Wilcoxon test (see Section 3.3 for details regarding the test and the interpretation of results). For all models (SingleNumber, SelectedNumber, AllNumber, SingleDensity, SelectedDensity, AllDensity, Size model, Best model) applied to all systems (System A1, System A2, System B) SPSS reported that the hypothesis about the equality of models can be rejected at 0.000 level, which practically means that the differences are significant for any conventional significance level. Additionally, the hypothesis for the Size model was rejected based on positive ranks, while for all other models it was rejected based on the negative ranks. It might be considered an indication, that the Size model is worse than the Random model (the sum of negative ranks was greater than sum of positive ranks – see Section 3.3 for details), while all other models are better, which supports our conclusions from the analysis of figures 3-5.

5. Discussion

5.1 Findings

The results obtained in our study are promising. All our models (AllNumber, SelectedNumber, SingleNumber, AllDensity, SelectedDensity, SingleDensity) represent a significant improvement compared to the Random model. It means that, when focusing fault detection efforts on a portion of the code only, more faults would be detected when using our model compared to analysing the classes in a random order. The exact value of the gain depends on the model selected, and the percentage of code analysed.

By analyzing Table 5 we can see that the best results are obtained when using the SingleNumber model. When applied to our three systems on average it produces 57% of the improvement of the Best model. The application to System A2 brings 75% of the maximum possible improvement. Application to System B brings 55% of the maximum possible improvement. The second best model, SelectedNumber, in the same situation brings 55% and 49% of the maximum possible improvement.

It is worth noticing that both our best models work well when relatively small percentages of code are analysed (see Figure 4 for SingleNumber and Figure 5 for SelectedDensity). For example, when we analyze about 40% of the code, then by following our two best models we should detect about 80% of faults. This is twice as many as if we were not following any model. A good performance when analyzing small percentages of code is probably of the largest practical value. This is the practical situation in which prediction models are most useful. If we decide to inspect 80% of the code even without using any model we already have a large statistical chance of finding many faults (80% on average). Therefore, using models in such a case must lead to a smaller benefit, basically because there is a much smaller room for improvement.

Another interesting finding from Table 5 is that in case of almost all models their performance is better when they are applied to System A2 compared to their performance when they are applied to System B. This seems to be reasonable, as System A2 is the next release of System A1 on which the models were built. It might mean that models produced within one product line have the best potential accuracy. However, this does not need to be a rule –in our case too we can see that in some cases our models work better in System B than in System A2, e.g., SingleDensity. The models that do exceptionally bad when applied to System B are the models based on all metrics (AllNumber, and AllDensity). An explanation might be that such models tend to overfit the dataset they were built on and therefore lack generality. AllNumber, and AllDensity work reasonably well in the case of System A2 but significantly worse in System B. This is the most apparent in case of the AllDensity model, which provides 60% of the maximum improvement in System A2 and only 16% in the case of System B. That would suggest that the models based on large number of metrics (i.e., AllDensity, AllNumber) are tightly fit to the unique characteristics of System A, which are present in System A1 and A2 but are not present in System B. Therefore, it seems that models based on a smaller number of metrics have better potential for stability and transferability to systems other than the system they were built on.

One more general finding from our study is that for modified code the class size is not a good predictor of fault density. This can be observed in Table 5, where the application of the Size model brings bad results in all our systems. In all cases the Size model is, on average, even worse than the Random model. Table 5, however, only presents average values. It might be that the Size model works well when a small percentage of code is analyzed and becomes really bad afterwards. Such a situation would indicate some applicability of the Size model. However, by analyzing Figure 3 we clearly see that it is not the case. In neither of our systems the Size model is significantly better than the Random model for small percentage of the code (only in System A1 it is slightly better for the first 30% of the code, but the improvement is not large).

Our results also support findings of other researchers [29, 37] that considered relative modification measures (i.e., the size of modification divided by the size of a code unit) as the best for predicting fault densities of modified classes. Our most successful model, SingleNumber, is based on such a relative modification measure.

Another general finding is that our dataset supports the Pareto principle (majority of faults are accumulated in a minority of code) for modified code. It is, however, difficult to pin-point *which* Pareto principle it exactly supports. It seems that System A1 follows the 60/20 rule stating that 60% of the faults can be found in 20% of the code. System A2 is closer to the classical 80/20 rule, while System B actually supports the extreme 80/10 rule.

5.2 Validity

As suggested in [42] we distinguish between four types of validity: internal, external, construct and conclusion validity.

The *internal validity* “concerns the causal effect, if the measured effect is due to changes caused by the researcher or due to some other unknown cause” [19]. Since our study is mostly based on correlations, by definition we can not claim the causal relationship between our dependant and independent variables. However, it is also not our ambition to claim that. There can be (and probably is) an underlying third factor that demonstrates itself in both dependent and independent variables and therefore it is possible to predict one of them using another. Because of that, by finding correlations we are able to build a useful prediction model.

The *external validity* concerns the possibility of generalising the findings. The study was performed on two systems, which are representative for systems of their class (i.e., telecommunication systems). The systems are rather large (up to 600 KLOC). In order to increase the external validity we have evaluated the models using the data different from the data used to build the models. One threat to external validity can be that all systems used in this study are telecommunication systems and that they were produced in the same company, which may make them somewhat similar. In the future we plan to evaluate our models in other kinds of systems developed by other companies.

The *construct validity* ”reflects our ability to measure what we are interested in measuring” [19]. One thing that may be worth discussing is the assumption that an effort connected with the fault detection activities is proportional to the size of the class. Many other studies consider the cost of detecting faults in the class to be a fixed value and therefore evaluate models only by how well they detect faults. We believe that the size of a class is a better cost indicator. At first we also considered the size of a change as a possible effort estimation metric. It is, however, not enough to analyse only the modified code, since the modification can violate some more general class assumption and result in fault in a part of the class that was not modified. Therefore, we selected size of the class for estimating analysis effort.
The *conclusion validity* concerns the correctness of conclusions we have made. When discussing conclusion validity we want to assess to what extent our conclusions are believable. The conclusion validity is mostly interested in checking if there is a correct relationship (i.e., statistically significant) between the variables. Therefore, where possible, we have presented the statistical significance of our findings.

6. Conclusions

The goal of this study was to build prediction models that would increase the efficiency of fault detection in modified code. We have built a number of models based on data collected from one release of a large telecommunication system. The objective of the model was to predict fault density in the classes. The models were evaluated using the next release of the system on which the models were built, as well as another large telecommunication system. The evaluation was performed against three reference models: a model based on random selection of the classes for analysis, the theoretical best model, and a simple model based on the size of the class.

We have found that our models provide a stable improvement compared to both the random and the size-based models. Our models are able to provide, on average, 38% to 57% of the maximal theoretical improvement in fault detection efficiency. The difference in performance of our models as compared to the random model was shown to be statistically significant.

As the most promising, we have found a model that predicts the number of faults based on the number of new and modified lines of code. The output of this model is divided by the class size to obtain the fault density. This model made it possible to achieve 75% of the maximum possible improvement when applied to the next release of the system on which it was built. When applied to a completely different system it achieved 55% of the maximum improvement. In both cases, these were the best results obtained for respective systems by any of our models.

The second most promising model was the one that predicted the number of faults based on the number of new and modified lines of code, the number of declarative and the number of executable statements in the class. This model made it possible to achieve 55% of the maximum possible improvement when applied to the next release of the system on which it was built, and 49% when applied to a different system.

We have also found yet another indication that models that consist of a small number of metrics that are highly correlated to faults tend to behave better when applied to a new dataset, as compared to models which use a large number of metrics. Models that use many metrics tend to overfit the dataset on which they were built, which makes them less stable when applied to other datasets.

In this study we have also managed to find empirical evidence for a number of popular hypotheses concerning faults. Our findings support the findings of those researchers that consider the relative size of modification as the best fault density predictor in modified code. Our datasets also comply with the Pareto principle. We have found an evidence of the 60/20 rule (60% of the faults can be found in 20% of the code), but also the 80/20 and even the 80/10 rule. Another finding concerns the applicability of the size metric to predict the fault density. We have shown that for modified classes the class size is a poor predictor of class fault-density.

7. Acknowledgments

The authors would like to thank Ericsson for providing us with the data for the study and The Collaborative Software Development Laboratory, University of Hawaii, USA (http://csdl.ics.hawaii.edu/) for the LOCC application.

This work was partly funded by The Knowledge Foundation in Sweden under a research grant for the project "Blekinge - Engineering Software Qualities (BESQ)" (http://www.bth.se/besq).

8. References

[1] *Understand C++ homepage*, in *http://www.scitools.com/ucpp.html*. (2005). Scientific Toolworks Inc.

[2] *SPSS package homepage*, in *http://www.spss.com*. (2006). SPSS Inc.

[3] A.D. Aczel and J. Sounderpandian, Complete business statistics, McGraw-Hill, Boston, Mass., (2006).

[4] B. Boehm and V.R. Basili, Software Defect Reduction Top 10 List. *Computer*, 34 (2001), 135-137.

[5] L.C. Briand, J. Wust, J.W. Daly, and D.V. Porter, Exploring the relationship between design measures and software quality in object-oriented systems. *The Journal of Systems and Software*, 51 (2000), 245-273.

[6] L.C. Briand, J. Wust, S.V. Ikonomovski, and L. H., Investigating quality factors in object-oriented designs: an industrial case study. *Proc. of the 1999 Int'l Conf. on Software Eng.*, (1999), 345-354.

[7] M. Cartwright and M. Shepperd, An empirical investigation of an object-oriented software system. *IEEE Transactions on Software Engineering*, 26 (2000), 786-796.

[8] S.R. Chidamber, D.P. Darcy, and C.F. Kemerer, Managerial use of metrics for object-oriented software: an exploratory analysis. *IEEE Transactions on Software Engineering*, 24 (1998), 629-639.

[9] S.R. Chidamber and C.F. Kemerer, A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20 (1994), 476-494.

[10] K. El Emam, W.L. Melo, and J.C. Machado, The prediction of faulty classes using object-oriented design metrics. *The Journal of Systems and Software*, 56 (2001), 63-75.

[11] N. Fenton and M. Neil, A critique of software defect prediction models. *IEEE Transactions on Software Engineering*, 25 (1999), 675-689.

[12] N. Fenton and N. Ohlsson, Quantitative analysis of faults and failures in a complex software system. *IEEE Transactions on Software Engineering*, 26 (2000), 797-814.

[13] N. Fenton and S.L. Pfleeger, Software metrics: a rigorous and practical approach, PWS, London; Boston, (1997).

[14] F. Fioravanti and P. Nesi, A study on fault-proneness detection of object-oriented systems. *Fifth European Conference on Software Maintenance and Reengineering*, (2001), 121-130.

[15] S. Gascoyne, Productivity improvements in software testing with test automation. *Electronic Engineering*, 72 (2000), 65-67.

[16] I. Graham, Migrating to object technology, Addison-Wesley Pub. Co., Wokingham, England; Reading, Mass., (1995).

[17] A. Gunes Koru and J. Tian, An empirical comparison and characterization of high defect and high complexity modules. *Journal of Systems and Software*, 67 (2003), 153-163.

[18] B. Henderson-Sellers, L.L. Constantine, and I.M. Graham, Coupling and cohesion (towards a valid metrics suite for object-oriented analysis and design). *Object Oriented Systems*, 3 (1996), 143-158.

[19] M. Host, B. Regnell, and C. Wohlin, Using Students as Subjects-A Comparative Study of Students and Professionals in Lead-Time Impact Assessment. *Empirical Software Engineering*, 5 (2000), 201-214.

[20] G. Keppel, Design and analysis: a researcher's handbook, Prentice Hall, Upper Saddle River, N.J., (2004).

[21] T.M. Khoshgoftaar, E.B. Allen, and J. Deng, Controlling overfitting in software quality models: experiments with regression trees and classification. *Proc. of The 17th International Software Metrics Symposium*, (2000), 190-198.

[22] T.M. Khoshgoftaar, E.B. Allen, and D. Jianyu, Using regression trees to classify fault-prone software modules. *IEEE Transactions on Reliability*, 51 (2002), 455-462.

[23] T.M. Khoshgoftaar, E.B. Allen, W.D. Jones, and J.P. Hudepohl, Accuracy of software quality models over multiple releases. *Annals of Software Engineering*, 9 (2000), 103-116.

[24] T.M. Khoshgoftaar and N. Seliya, Fault Prediction Modeling for Software Quality Estimation: Comparing Commonly Used Techniques. *Empirical Software Engineering*, 8 (2003), 255-283.

[25] B. Kitchenham, L. Pickard, and S.L. Pfleeger, Case studies for method and tool evaluation. *IEEE Software*, 12 (1995), 52-62.

[26] B.A. Kitchenham, S.L. Pfleeger, L.M. Pickard, P.W. Jones, D.C. Hoaglin, K. El Emam, and J. Rosenberg, Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on Software Engineering*, 28 (2002), 721.

[27] J.S. Milton and J.C. Arnold, Introduction to probability and statistics: principles and applications for engineering and the computing sciences, McGraw-Hill, Boston, (2003).

[28] J.C. Munson and S.G. Elbaum, Code churn: a measure for estimating the impact of code change. *Proceedings of the International Conference on Software Maintenance*, (1998), 24-31.

[29] N. Nagappan and T. Ball, Use of relative code churn measures to predict system defect density. *Proceedings of the 27th International Conference on Software Engineering ICSE 2005.*, (2005), 284-292.

[30] A.P. Nikora and J.C. Munson, Developing fault predictors for evolving software systems. *Proc. of The Ninth International Software Metrics Symposium*, (2003), 338-349.

[31] N. Ohlsson, A.C. Eriksson, and M. Helander, Early Risk-Management by Identification of Fault-prone Modules. *Empirical Software Engineering*, 2 (1997), 166-173.

[32] N. Ohlsson, M. Zhao, and M. Helander, Application of multivariate analysis for software fault prediction. *Software Quality Journal*, 7 (1998), 51-66.

[33] T.J. Ostrand, E.J. Weyuker, and R.M. Bell, Predicting the Location and Number of Faults in Large Software Systems. *IEEE Transactions on Software Engineering*, 31 (2005), 340-355.

[34] M. Pighin and A. Marzona, An empirical analysis of fault persistence through software releases. *Proceedings of the International Symposium on Empirical Software Engineering*, (2003), 206-212.

[35] M. Pighin and A. Marzona, Reducing Corrective Maintenance Effort Considering Module's History. *Proc. of Ninth European Conference on Software Maintenance and Reengineering*, (2005), 232-235.

[36] D.G. Rees, Essential statistics, Chapman & Hall, London; New York, (1995).

[37] R.W. Selby, Empirically based analysis of failures in software systems. *IEEE Transactions on Reliability*, 39 (1990), 444-454.

[38] I. Sommerville, Software engineering, Addison-Wesley, Boston, Mass., (2004).

[39] U.o.H. The Collaborative Software Development Laboratory, USA, *LOCC Project Homepage, http://csdl.ics.hawaii.edu/Tools/LOCC/*. (2005), The Collaborative Software Development Laboratory, University of Hawaii, USA. The Collaborative Software Development Laboratory, University of Hawaii, USA.

[40] P. Tomaszewski, J. Håkansson, L. Lundberg, and H. Grahn, The Accuracy of Fault Prediction in Modified Code – Statistical Model vs. Expert Estimation. *Proceedings of the 13th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems*, (2006), 334-343.

[41] P. Tomaszewski, L. Lundberg, and H. Grahn, Increasing the Efficiency of Fault Detection in Modified Code, *Asian Pacific Software Engineering Conference, APSEC*, Taipei, Taiwan, (2005), 421-430.

[42] C. Wohlin, P. Runeson, M. Höst, M. Ohlsson, B. Regnell, and A. Wesslen, Experimentation in software engineering: an introduction, Kluwer, Boston, (2000).

[43] P. Yu, T. Systa, and H. Muller, Predicting fault-proneness using OO metrics. An industrial case study. *Proc. of The Sixth European Conference on Software Maintenance and Reengineering*, (2002), 99-107.

[44] M. Zhao, C. Wohlin, N. Ohlsson, and M. Xie, A comparison between software design and code metrics for the prediction of software fault content. *Information and Software Technology*, 40 (1998), 801-809.

Paper VI

A Method for an Accurate Early Prediction of Faults in Modified Classes

Piotr Tomaszewski, Håkan Grahn, Lars Lundberg

To appear in the Proceedings of 22nd IEEE International Conference on Software Maintenance, September 2006, Philadelphia, USA

Abstract

In this paper we suggest and evaluate a method for predicting fault densities in modified classes early in the development process, i.e., before the modifications are implemented. We start by establishing methods that according to literature are considered the best for predicting fault densities of modified classes. We find that these methods can not be used until the system is implemented. We suggest our own methods, which are based on the same concept as the methods suggested in the literature, with the difference that our methods are applicable before the coding has started. We evaluate our methods using three large telecommunication systems produced by Ericsson. We find that our methods provide predictions that are of similar quality to the predictions based on metrics available after the code is implemented. Our predictions are, however, available much earlier in the development process. Therefore, they enable better planning of efficient fault prevention and fault detection activities.

1. Introduction

A majority of software systems evolve during their lifetime. This system evolution causes many changes to be introduced in the original source code. Such code modifications are an important source of faults [9, 13, 20, 21]. It is widely known that faults are one of the major cost drivers in software development projects. Activities connected with fault handling account for a significant part of the project budget, e.g., in the study reported in [4] 45% of the project resources were devoted to testing and simulation. Therefore, any method that reduces the cost associated with faults handling is likely to bring significant project cost savings.

The fact that about 60%-80% of the faults can be found in about 20% of the code modules [1, 11] and that about half of the code modules are usually defect free [1] shows that there is a potential for savings if we manage to focus our fault handling efforts on the portion of the code that actually contains faults. A popular method for identifying fault-prone code is using a fault prediction model (e.g., [6, 11, 13-15, 22]). If we assume that the cost of finding faults in a class is proportional to the size of the class (like in [2, 3]) then, by selecting classes with the highest fault densities, such a prediction model increases the *fault detection efficiency* (i.e., the number of faults found per amount of code analyzed). As a result, more faults are removed within a given budget. Therefore, in this study we build models that predict fault density.

Fault prediction models are usually based on different characteristics of the software, e.g., design or code metrics (e.g., [6, 22]). Some of those metrics are available only after the system is implemented, e.g., the number of lines of code or McCabe complexity [7]. There are also metrics that are available before the coding has started. For example, many design metrics, like the number of methods or coupling [5], can be calculated from the design documentation. Prediction models based on such design metrics are able to identify fault-prone classes even before these classes are actually modified. Being able to identify the most fault-prone classes so early in the development process makes it possible to apply preventive measures to such classes. For example, they can be assigned to more experienced developers or an increased number of code reviews/inspections can be planned for such classes.

There are a lot of studies that attempt to predict faults in the modified code units [8, 10, 17, 19-21]. One general conclusion from these studies is that the most promising indicator of fault density of a modified code unit is the relative size of the modification of this code unit, i.e., the size of the modification divided by the size of the whole code unit (see Section 2 for details concerning these studies).

In this paper we apply the idea of a relative modification size to the metrics that are available before the system is implemented. We define a number of metrics, available at design time, that approximate the relative size of the modification. We evaluate their ability to predict fault densities of classes before these classes are implemented. We show that our metrics are able to predict fault densities of classes with accuracy similar to the accuracy of a prediction based on metrics that are available after the code is implemented.

Our evaluation is based on data describing three releases of two telecommunication systems developed by Ericsson. These are large systems (about 1000 classes, 500 KLOC each) that are mission-critical for mobile network operators. Because of that, they undergo extensive and therefore expensive quality assurance before they are released to the market. The systems are mature and have been available on the market for over six years.

The rest of the paper is structured as follows: in Section 2 we present work that has been done by others in the area of fault prediction in modified code. Section 3 describes the metrics we have defined to predict fault densities in modified classes. In Section 4 we present our evaluation method. Section 5 presents the results of the evaluation. In Section 6 we discuss our findings. In the last section (Section 7) we present the most important conclusions from our study.

2. Related work

As we indicated in the introduction there is a lot of research that aims at predicting faults in evolving systems. Nagappan and Ball [10] evaluated the applicability of relative code churn measures to predict the fault densities of software units. As relative code churn measures they understand the amount of code change normalized by the size of the code unit the change was introduced to. Their study was based on the code churn between Windows Server 2003 and Windows Server 2003 Service Pack 1. The authors concluded that the relative code churn measure could be used as predictor of a system's fault density. The measures described in [10] are typical code metrics. To calculate them the system must be implemented, which limits the usage of the prediction models to after the system is implemented.

Munson and Elbaum [9], analyzed large software system and they also noticed that relative measures are very good predictors of the fault-proneness of modified code. The metric they evaluated was the relative complexity of modified modules. They showed that this metric was highly correlated with the fault density.

Selby [17] reached a similar conclusion. He observed that the number of faults in a modified class tends to increase with the size of the modification of the class. The information about the modification of a file was also considered very useful by Ostrand at al. [12]. They noticed that modified files are very fault-prone – more fault prone than new files.

We also performed studies (Paper V, Paper VII) in which we built models that predict fault densities in modified classes. We found that the most promising metric for estimating the number of faults in the modified code was the size of the modification, which we calculated as a number of new and modified lines of code in the class. As a consequence, the best fault density prediction metric was the relative modification size, obtained by dividing the size of the modification by the size of the class.

In all studies described above the faults are predicted in modified code, but only after the system is implemented. There are also studies that report promising results when it comes to predicting faults before the implementation has started. For example, Zhao at al. [22] compared the accuracy of fault prediction using design metrics with the accuracy of fault prediction using code metrics. The authors concluded that the results obtained from models based on design metrics are even more accurate than the results obtained using code metrics only. The authors, however, did not say if the modules analyzed were new or modified. Also the design metrics collected are mostly different SDL related metrics (the number of SDL diagrams, the number of task symbols in SDL descriptions, etc.), which limits their usage to systems designed using SDL.

There are studies that evaluate the applicability of other metric suits to predict faults. For example, Yu *et al.* [15] evaluated the applicability of the most common object-oriented metrics for predicting the number of faults. The authors obtained rather promising results but their study was based on new classes only.

To check if object-oriented metrics are also applicable for predicting faults in modified code we performed a study [20], in which we compared the accuracy of fault predictions using object oriented metrics with the accuracy of predictions using code metrics. It turned out that our results were similar when we used design or code metrics that described the characteristics of a final system. However, when we introduced the code metric describing the size of modification, it largely increased the quality of prediction using code metrics. This metric, alone, achieved higher prediction accuracy than all metrics describing the characteristics of a final system combined into one multivariate prediction model. Therefore, we concluded that to improve the quality of early (i.e., available before implementation) prediction of faults we must look for metrics that:

- describe the characteristics of the modification
- are available before the implementation is done

In this paper we suggest such metrics and we evaluate their ability to predict fault proneness of modified classes.

3. Predictor metrics

As we indicated in previous sections, our goal is to find metrics that are available at the time when the new release of the system is already designed, but not yet implemented. The metrics should describe the relative size of modification (*RelMod*), i.e., the size of the modification divided by the size of the class:

$$\text{RelMod} = \frac{\text{Size(Modification)}}{\text{Size(Class)}} \qquad (1)$$

In studies where the prediction is performed after the code is implemented, such a metric was shown to be very successful for predicting fault densities of modified files (see Section 2 for details). However, the task of obtaining such a metric is significantly simpler when the code is implemented. At that time we can simply measure Size(Modification), i.e., the number of added and changed lines of code in the class, and Size(Class), i.e., the number of code lines in the class. Both values are easily available from version control systems. However, at the design time none of these metrics are available. For that reason, they must be approximated by some other metrics.

Typically size metrics measure the length of code and therefore they are based on counting the number of some language constructs, e.g., the number of statements, the number of code lines, or the number of operands. Even though all these metrics do not measure exactly the same thing, they usually tend to be highly correlated, which makes it possible to predict one of them using another. One size metric of that kind that is available from the design documentation is the number of methods (*NoM*). This metric was shown to be a very good predictor of the final size of the system measured in the number of code lines [16]. In our study two metrics are based on the concept of counting methods:

- *NoM*– the Number of Methods in the Class, which we use as a Size(Class) metric
- *NoACM* – the number of Added or Changed Methods in the Class, which we use as a Size(Modification) metric

One can argue that one problem with using NoM as a size metric is that the average size of a method (in lines of code) may be different in different classes. Studies like [16] show, that these differences tend to average out at the project level. However, since for modified classes we actually have information about the average size of the method, we decided to check if using this information improves the accuracy of a prediction. The average size of a method can be calculated from the previous release of the system. Therefore, we introduced a new metric *ApproxSize* (approximated size of the class) which we define in the following way:

$$\text{ApproxSize} = \text{NoM}_{\text{CurRel}} \bullet \frac{\text{Size}_{\text{PrevRel}}}{\text{NoM}_{\text{PrevRel}}} \qquad (2)$$

where *CurRel* indicates that the metric concerns the release for which we perform predictions, while *PrevRel* indicates that a certain metric concerns the previous release of the system. Obviously, we use ApproxSize as Size(Class) metric.

Based on the metrics introduced above (NoM, NoACM, and ApproxSize) we defined two metrics describing the relative size of the modification.

The first one, *RelMod*$_{NoM}$, measures the modification as the number of new or modified methods in the class in relation to the number of all methods in the class:

$$\text{RelMod}_{\text{NoM}} = \frac{\text{NoACM}}{\text{NoM}} \qquad (3)$$

The second one, *RelMod*$_{ApproxSize}$, uses the ApproxSize metric to approximate the size of the class. Therefore, RelMod$_{\text{ApproxSize}}$ is defined in the following way:

$$\text{RelMod}_{\text{ApproxSize}} = \frac{\text{NoACM}}{\text{ApproxSize}} \qquad (4)$$

4. Evaluation method

The evaluation of our metrics is performed using the data collected from three releases of two large telecommunication systems developed by Ericsson. From now on, we call these systems System A1, System A2, and System B, where System A1 and System A2 are two consecutive releases of one system. As we indicated in Section 1, these systems are large, they comprise of about 1000 classes and about half a million code lines each. In the releases under study a significant amount of code was introduced as a modification of already existing classes. In System A1 44% of the code was introduced as the modifications of existing classes, in System A2 43% of the code introduced in this release was introduced in existing classes. In System B 37% of the code written in this release was written in existing classes. An interesting thing is that 78%, 60% and 62% of faults that were found in System A1, System A2, and System B, respectively, were located in modified classes. This clearly suggests that modified classes are an important source of faults.

We evaluate our metrics ($RelMod_{NoM}$,, and $\mathrm{RelMod}_{\mathrm{ApproxSize}}$) from the perspective of their applicability to predict the fault proneness of modified classes. We order classes in the order of their decreasing fault density. We evaluate the different metrics by plotting the percentage of faults that would be detected if analyzing a system according to its suggestion against the accumulated percentage of the code that would have to be analyzed. Since our prediction method is meant for modified classes in our evaluations we use only modified classes from the respective systems.

To obtain a point of reference for our evaluations, we introduce two theoretical reference models:

- *Random model* – the model describing a completely random search for faults
- *Best model* – the model that makes only the right choices about which classes to analyze first

The Random model provides a baseline for evaluating our predictions, as it describes what results, on average, we could expect if we analyzed the code not following any model at all. On average, by analyzing *n*% of code we find *n*% of faults. Therefore, the Random model looks the same for all systems. By comparing the performance of our prediction with the Random model we can see if our prediction method provides an improvement over not using any prediction method at all.

The Best model provides a boundary of how good the prediction can be. In this theoretical model the code units are selected according to their actual fault density. The Best model looks differently for different systems, because it depends on the actual distribution of faults in the system. By comparing the performance of our prediction with the Best model we can see how far our prediction is from the best possible prediction.

The models described above are theoretical models. Other studies (see Section 2 for details) indicate that the best prediction practically available can be obtained by using the actual relative size of code modification. Therefore, we additionally include this metric as a point of reference. The relative size of code modification ($RelMod_{Code}$) is defined as:

$$\mathrm{RelMod}_{\mathrm{Code}} = \frac{\mathrm{NoACLOC}}{\mathrm{NoLOC}} \qquad (5)$$

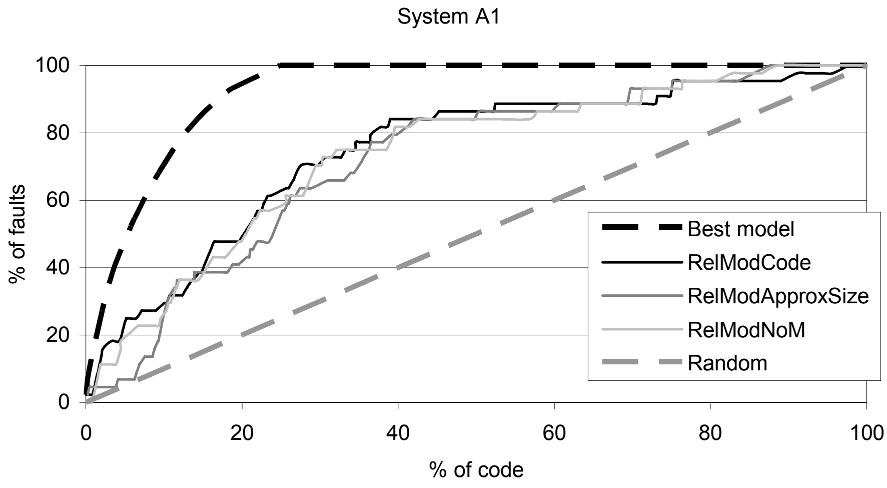
where NoACLOC is the number of added and changed lines of code in the class, while NoLOC is the total number of lines of code in the class. The reader must bear in mind that $\mathrm{RelMod}_{\mathrm{Code}}$ is available only after the code is implemented. It can be seen as the current “state-of-the-art” in prediction of fault densities in the modified classes. Therefore, it is not evaluated in our study but it is included in our evaluations as a point of reference.

5. Results

The results of the evaluation using System A1 are presented in Figure 1. As can be noticed, there is no visible difference in the prediction quality between our metrics ($\mathrm{RelMod}_{\mathrm{NoM}}$ and $\mathrm{RelMod}_{\mathrm{ApproxSize}}$) and the relative modification metric measured after the code is implemented ($\mathrm{RelMod}_{\mathrm{Code}}$). This indicates that the fault densities of the classes in System A1 could be predicted equally accurately before the system was implemented and after the system was implemented. There is no obvious difference between the performance of $\mathrm{RelMod}_{\mathrm{NoM}}$ and $\mathrm{RelMod}_{\mathrm{ApproxSize.}}$

On average, our prediction models provide about half of the maximum possible improvement over the Random model. This is not any formal quantification, but an observation based on the fact that in Figure 1 our predictions are placed more or less half way between the Random model and the Best model.

Figure 1. Evaluation of the applicability of metrics to predict the fault-densities of modified classes in System A1.



The results of evaluation using System A2 are presented in Figure 2. By analyzing Figure 2 we can see that $\text{RelMod}_{\text{Code}}$ and $\text{RelMod}_{\text{ApproxSize}}$ predict fault densities with a similar accuracy. Therefore, the best prediction available before the code is implemented gives similar results as the best prediction available after the code is implemented. The accuracy of $\text{RelMod}_{\text{NoM}}$ is actually similar to the accuracy of the two remaining prediction models, apart from between 30% and 40% of code where it is clearly worse. Similarly to the results obtained when evaluating our prediction method using data from System A1, in System A2 our predictions offer about half of the maximal possible improvement.

The results of the evaluation of our prediction methods using data collected from System B are presented in Figure 3. In Figure 3 we can see that the prediction using $\text{RelMod}_{\text{Code}}$ is more accurate than any of the two prediction methods available at the design time. In practice, however, it is visible only when between 20% and 40% of the code is considered.

In System B the prediction using $\text{RelMod}_{\text{ApproxSize}}$ seems to be more accurate compared to the prediction using $\text{RelMod}_{\text{NoM}}$, especially when low percentages of the code are considered (up to 30%). However, as in case of Systems A1 and A2, in System B the overall difference in performance between $\text{RelMod}_{\text{ApproxSize}}$ and $\text{RelMod}_{\text{NoM}}$ is not large.

Figure 2. Evaluation of the applicability of metrics to predict the fault-densities of modified classes in System A2.

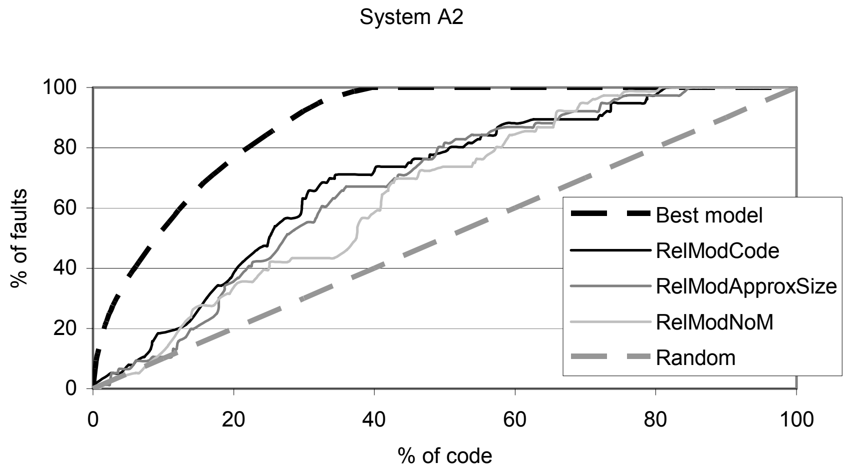
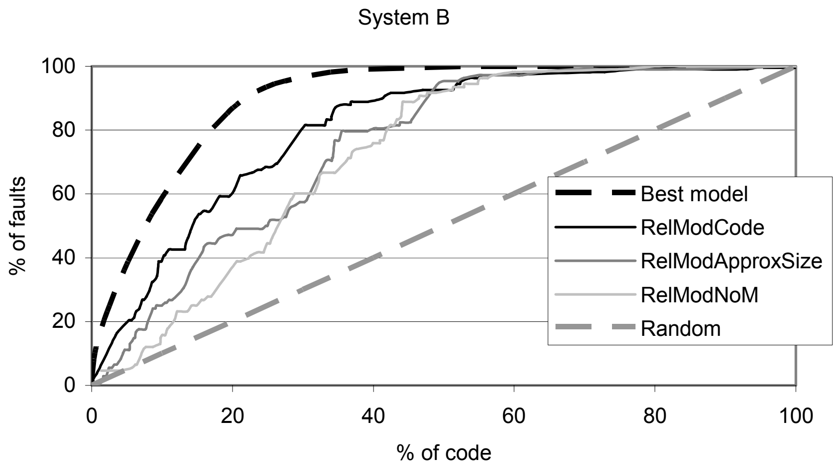


Figure 3. Evaluation of the applicability of metrics to predict the fault-densities of modified classes in System B.



Also similarly to the previous cases (i.e., System A1 and System A2) in System B the early prediction methods are stable in providing about a half of the maximum possible improvement over the Random model. The prediction using $\text{RelMod}_{\text{Code}}$ seems to be more accurate here than in the previous cases – in Figure 3 the $\text{RelMod}_{\text{Code}}$ for all percentages of the code is closer to the Best model than to the Random model.

6. Discussion

Our findings clearly show that it is possible to perform accurate predictions concerning the fault densities of modified classes at the design stage, i.e., before these classes are actually implemented. Our evaluation, in which we used three releases of large telecommunication systems, showed that in all three cases the quality of the prediction based on the data available before the implementation was comparable with the quality of the best prediction available after the code was implemented. These findings are promising, as they indicate that it is possible to obtain the information that can be used for planning fault detection and fault prevention activities at the time when this information is most needed, i.e., early in the development process.
The results indicate that our method of approximating the size of code modification by using the information about the number of new and modified methods in the class works well and is accurate enough for making predictions. Also both our methods for approximating the final size of the class are accurate enough. It seems, however, that the method, in which we use the information about the size of the class from before the modification is slightly more accurate compared to the method that takes only the number of methods in the class into account. It can be observed because the predictions obtained using this approximation (i.e., $\text{RelMod}_{\text{ApproxSize}}$) are very similar to the predictions using the actual size of the class after modification (i.e., $\text{RelMod}_{\text{Code}}$).

One reason for the higher accuracy of predictions based on the number of methods and the size of the class from previous release of the system as compared to only using the number of methods might be that the spread of sizes of methods seems to be smaller within the classes than between classes. This can be explained by the fact that there is usually one person responsible for implementing a class and, therefore, this person's "programming style" may make the methods similar in size. This is, however, only a hypothesis, which we have not evaluated in this study.

On the other hand, by looking at figures 1-3 we see that the actual difference between $\text{RelMod}_{\text{ApproxSize}}$ and $\text{RelMod}_{\text{NoM}}$ is, in practice, very small. It would indicate that the sizes of the methods are not very different even between classes. It can mean that there are some common design practices that are followed by different designers within the company, which make their methods somewhat similar in size.

Even though, based on our evaluations, we would rather suggest using $\text{RelMod}_{\text{ApproxSize}}$, we must clearly state that using $\text{RelMod}_{\text{NoM}}$ also provides an improvement over not using any prediction method at all (i.e., following the Random model). The improvement is not much smaller compared to using $\text{RelMod}_{\text{ApproxSize}}$. The main difference, as we see it, is that $\text{RelMod}_{\text{ApproxSize}}$ seems to be more stable (see Figure 2 and Figure 3). This is, however, only our subjective judgment based on the observation of figures 1-3, not supported by any formal statistical analysis.

One can argue that one of the greatest advantages of fault prediction models based on code metrics, as well as those based on some design metrics, is that the measurements necessary for predictions can be obtained automatically. For example, for our $\text{RelMod}_{\text{Code}}$ it is possible to write an application that will get as an input the code from current and previous releases of the system, and as output will produce the prediction. The information about class sizes and modification sizes can be measured by a software tool, e.g., LOCC [18], or can be obtained from a version control system.

Such a full automation in case of our prediction method will be hard to achieve. Some things, like class size in the previous release of a system or the number of functions in the planned release are relatively easy to obtain automatically. Class size in the previous release of the system can be measured using some code measuring tool. If the design of the system is done using, e.g., UML modeling language, it is also relatively easy to extract the information about the number of methods in the class in the designed system. We are, however, not aware of any method for automatically obtaining the information regarding the number of new and modified methods in the class at design time. Therefore, if such prediction method is to be implemented, the company must introduce a process, in which each designer manually quantifies the number of methods to be modified and added to a class when planning the modification of this class. This should be a neither difficult nor expensive process. It must, however, be used rigorously for our prediction method to work.

One validity threat to our study is that the systems on which we evaluate our models come from the same company (i.e., Ericsson) and the same application domain (i.e., telecommunications). As we indicated before, it is possible that within this particular company there is some kind of “style guide” that e.g., makes the differences between the method sizes small and therefore makes the number of methods an accurate predictor of the size measured in code lines. We investigated this factor and, to our knowledge, there is no such guide stated explicitly. It is however, still possible that there is some implicit “programming style” within the company that is followed by the designers. This could potentially limit the applicability of our findings to this company only. Therefore, to further evaluate the models, an evaluation using data describing systems developed in some other companies and for different application domains would be recommended.

7. Conclusions

The goal with this paper is to suggest and evaluate a method for predicting fault densities in modified classes early in the development process. In this study we focus on predicting fault densities of classes before they are actually modified. Access to information about the fault-proneness of the classes before they are modified enables more efficient planning of different fault prevention and fault detection activities. For example, in order to assign more experienced developers to especially fault-prone classes, the information about fault-proneness of the classes in the system must be available before the coding actually begins.

In our study we establish the current “state-of-the-art” when it comes to predicting the fault densities of modified classes. We find that the relative size of code modification is considered as the best fault density predictor, i.e., the size of the code modification divided by the size of the class. This metric is available only after the system is implemented, so it is not applicable for the early prediction of fault-proneness.

Since the relative size of the modification is considered as the best fault density predictor for modified classes, we want metrics that approximate this measure but that are available before the coding starts. We suggest two such measures. Both of them approximate the size of modification by counting the number of added and modified methods in the modified classes. As class size metric one of them uses the number of methods in the class, while the other one also incorporates the information about the average size of the method in the previous release of a certain class.

We evaluate both our prediction methods and obtain promising results. Both our methods provide a prediction of quality similar to the quality of the prediction using the “state-of-the-art” solution that is only available after the code is implemented. It means that, by using our method, it is possible to obtain the information of similar quality much earlier in the development process.

Since the measurements necessary for our prediction can not be obtained automatically we also discuss the changes that need to be introduced to the development process in order to collect all the data we need for making our predictions. We conclude that, even though the data must be collected manually, the process of obtaining it is very simple and inexpensive. It must, however, be followed rigorously for our method to work.

8. Acknowledgments

The authors would like to thank Ericsson for providing us with the data for the study and The Collaborative Software Development Laboratory, University of Hawaii, USA (http://csdl.ics.hawaii.edu/) for the LOCC application.

This work was partly funded by The Knowledge Foundation in Sweden under a research grant for the project "Blekinge - Engineering Software Qualities (BESQ)" (http://www.bth.se/besq).

9. References

[1] B. Boehm and V.R. Basili, Software Defect Reduction Top 10 List. *Computer*, 34 (2001), 135-137.

[2] L.C. Briand, J. Wust, J.W. Daly, and D.V. Porter, Exploring the relationship between design measures and software quality in object-oriented systems. *The Journal of Systems and Software*, 51 (2000), 245-273.

[3] L.C. Briand, J. Wust, S.V. Ikonomovski, and L. H., Investigating quality factors in object-oriented designs: an industrial case study. *Proc. of the 1999 Int'l Conf. on Software Eng.*, (1999), 345-354.

[4] M. Cartwright and M. Shepperd, An empirical investigation of an object-oriented software system. *IEEE Transactions on Software Engineering*, 26 (2000), 786-796.

[5] S.R. Chidamber and C.F. Kemerer, A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20 (1994), 476-494.

[6] K. El Emam, W.L. Melo, and J.C. Machado, The prediction of faulty classes using object-oriented design metrics. *The Journal of Systems and Software*, 56 (2001), 63-75.

[7] N. Fenton and S.L. Pfleeger, Software metrics: a rigorous and practical approach, PWS, London; Boston, (1997).

[8] T.L. Graves, A.F. Karr, J.S. Marron, and H. Siy, Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*, 26 (2000), 653-661.

[9] J.C. Munson and S.G. Elbaum, Code churn: a measure for estimating the impact of code change. *Proceedings of the International Conference on Software Maintenance*, (1998), 24-31.

[10] N. Nagappan and T. Ball, Use of relative code churn measures to predict system defect density. *Proceedings of the 27th International Conference on Software Engineering ICSE 2005.*, (2005), 284-292.

[11] N. Ohlsson, A.C. Eriksson, and M. Helander, Early Risk-Management by Identification of Fault-prone Modules. *Empirical Software Engineering*, 2 (1997), 166-173.

[12] T.J. Ostrand, E.J. Weyuker, and R.M. Bell, Predicting the Location and Number of Faults in Large Software Systems. *IEEE Transactions on Software Engineering*, 31 (2005), 340-355.

[13] M. Pighin and A. Marzona, An empirical analysis of fault persistence through software releases. *Proceedings of the International Symposium on Empirical Software Engineering*, (2003), 206-212.

[14] M. Pighin and A. Marzona, Reducing Corrective Maintenance Effort Considering Module's History. *Proc. of Ninth European Conference on Software Maintenance and Reengineering*, (2005), 232-235.

[15] Y. Ping, T. Systa, and H. Muller, Predicting fault-proneness using OO metrics. An industrial case study. *Proc. of The Sixth European Conference on Software Maintenance and Reengineering*, (2002), 99-107.

[16] M. Ronchetti, G. Succi, W. Pedrycz, and B. Russo, Early estimation of software size in object-oriented environments a case study in a CMM level 3 software firm. *Information Sciences*, 176 (2006), 475-489.

[17] R.W. Selby, Empirically based analysis of failures in software systems. *IEEE Transactions on Reliability*, 39 (1990), 444-454.

[18] U.o.H. The Collaborative Software Development Laboratory, USA, *LOCC Project Homepage, http://csdl.ics.hawaii.edu/Tools/LOCC/.* (2005), The Collaborative Software Development Laboratory, University of Hawaii, USA. The Collaborative Software Development Laboratory, University of Hawaii, USA.

[19] P. Tomaszewski, J. Håkansson, L. Lundberg, and H. Grahn, The Accuracy of Fault Prediction in Modified Code – Statistical Model vs. Expert Estimation. *Proceedings of the 13th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems*, (2006), 334-343.

[20] P. Tomaszewski, L. Lundberg, and H. Grahn, The Accuracy of Early Fault Prediction in Modified Code. *Fifth Conference on Software Engineering Research and Practice in Sweden*, (2005), 57-63.

[21] P. Tomaszewski, L. Lundberg, and H. Grahn, Increasing the Efficiency of Fault Detection in Modified Code, *Asian Pacific Software Engineering Conference, APSEC*, Taipei, Taiwan, (2005), 421-430.

[22] M. Zhao, C. Wohlin, N. Ohlsson, and M. Xie, A comparison between software design and code metrics for the prediction of software fault content. *Information and Software Technology*, 40 (1998), 801-809.

Paper VII

Statistical Models vs. Expert Estimation for Fault Prediction in Modified Code – an Industrial Case Study

Piotr Tomaszewski, Jim Håkansson, Håkan Grahn, Lars Lundberg

Submitted to a journal.

Abstract

Statistical fault prediction models and expert estimations are two popular methods for deciding where to focus the fault detection efforts when the fault detection budget is limited. In this paper we present a study in which we empirically compare the accuracy of fault prediction offered by statistical prediction models with the accuracy of expert estimations. The study is performed in an industrial setting. We invited eleven experts that are involved in the development of two large telecommunication systems. Our statistical prediction models are built on historical data describing one release of one of those systems. We compare the performance of these statistical fault prediction models with the performance of our experts when predicting faults in the latest releases of both systems. We show that the statistical methods clearly outperform the expert estimations. As the main reason for the superiority of the statistical models we see their ability to cope with large datasets. This makes it possible for statistical models to perform reliable predictions for all components in the system. This also enables prediction at a more fine-grain level, e.g., at the class instead of at the component level. We show that such a prediction is better both from the theoretical and from the practical perspective.

1. Introduction

The high cost of finding and correcting faults in software projects has become one of the major cost drivers of software development. In literature, we can find many case studies, which show that the activities connected with fault detection account for a significant part of the project budget. On the other hand, software projects often face budget limitations that put stringent restrictions on extensive and expensive quality assurance. To achieve the highest possible product quality, software developers need to decide where to focus their fault detection efforts in order to detect as many faults as possible within a given budget.

If we assume that the cost of fault detection (e.g., inspection) for a code unit (e.g., a class or a component) is proportional to the size of this unit, we can see that fault detection is most efficient when it is focused on the code units with the highest *fault density*, i.e., with the largest number of faults per line of code. It is commonly known that faults are very rarely distributed evenly in software systems. Typically, the majority of faults can be found in a minority of the code units (for an overview of research concerning this issue see [10]). Therefore, when the budget is limited, fault detection should be performed on the code units in order of their decreasing fault density. To plan such fault detection, we must be able to predict the fault density of the code units.
One approach to perform fault density prediction is to build statistical fault prediction models. Such models predict fault-proneness of the code units based on their characteristics, e.g., size, complexity, etc. This approach is very popular in academia – there is a lot of research describing and evaluating such models [3, 5, 6, 8, 12, 18, 21, 28, 31]. Our own experience shows, however, that fault prediction models are less popular and not so widespread in industry.

Another approach to predict the fault-proneness of the code units are expert estimations. In this approach, human experts suggest the order in which the code units should be analyzed. The experts usually base such decisions on their experience and knowledge about the system. In contrary to the statistical prediction models, this approach seems to be very popular in industry but is not well researched.

An expert judgement is an accepted and a common way of performing estimations in many software engineering related areas [2, 15, 29]. Despite this fact, we have failed to find any report presenting a comparative evaluation of the applicability of expert judgments vs.

statistical prediction models for predicting the fault-proneness of code units. Therefore, in this study our goal is to compare the accuracy of the fault prediction made by statistical fault prediction models with the accuracy of expert estimations.

The study is industry based. As study objects we have selected two large software systems from the telecommunication domain developed at Ericsson. We denote them as System A and System B. We use one release of System A and two releases of System B (called System B1 and System B2). System B1 is used to build our statistical prediction models. The models are evaluated on System A and System B2. To perform the expert estimation we have invited six persons involved in the development of System A and five persons involved in the development of System B2.

Each system release that we examine in this study introduces a significant amount of new functionality. Typically, the new functionality is introduced either as new classes or as modifications of existing classes. In our dataset we have found that code inserted as modifications of existing classes accounts for a minority of the code introduced in each system's release. At the same time the modified classes contained a majority of the faults. Therefore, in this study we focus specifically on predicting fault density in the modified code.

The reminder of this paper is structured as follows. In Section 2 we present the work done by others in the area of fault prediction. Section 3 contains more detailed information concerning the systems and the experts in our study. In Section 4 we introduce the methods that we use in this study. In Section 5 we present the results, and in Section 6 we discuss our findings and their validity. Section 7 contains the most important conclusions from our study.

2. Related work

A lot of work has been done in the area of fault detection improvement. A large portion of this research focuses on building fault prediction models. Depending on the output (the dependant variable), these fault prediction models belong to one of the following groups [19]:

- Quality prediction models - these models attempt to quantify the quality of the code unit, e.g. by predicting the number of faults in the code unit. Examples of such models can be found in [5, 6, 21, 28, 31].

- Classification models – these models classify code units as fault-prone or not, i.e., they predict if the code unit contains faults. Examples of such models can be found in [3, 8, 12, 18].

The models often operate at different levels of the logical structure of the code. There are models that predict fault-proneness of classes [1, 4, 5, 8, 20, 31], modules [10, 17, 18, 24], components [22], or files [25].

The prediction models are usually based on different characteristics of the code units. These characteristics are commonly presented in the form of different code metrics (e.g., [16, 27, 31]) or, for classes, variations of C&K [7] object oriented metrics (e.g., [3, 8, 31]). There are also studies that take historical information about fault-proneness of code units into account (e.g., [26, 27]).

The construction of a prediction model usually starts with the selection of the independent variables (i.e., the variables that are used to predict a dependant variable). The initial set of independent variables is often large. A common assumption is that models based on a large number of variables are less robust and have a lower practical value (more metrics have to be collected) [5, 9]. Therefore, some authors (e.g., [5]) focus on building only simple models, containing one or at most two predicators (independent variables).

A commonly used method to select the best fault predicators is correlation analysis ([5, 8, 31]). The methods for building prediction models range from uni- and multivariate linear regression (e.g., [5, 6, 21, 24, 28, 31]) and logistic regression (e.g., [3, 8, 12, 18]) through regression trees (e.g., [16, 17]) to neural networks (e.g., [19, 30]).

Despite the fact that expert judgements are an accepted and widely practiced way of performing estimations [2, 15, 29], we have found only very little research that connect expert estimations with predictions of the fault-proneness of individual code units. The only examples that we have found are studies [32, 33] in which expert estimations are used together with statistical analysis as complementary methods. The statistical methods are used to group code units with similar characteristics. Then, it is up to the expert to estimate if a given group of code units is fault-prone. We have, however, failed to find any report presenting a comparative evaluation of expert judgments and statistical fault prediction models.

3. Study objects

3.1 Systems under study

In this study we use the most current release of System A and two latest releases of System B. These are large telecommunication systems. The sizes of these systems are about 800 classes (500 KLOC), and over 1000 classes (600 KLOC) for System A and System B, respectively. Both systems are mature and have been on the market for over 6 years. Over that time the systems have evolved – a number of releases of each of them have been produced. Both systems are implemented in object oriented technology using the same programming language. One of the systems has been developed in Sweden. The other one has mostly been developed in China and is currently being transferred to Sweden.

The systems are logically divided into a number of subsystems. Each subsystem is built of components. Each component consists of a number of classes. The numbers of components that have been modified in the examined releases of the products are 35 in System A, 41 in System B1, and 43 in System B2. That corresponds to 249 modified classes in System A, 319 modified classes in System B1, and 180 modified classes in System B2. The information about faults is available at the class level. Therefore, we are able to assign faults to the particular classes and, through them, to the components.

When analyzing the code and the fault data we have found that in all three cases (System A, System B1, and System B2) the most fault-prone code is the code introduced as modifications of existing classes. In System A the code introduced as modifications of the classes from the previous release accounts for 37% of the code written (63% of the new code was introduced as new classes). These 37% of the code contained 62% of the faults found in the project release that we examine in this study. A similar trend has also been observed in System B. In System B1 about 44% of the introduced code modifies classes from the previous release. These 44% contain 78% of all faults. In System B2 the modified code accounted for 45% of code introduced in this release. The modified classes in System B2 contained 59% of faults. It can be noticed that modified code is not only more fault-prone but it also is a significant source of faults in evolving systems, like the ones we examine in this study. Therefore, in this study we focus specifically on predicting the fault densities of modified code units.

3.2 Participating experts

In total, we have invited eleven experts to this study. Six of them have been involved in the development of System A, and five of them have been involved in the development of System B2. All of our experts have several years of working experience with telecommunication systems. Their tasks are system design and implementation. All our experts are familiar with the architectures and functionalities of their respective systems. They also know the scopes of the releases under study. They know what functionality that was added in the releases they were asked to perform estimations for.

The major difference between experts involved in performing estimations concerning System A and those performing estimations concerning System B2 is their experience with the respective products. System B2 is currently being transferred from an offshore development site. Therefore, all of the experts involved in performing estimations concerning System B2 have limited experience (up to one year) of working with System B2, as compared to the six-year experience of the experts involved in the development of System A.

At the time of the study, the development of the examined releases of the systems was finished. One risk of such a study set-up is that the experts may basically know the fault distribution. We believe it has not been the case in our study. The work in the project is organized according to the "component responsibility" principle - each developer is fully responsible for one or more components. Because of that, in practice, the developers do not have a global picture concerning fault distribution – they only receive information concerning the faults that were found in the components they are responsible for. Normally, during the project time, they are not provided with any global statistics concerning faults. Therefore, their predictions concerning the faults made in this study are not based on any global statistics but on their own "gut-feeling", based on experience and knowledge of the scope of a project.

4. Methods

In this section we present the methods, which we use in this study. Section 4.1 presents the methods we used to build our prediction models. Section 4.2 presents the way we collected and used the data gathered from our experts. In Section 4.3 we present the methods we use to evaluate and compare our prediction models with the estimations of our experts.

4.1 Building prediction models

Our prediction models are built based on data from System B1. The goal of our models is to predict fault density of different code units. As we see it, the fault density can be predicted in two ways:

- by predicting the fault density (Faults/Size) – fault density is a dependant variable in the model.
- by predicting the number of faults (Faults) and dividing the predicted number of faults by real size (Size) of the code unit – Faults are predicted by the model, while size is measured.

In this study we have collected data that makes it possible for us to perform predictions both at the class and at the component level. The metrics collected at the class level are summarized in Table 1. These are mostly C&K [7] design metrics, and code metrics. The metrics collected at the component level are summarized in Table 2. These are simple code metrics measuring the size of the component and the size of the change. Within the components we have performed measurements only on those classes that were modified.

Similarly to [5], we have decided to build simple prediction models that are based on one predicator only. Such models do not suffer from the risk of multicolinearity, which is a typical risk for multivariate models [9]. Therefore, simple models are usually more likely to be stable over releases. An additional benefit from using simple models is that they require less data to be collected, as compared to multivariate models. Obviously, by using one metric only, we deliberately give up the potential benefit from introducing more information, carried by other metrics, into the model. However, as our previous research shows (see Paper V) in practice the prediction using a univariate model may be almost as good as the prediction using a multivariate model.

In order to select the best single fault predicators from the class and the component metrics we perform a correlation analysis. The correlation analysis is commonly used for that purpose by other researchers [23, 31]. It quantifies the relation between two metrics as a value between -1 and 1. An absolute value of a correlation close to 1 characterizes good predicator variables. The values close to zero indicate a very weak linear relationship between the variables, and thus a low applicability of one variable to predict the other.

Table 1. Metrics collected at the class level.

Name	**Variable**	**Description**
	Independent variables	
Coup	Coupling	Number of classes the class is coupled to [7, 11]
NoC	Number of Children	Number of immediate subclasses [7]
WMC	Weighted Methods per Class	Number of methods defined locally in the class [7]
RFC	Response for Class	Number of methods in the class including inherited ones[7]
DIT	Depth of Inheritance Tree	Maximal depth of the class in the inheritance tree[7, 10]
LCOM	Lack of Cohesion	*"how closely the local methods are related to the local instance variables in the class"* [11]. In the study LCOM was calculated as suggested by Graham [5, 6, 13, 14, 21, 28, 31]
ClassStmt	Number of statements	Number of statements in the code (used as the size metric in our study)
MaxCyc	Maximum cyclomatic complexity	The highest McCabe complexity of a function within the class
ClassChg	Change Size	Number of new and modified LOC (from previous release)
	Dependent variables	
Faults	Number of faults	Number of faults found in the class
FaultDensity	Fault density	Fault density of the class

In this study we build two prediction models. One of them predicts faults at the class level and the other one predicts faults at the component level. The models are built using a univariate linear regression. The univariate linear regression estimates the value of the dependant variable (the number of faults or the fault-density) as the function of an independent variable [24]:

$$f(x) = a + bx \quad (1)$$

Even though our prediction models attempt to predict the actual value of fault density, in this study we use this information only as an indicator of the order in which the code units should be analyzed.

Table 2. Metrics collected at the component level.

Name	Variable	Description
Independent variables		
CompStmt	Number of statements	Number of statements in the component (only statements from modified classes in the component were counted)
CompMeth	Number of methods	Number of statements in the component (only methods from modified classes in the component were counted)
CompClass	Number of modified classes	Number of modified classes in the component
CompChg	Changesize	Number of new and modified LOC (compared to previous release)
Dependent variables		
CompFaults	Number of faults	Number of faults found in the component
CompFaultDensity	Fault density	Fault density of the component (CompFaults divided by the accumulated size of the modified classes in the component)

4.2 *Expert estimation*

The expected outcome of the expert estimation is a ranking of the code units according to their decreasing fault density. Such a ranking makes it possible to compare the accuracy of an expert estimation with the accuracy of a prediction made by our prediction models.

In the beginning of this study we have performed a number of interviews with our experts. The goal was to establish an appropriate level for performing the expert predictions. The question was if the experts should perform estimations at the class or at the component level. It quickly turned out that the class level presents too fine-grained information. Even though the experts knew what each component does, it was very difficult for them to predict the responsibility of particular classes within components. Additionally, the amount of data (249 classes for System A, and 180 for System B2) was considered unmanageable. The number of components is significantly smaller – there are 35 components with modified classes in System A and 43 in System B2. Therefore, in this study the expert estimation is performed only at the component level.

The expert estimation was performed individually by each of our experts. During the individual rankings the experts were provided with the list of modified components. Additionally, for each component, we enclosed the information concerning the subsystem to which the component belongs, as well as the accumulated size of the modified classes within the component. The experts were asked to rank the components according to their decreasing fault density. Each expert was given a clear explanation concerning our study in order to assure a full understanding of the task. Additionally, for System A we managed to organize a consensus meeting. As input to this meeting we provided the experts with the individual rankings. The goal of the consensus meeting was to prepare a common "joint" ranking of components. In all cases, the experts were allowed to not rank all the components.

4.3 Evaluation of prediction accuracy

We evaluate the statistical prediction models and the expert predictions from the perspective of the increase of the efficiency of fault detection that they provide. We consider a prediction method better if, by following it, we are able to detect more faults by analyzing the same amount of code as compared to another prediction method. Therefore, we evaluate different predictions by plotting the percentage of faults that would be detected if analyzing a system according to a certain prediction method against the accumulated percentage of code that would have to be analyzed.

To obtain a point of reference for our evaluations, we introduce two reference models:

- Random model – the model describing a completely random search for faults

- Best model – the theoretical model that makes only the right choices about which code unit to analyze first

The Random model provides a baseline for evaluating our models, as it describes what results, on average, we could expect if we analyzed the code not following any model at all. The Random model is the same for all systems – on average by analyzing *n*% of code we find *n*% of faults. The Random model looks the same for the prediction at the class and at the component level.

The Best model provides a boundary of how good the prediction can be. In this theoretical model the code units are selected according to their actual fault density. The Best model looks differently for different systems, because it depends on the actual distribution of faults in the system. The Best model is also different for predictions at the class and at the component level. The class level prediction has finer granularity and therefore, at least theoretically, it is able to provide more precise results. In this study we assess the practical value of having finer granularity prediction by comparing the Best model for components and for classes.

The evaluation of model predictions vs. expert estimations is performed by checking how each particular solution performs compared to the Best model, the Random model, and to each other. The closer the prediction is to the Best model the better it is. If the prediction is better than the Random model then we can say that using it presents an improvement over not using any method at all.

5. Results

5.1 Building prediction models

As described in Section 4.1, we begin building our prediction models with selecting the best individual fault predicator. We do that by performing a correlation analysis. In the correlation analysis we look for the best predicator of either fault density or the number of faults, as from the number of faults we can calculate the fault density by dividing the predicted number of faults by the size of a code unit (i.e., a class or a component). The correlation analysis is performed for both class and component level metrics. The class level metrics are explained in Table 1, and the component level metrics are explained in Table 2. The results of correlation analysis are presented in Table 3.

Table 3. Correlation analysis results at the class and the component level. The highest correlations for respective levels (class, component) are marked in bold.

	Class level metrics									Component level metrics			
	Coup	NOC	WMC	RFC	Class Stmt	MaxCyc	DIT	LCOM	Class Chg	Comp Chg	Comp Stmt	Comp Meth	Comp Class
Faults	0.25	-0.01	0.14	0.04	0.26	0.31	-0.07	0.13	**0.60**	**0.79**	0.63	0.35	0.55
Fault Density	0.06	-0.01	-0.01	-0.03	-0.02	0.01	-0.01	0.05	0.20	0.21	0.07	0.01	0.09

The highest correlations are marked in bold in Table 3. As it can be noticed, the most promising fault predicator for both classes and components is the size of the modification (ClassChg metric at the class level, and CompChg metric at the component level). In both cases the correlation coefficients are the highest when predicting the number of faults. Therefore, we build models that predict the number of faults and we divide their output by the size of the respective code unit, i.e., the class or component.

The models based on ClassChg and CompChg are built using the linear regression. The results of model building are presented in Table 4. As both models are based on the information concerning the size of the modification, not surprisingly they look quite similar.

Table 4. Prediction models build in the study. "Prediction level" indicates if the models works at class or at component level. "Model calculated" is the model obtained by linear regression. "Model applied" is the transformation of the "Model calculated" so that it predicts fault density instead of the number of faults.

Model name	Prediction level	Model calculated	Model applied
ComponentPred	Component	Faults=0.002*ComChg + 0.209	FaultDensity=(0.002*ComChg + 0.209) / CompStmt
ClassPred	Class	Faults=0.002*ClassChg + 0.018	FaultDensity=(0.002*ClassChg + 0.018) / ClassStmt

5.2 *Expert estimations*

5.2.1 Expert predictions concerning System A

In total, six experts performed predictions concerning System A. At first they performed ranking of the components individually. Later the group of experts was presented with the task of making one joint decision using individual results as input to the discussion. The distribution of “votes” of individual experts, the group consensus, and the actual ranks of the components are presented in Table 5. Only those components that were selected by at least one expert are presented.

In the individual rankings neither of our experts ranked all 35 components. In fact, each expert ranked between 5 and 8 components. Altogether, the experts pointed out 15 different components, i.e., neither of them had any opinion about the fault-proneness of the remaining 20 components. These 15 ranked components together account for about 60% of the code. The “group consensus” was apparently more difficult to reach than the individual rankings because the experts ranked only 4 components. These four components accounted for about 30% of the code.

Table 5. The rankings of individual experts and the joint ranking of all experts. Only 15 components out of 35 were selected, and only those components are presented in the table below. Lower rank value indicates higher fault-density in the component predicted by expert. “Correct ranking” is the actual rank of the component when all 35 components are ranked. The components are presented in the order of their decreasing fault density.

	Component														
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
Expert1 ranking				1	4	2			3	5					
Expert2 ranking		3				1						2		4	5
Expert3 ranking	5	3		6	7	1				2		4			
Expert4 ranking		2		6	3	1	5			4		7			
Expert5 ranking		3	5		1	2		4	6			7			8
Expert6 ranking		4		1	2					5	7	8	3	6	
Group consensus ranking		3		4		1				2					
Correct ranking	5	6	7	8	9	10	12	13	16	17	18	22	24	25	26

It can be noticed that in the individual rankings the components can be divided into two subgroups. One subgroup contains components that were selected by a majority of the experts, i.e., four and more experts pin-pointed them. These are components with numbers: 2, 4, 5, 6, 10, 12. The other group consists of components selected only by one or two interviewees, i.e., components with numbers: 1, 3, 7, 8, 9, 11, 13, 14, 15. It is quite clear that apart from two exceptions (Component 2, and Component 6) there is a rather large discrepancy between the ranks assigned by the experts to the components. This means that, despite the fact that most experts considered a certain component fault-prone, their estimation of its fault-density was different.

All components ranked in the “group consensus” ranking are the components that were selected by the majority of experts in the individual rankings. Component 2 and Component 6 were ranked according to the trend from the individual rankings, which was not surprising because the experts were quite consistent in ranking them as the first and the third in the individual rankings. Ranking Component 10 and Component 4 as the second and the fourth, respectively, must have been an outcome of the group discussion, because such a ranking was not suggested by any individual expert.

At this point we can also perform some initial assessment of the accuracy of the expert prediction concerning System A. By comparing the results of individual experts with the actual ranking of components (the last row in the table) we can see that out of the 15 components the experts pointed to, only 8 belong to the actual top 15 components with the highest fault densities in System A. By comparing the “consensus group” ranking with the actual ranking of components we can see that neither of the components pin-pointed by the experts belongs to the top 4 components with the highest fault-densities. In fact, also in the individual rankings none of the experts identified any of the four most fault-prone components in System A.

5.2.2 Expert predictions concerning System B2

Five experts performed predictions concerning System A. Their individual rankings together with the actual ranks of the components are presented in Table 6. Only those components that were selected by at least one expert are presented. Out of 43 components that were modified in System B2, the experts pointed out 16 components, i.e., neither of our experts had any opinion regarding remaining 27 components. The 16 components selected by our experts account for about 66% of the code.

Similarly to the predictions concerning System A, in System B2 the ranked components can be divided into two subgroups. One subgroup consists of the components that were selected by the majority of experts, i.e., that were selected by at least 3 experts. To this group belong components with numbers: 2, 5, 6, 8, 12. The other subgroup consists of components that were selected by the minority of our experts. These are components with numbers: 1, 3, 4, 7, 9, 10, 11, 13, 14, 15, 16.

It seems that the agreement concerning the rankings was quite low among the experts that performed predictions in System B2. From the components selected by the majority of the experts, the highest agreement was achieved for the components with numbers 5 and 12. We see this agreement as weaker, as compared to the agreement concerning Component 2 and Component 6 in System A. This is, however, our subjective judgment only.

Table 6. The rankings of individual experts concerning System B2. 16 components out of 43 were selected, and only those components are presented in the table below. Lower rank value indicates higher fault-density in the component predicted by expert. "Correct ranking" is the actual rank of the component when all 43 components are ranked. The components are presented in the order of their decreasing fault density.

	Component															
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
Expert1 ranking	1		4		3							2				
Expert2 ranking		4			1	3	6		2	5						
Expert3 ranking					1	3		4				2				
Expert4 ranking		8	9	3	2	7		1			4	5	6	9	10	11
Expert5 ranking		2		4	5			1				3				
Correct ranking	2	3	9	10	11	12	13	14	15	16	19	20	21	22	23	24

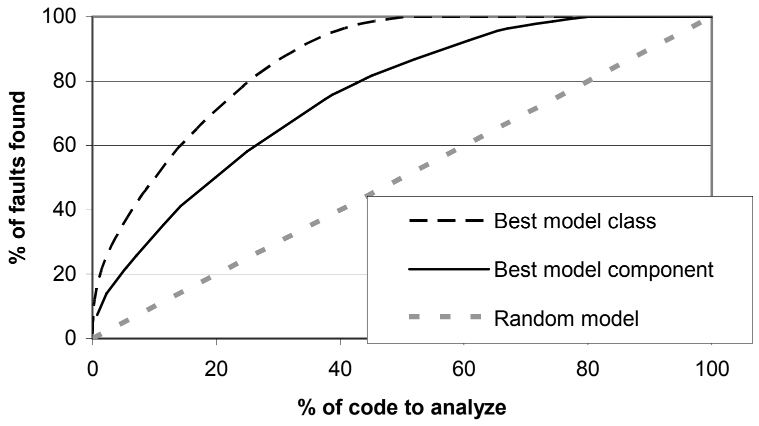
The accuracy of expert predictions concerning System B2 seems to be similar to the accuracy of expert predictions concerning System A. Out of 16 components selected, 10 belong to the actual top 16 most fault prone components in System B2.

5.3 *Evaluation of prediction accuracy*

Our prediction starts with building the reference models. For both systems we build three reference models, one describing an average result of random picking of code units for analysis (Random model), and two models describing the theoretical best result that can be obtained. One of them describes the maximum that can be obtained when predicting faults at the class level (Best model class), the other when predicting at the component level (Best model component).

The reference models created for System A are presented in Figure 1. The reference models created for System B2 are presented in Figure 2. As can be noticed, both graphs look similar, and some common conclusions can be drawn for both systems. In both systems the Random model is quite far from the best possible model, which indicates that there is large room for efficiency improvement that can be filled by an accurate fault prediction. For example, by analyzing 20% of the code randomly we can find 20% of faults. Ideally, in both systems by analyzing the most fault-prone 20% of the code we should be able to find up to 70% of the faults in case of the class level prediction (see Best model class in Figure 1 and Figure 2), and up to about 50% of the faults, if the prediction is made at the component level (see Best model component in Figure 1 and Figure 2).

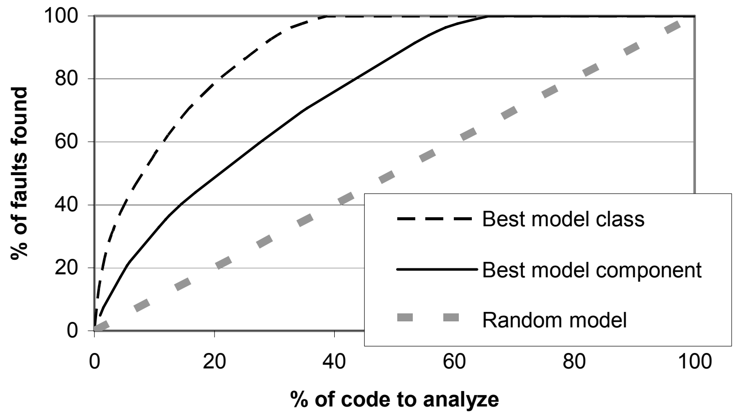
Figure 1. **Reference models in System A – the evaluation**



Although theoretical, the higher maximum possible improvement achieved by predicting at the class level indicates that the class level prediction should be able to give better results. The class level prediction is made based on more fine-grained information and, therefore, it is more precise. From Figure 1 we see that, theoretically, the best component level prediction is capable of providing about two-third of the improvement over the random model offered by the best class level prediction (in Figure 1 the distance between Best model component and Best model class is more or less equal to 2/3 of the distance between the Best model class and the Random model). The gain from using a class level prediction is even more visible in System B2. In Figure 2 we can see that the best component level prediction can be only half as good as the best class level prediction. The reader must bear in mind that this discussion concerns the best possible models that predict fault density at the respective code unit levels. It does not reflect the performance of our models.

The evaluation of expert estimations and our prediction models when applied to System A is presented in Figure 3. In Figure 3 we present all the individual expert estimations, "group consensus" estimation, both of our statistical prediction models (ClassPred, ComponentPred), and three reference models (Random model, Best model class, and Best model component).

Figure 2. Reference models in System B2 – the evaluation

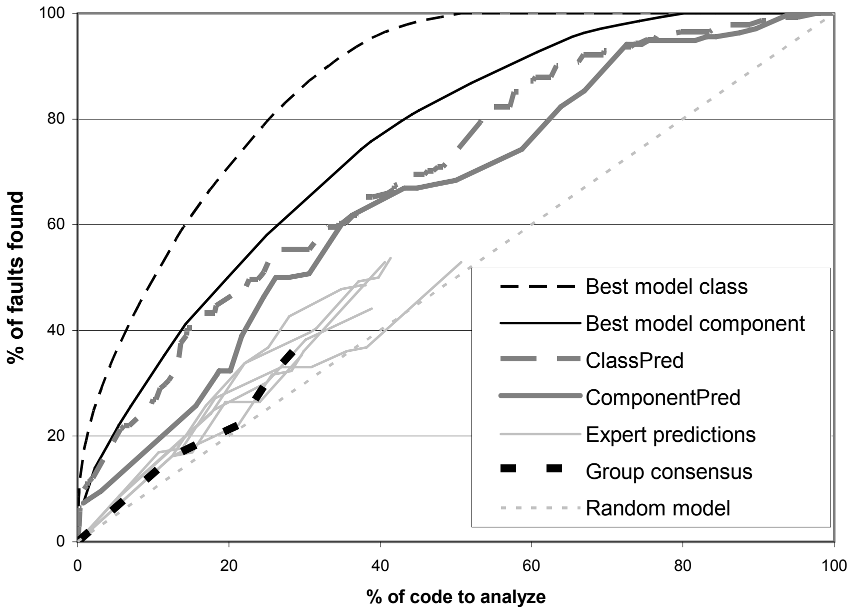


From Figure 3 we can conclude that both statistical prediction models clearly outperform the expert estimations. They not only offer higher accuracy in the range of code covered by any of the expert estimations (approximately up to 50% of code of System A) but also provide predictions that are significantly better compared to the Random model for the rest of the code. By comparing ClassPred with the best of the expert estimations for the percentage of code covered by the expert estimations we can see that ClassPred offers three times as big improvement over the Random model as the best of expert estimations.

Other findings from Figure 3 concern the practical gain from using more fine grained information and predicting at the class level. As we can see there is a clear gain connected with predicting at the class level. For example, for the range of code covered by expert estimations the gain from using ClassPred is almost equal to the maximum possible gain from using any component level prediction model, i.e., compared to the Best model component.

Quite surprisingly the “Group consensus” estimation turns out to be one of the worst estimations made by our experts. Some of the individual estimations are actually not only more correct but they also account for more code.

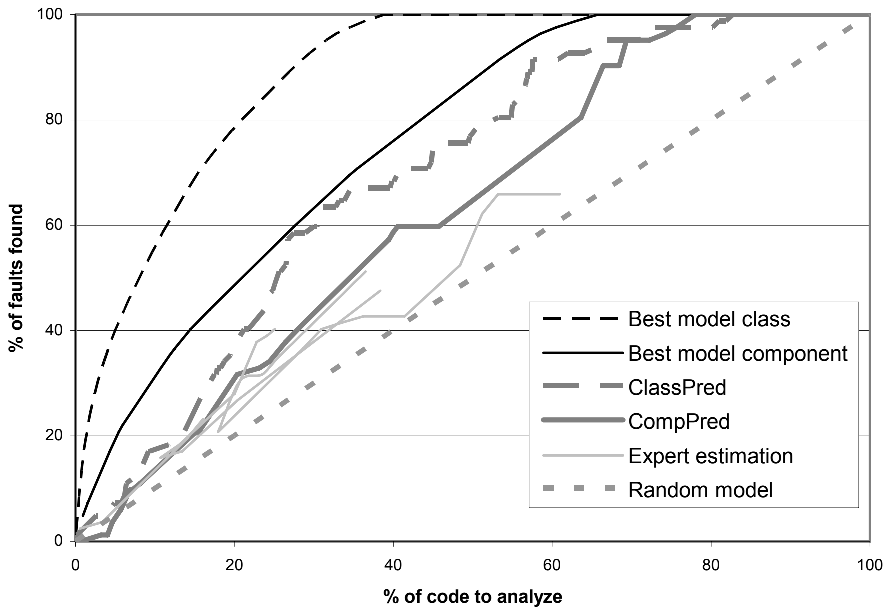
Figure 3. Statistical prediction model vs. expert prediction in System A – the evaluation of accuracy.



The evaluation of expert estimations and our prediction models when applied to System B2 is presented in Figure 4. As in the case of System A, we present all the individual expert estimations, both of our statistical prediction models (ClassPred, ComponentPred), and three reference models (Random model, Best model class, and Best model component).

In Figure 4 we can see that for small percentages of the code (i.e., up to about 15% of the code) both statistical prediction methods and expert estimations provide equal gain over the Random model. When over 15% of code is analyzed, the gain from using the statistical prediction model is significantly larger compared to the gain from using any of the expert estimations. It is clearly visible in the case of the class level prediction (i.e., the ClassPred model). ClassPred provides a constant improvement over the Random model. It not only outperforms all expert estimations, but provides a significant improvement over the Random model for the range of code not covered by any of the expert estimations.

Figure 4. Statistical prediction model vs. expert prediction in System B2 – the evaluation of accuracy.



The CompPred model is visibly worse than ClassPred. It is, however, not worse than the best of expert estimations. Additionally, the CompPred model provides an improvement over the Random model even for the range of code not covered by the expert estimations.

6. Discussion

6.1 Findings

The results obtained in our study seem to support the idea of building fault prediction models. We have shown that statistical models have some advantages over human expert estimation. The biggest advantage of statistical models is that they are not negatively affected by the size of the dataset. Therefore, statistical prediction models are able to estimate the fault-proneness of all code units even in large systems. Ranking all code units in a large system may be a difficult task for human experts. For example, our experts were quite confident when it comes to ranking the first couple of most fault-prone components.

Beyond a certain number of components they admitted they would put remaining components in a random order.

The other advantage of prediction models is a direct effect of their ability to cope with large datasets. As we have shown, the statistical prediction models can successfully operate on fine-grained data, e.g., they can predict the fault-proneness of individual classes instead of predicting the fault-proneness of entire components. Our results indicate that human experts, irrespectively of their experience, may not be able to grasp large and complex system structures. This makes it difficult for human experts to make predictions at a low level of a system structure. At the same time we have shown that predicting at a low level brings not only theoretical but also practical benefits. In both systems, which we analyzed, the theoretical best prediction at the component level provides on average only about 40% to 60% of the improvement that can be offered by the best theoretical prediction at the class level (compare Best model component with Best model class in Figures 3 and 4). The superiority of the class level prediction is also visible in practice. Our class level prediction model noticeably outperforms our component level prediction model in all cases.

There is also one more advantage of predication models, which we have not evaluated in this study. It is their cost. They are reasonably cheap to build and even cheaper to apply – normally, they can be implemented in a form of e.g., a script that collects and processes all the required information automatically. An expert estimation is more expensive, since for each project it must be set up and performed independently. Obviously, to perform expert estimation we need experts. Sometimes, in relatively new projects, or when projects are overtaken by another team of designers, the experts may simply not be available.

On the other hand, an expert estimation has some positive aspects, also not evaluated in this study. Our statistical prediction models predict faults, but do not classify them in any way. Naturally, not all faults are the same – some of them may be more difficult to find than the others. Some faults may be more severe than the others. It is possible that the experts tend to pin-point more correctly the components that are more likely to contain these kinds of faults. Due to the lack of appropriate data we could not verify this hypothesis in our study.

Another benefit of the expert estimation can be its flexibility. The experts can take into account information that is not present in the statistical model. For example, in our statistical prediction models the size of the modification is considered to be the best fault predicator and all our models are based on it. However, it may happen so, that even though the change in the component is relatively small, there was a number of people involved in introducing it, which may make such a change more prone to faults compared to a change introduced by a single designer. Such rare, project-specific issues are likely to be captured by experts but it is very difficult to predict them in advance and incorporate them into a statistical prediction model.

However, when analysing estimations of our experts, we have found a number of worrying factors. The experts do not agree with each other, they either select different components, or, if they select the same components, they estimate their fault density differently. They also seem to have problems identifying the most fault prone components in the system. In Table 5 and Table 6 we can see that, even though most of the experts agree on the high fault densities in some of the components, these components are actually not the most fault prone in the respective systems.

Another interesting observation that can be made when comparing the performance of experts in System A and in System B2 is that much longer experience with the product does not affect the accuracy of the expert prediction. Our experts involved in performing the estimations concerning System A have about five years longer experience with the product than the experts involved in the estimations concerning System B2. However, the accuracy of the expert predictions concerning both systems is not very different. In both systems the experts have selected a similar number of components. These components account for a similar percentage of code (see Sections 5.2.1 and 5.2.2 for the details concerning the expert prediction results). From Figure 3 and Figure 4 we can see that the fault detection efficiency improvement gained by using expert predictions is similar in both systems. Since it is unreasonable to assume that product related experience has no impact on the accuracy of fault prediction, the only possible conclusion is that there is some threshold value connected with experience, after which the accuracy of predictions is more or less similar. It is, however, important to remember that what we discuss here is a product related experience, not the experience as a whole. All our experts had experience in the development domain (i.e., telecommunications), which may additionally explain the similarity in their performance.

6.2 Validity

The reader must bear in mind that this paper has been meant more as an experience report than a formal experiment report. Our selection of projects was convenience-based – we have selected projects that were available to us. Also, since the entire exercise has not been performed as a controlled experiment, we can not assure that e.g., the experts did not have some at least partial knowledge about the actual fault-proneness of the components. For the reasons described in Section 3.2, and because of their poor performance, we believe this was not the case but we can not claim that we have eliminated this risk utterly. The number of experts involved in each project may also be considered small and therefore it is difficult to perform any meaningful statistical analysis of their performance. However, there are a number of issues that make it easier to generalize findings from our study. The study was performed in an industrial setting. We used real, large telecommunication systems. Our experts had real experience and knowledge about the project. Additionally, they were motivated and interested in the study, which should have contributed positively to the quality of their predictions.

We also believe that our statistical prediction models obtained in this study are general. When building them, we followed the good academic practice of building models on different data than the data used to evaluate the models. We evaluated our models not only using the next release of the system the models were built on, but also using another system. We believe that all these factors make the evaluation of our statistical prediction models reliable.

Therefore, we believe that some general lessons can be learned from our study. It seems very probable that most experts would face the problems our experts faced, e.g., problems with coping with large amounts of data. It is also very likely that prediction at a low level, like e.g., at the class level, would give better results compared to prediction at a higher level, e.g., at the component level. We are almost sure that for most medium-to-large systems the class level prediction is not feasible to be performed by people.

Most issues concerning the expert estimation validity, like experts' possible knowledge about the actual fault distribution, should result in better than average performance of the experts. It might be considered as an argument supporting our conclusions, because even with this "handicap", the expert estimations were outperformed by the statistical prediction models.

7. Conclusions

The goal of this study was to compare the accuracy of fault predictions made by statistical fault prediction models with the accuracy of fault predictions made by human experts. We compared both prediction methods by applying them to two large software systems from the telecommunication domain. To perform the study we invited eleven experts involved in the development of these systems and we built two statistical fault prediction models. Our statistical fault prediction models were built based on data different from the data used in the evaluation.

The evaluation was performed from the perspective of an increase of the fault detection efficiency that could have been obtained if analyzing the code units in the order suggested by the experts or in the order suggested by our statistical models. Both prediction methods were evaluated against three reference models: a model based on a random selection of the code units for analysis, the theoretically best model for predicting faults at the class level, and the theoretically best model for predicting faults at the component level.

We found that both the expert estimations and the statistical prediction models provided an improvement over the random selection of code units for analysis. When comparing the performance of the expert estimations with the performance of the statistical models we found that the statistical prediction models outperformed the expert estimations. For example, for the systems that we analyze in this study, we find that for the portion of code covered by the expert estimations our statistical fault prediction models offered a higher improvement as compared to the best of the expert estimations. Moreover, the statistical predictions continued to provide an efficiency improvement over not using any model even after the point where our experts gave up.

We identified a number of reasons for the statistical models being better. Statistical models are not affected by the size of the dataset so they perform equally well on small and large systems, while the human ability to grasp the complexity of larger systems is limited. In addition, the ability to deal with large datasets makes it possible for the statistical models to perform more fine-grained predictions, i.e., predictions at a lower level. We showed that a more fine-grained prediction, e.g., a prediction at the class level instead of at the component level, is not only better from a theoretical but also from a practical perspective. Our class level prediction model was more accurate compared to our component level prediction model in both examined systems.

We also made a number of observations concerning estimations made by our experts. One worrying factor was that the components which the experts selected, in large proportion were not the actual most fault-prone components in the systems. Another worrying issue was a low agreement between our experts concerning the estimation of the fault-proneness of components. Our experts either selected different components for analysis or, if they selected the same components, they assessed their fault-proneness differently.

In this study we also discussed other advantages and disadvantages of statistical prediction models and expert estimations. The statistical prediction methods are reasonably cheap to build and apply, as well as they can be used in the absence of experts, e.g., when a project is transferred to another development organization. On the other hand, expert estimations are more flexible and can take into account some project specific issues that can affect fault-proneness of the components. Such project specific issues are usually hard to incorporate into otherwise general statistical fault prediction models.

8. Acknowledgments

The authors would like to thank Ericsson for providing us with the data, Ericsson staff members for active participation in this study, and The Collaborative Software Development Laboratory, University of Hawaii, USA (http://csdl.ics.hawaii.edu/) for the LOCC application.

This work was partly funded by The Knowledge Foundation in Sweden under a research grant for the project "Blekinge - Engineering Software Qualities (BESQ)" (http://www.bth.se/besq).

9. References

[1] V.R. Basili and L.C. Briand, A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22 (1996), 751-762.

[2] B.W. Boehm, Software engineering economics, Prentice-Hall, Englewood Cliffs, N.J., (1981).

[3] L.C. Briand, J. Wust, J.W. Daly, and D.V. Porter, Exploring the relationship between design measures and software quality in object-oriented systems. *The Journal of Systems and Software*, 51 (2000), 245-273.

[4] L.C. Briand, J. Wust, S.V. Ikonomovski, and L. H., Investigating quality factors in object-oriented designs: an industrial case study. *Proc. of the 1999 Int'l Conf. on Software Eng.*, (1999), 345-354.

[5] M. Cartwright and M. Shepperd, An empirical investigation of an object-oriented software system. *IEEE Transactions on Software Engineering*, 26 (2000), 786-796.

[6] S.R. Chidamber, D.P. Darcy, and C.F. Kemerer, Managerial use of metrics for object-oriented software: an exploratory analysis. *IEEE Transactions on Software Engineering*, 24 (1998), 629-639.

[7] S.R. Chidamber and C.F. Kemerer, A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20 (1994), 476-494.

[8] K. El Emam, W.L. Melo, and J.C. Machado, The prediction of faulty classes using object-oriented design metrics. *The Journal of Systems and Software*, 56 (2001), 63-75.

[9] N. Fenton and M. Neil, A critique of software defect prediction models. *IEEE Transactions on Software Engineering*, 25 (1999), 675-689.

[10] N. Fenton and N. Ohlsson, Quantitative analysis of faults and failures in a complex software system. *IEEE Transactions on Software Engineering*, 26 (2000), 797-814.

[11] N. Fenton and S.L. Pfleeger, Software metrics: a rigorous and practical approach, PWS, London; Boston, (1997).

[12] F. Fioravanti and P. Nesi, A study on fault-proneness detection of object-oriented systems. *Fifth European Conference on Software Maintenance and Reengineering*, (2001), 121-130.

[13] I. Graham, Migrating to object technology, Addison-Wesley Pub. Co., Wokingham, England; Reading, Mass., (1995).

[14] B. Henderson-Sellers, L.L. Constantine, and I.M. Graham, Coupling and cohesion (towards a valid metrics suite for object-oriented analysis and design). *Object Oriented Systems*, 3 (1996), 143-158.

[15] R.T. Hughes, Expert judgement as an estimating method. *Information and Software Technology*, 38 (1996), 67-76.

[16] T.M. Khoshgoftaar, E.B. Allen, and J. Deng, Controlling overfitting in software quality models: experiments with regression trees and classification. *Proc. of The 17th International Software Metrics Symposium*, (2000), 190-198.

[17] T.M. Khoshgoftaar, E.B. Allen, and D. Jianyu, Using regression trees to classify fault-prone software modules. *IEEE Transactions on Reliability*, 51 (2002), 455-462.

[18] T.M. Khoshgoftaar, E.B. Allen, W.D. Jones, and J.P. Hudepohl, Accuracy of software quality models over multiple releases. *Annals of Software Engineering*, 9 (2000), 103-116.

[19] T.M. Khoshgoftaar and N. Seliya, Fault Prediction Modeling for Software Quality Estimation: Comparing Commonly Used Techniques. *Empirical Software Engineering*, 8 (2003), 255-283.

[20] X. Li, Z. Liu, B. Pan, and D. Xing, A measurement tool for object oriented software and measurement experiments with it, *10th International Workshop New Approaches in Software Measurement*, Springer-Verlag, Berlin, Germany, (2001), 44-54.

[21] A.P. Nikora and J.C. Munson, Developing fault predictors for evolving software systems. *Proc. of The Ninth International Software Metrics Symposium*, (2003), 338-349.

[22] M.C. Ohlsson, A. Andrews Amschler, and C. Wohlin, Modelling fault-proneness statistically over a sequence of releases: a case study. *Journal of Software Maintenance and Evolution: Research and Practice*, 13 (2001), 167-199.

[23] N. Ohlsson, A.C. Eriksson, and M. Helander, Early Risk-Management by Identification of Fault-prone Modules. *Empirical Software Engineering*, 2 (1997), 166-173.

[24] N. Ohlsson, M. Zhao, and M. Helander, Application of multivariate analysis for software fault prediction. *Software Quality Journal*, 7 (1998), 51-66.

[25] T.J. Ostrand, E.J. Weyuker, and R.M. Bell, Predicting the Location and Number of Faults in Large Software Systems. *IEEE Transactions on Software Engineering*, 31 (2005), 340-355.

[26] M. Pighin and A. Marzona, An empirical analysis of fault persistence through software releases. *Proceedings of the International Symposium on Empirical Software Engineering*, (2003), 206-212.

[27] M. Pighin and A. Marzona, Reducing Corrective Maintenance Effort Considering Module's History. *Proc. of Ninth European Conference on Software Maintenance and Reengineering*, (2005), 232-235.

[28] Y. Ping, T. Systa, and H. Muller, Predicting fault-proneness using OO metrics. An industrial case study. *Proc. of The Sixth European Conference on Software Maintenance and Reengineering*, (2002), 99-107.

[29] M. Shepperd and M. Cartwright, Predicting with sparse data. *IEEE Transactions on Software Engineering*, 27 (2001), 987-998.

[30] H. SungBack and K. Kapsu, Identifying fault-prone function blocks using the neural networks - an empirical study. *IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, 2 (1997), 790-793.

[31] M. Zhao, C. Wohlin, N. Ohlsson, and M. Xie, A comparison between software design and code metrics for the prediction of software fault content. *Information and Software Technology*, 40 (1998), 801-809.

[32] S. Zhong, T.M. Khoshgoftaar, and N. Seliya, Analyzing software measurement data with clustering techniques. *IEEE Intelligent Systems*, 19 (2004), 20-27.

[33] S. Zhong, T.M. Khoshgoftaar, and N. Seliya, Unsupervised learning for expert-based software quality estimation. *Proceedings of the Eighth IEEE International Symposium on High Assurance Systems Engineering*, (2004), 149-155.

Paper VIII

Comparing the Fault-Proneness of New and Modified Code – An Industrial Case Study

Piotr Tomaszewski, Lars-Ola Damm

To appear in the Proceedings of International Symposium on Empirical Software Engineering, September 2006, Rio de Janeiro, Brazil

Abstract

Faults are considered as one of the important factors affecting the cost of software development projects. To be able to efficiently handle faults, we must increase our understanding of the factors that make the code fault-prone. A majority of large software systems evolve during their lifetime. In each new release of the system the functionality can be added by writing new classes or/and by modifying already existing ones. In this study we compared the fault-proneness of new and modified classes in such systems. Our study is based on two releases of two large telecommunication systems developed at Ericsson. The major finding of the study is that the risk of introducing faults (the number of faults in the class/the number of new or modified lines of code in the class) is 20 to 40 times as high in modified classes compared to new ones. In the systems which we analyzed a small modification (a few percent) of the class resulted in as many faults as we would expect when the same class was written from scratch. Previous research on this relationship does not appear to exist. Partly in conflict with related research, we found that there is no statistically significant difference between the average number of faults in modified and new classes, and that the average fault-densities (the number of faults/the size of the entire class) in new and in modified classes are very similar. Finally, we also suggest how our findings can be used in practice..

1. Introduction

Faults are widely recognized as one of the most important cost drivers in software development. A lot of work has been put into finding methods for efficient fault handling [2, 9]. There are methods for finding faults in the code but there are also preventive measures that can lead to producing less fault prone code. In order to use them efficiently it is important to focus them on the code that is most likely to be the most fault-prone [2, 9]. To achieve that we must increase our knowledge and understanding of the characteristics of the code that in practice make it fault-prone.

It is more a rule than an exception that large software systems evolve. New versions of systems are produced to adapt them to new or changing needs. Adding new functionality means that some new code must be introduced into the current system. In object oriented systems this code can be introduced either as modification of already existing classes or as new classes.

The relation between the fault-proneness of the code introduced as modification of an existing code unit and the fault-proneness of the code introduced as a new code unit is not entirely clear. Some researchers see modification of existing code as an example of reuse, and therefore consider modified code less fault-prone than new code, e.g., [8]. Other researchers report that there is no significant difference between the fault-proneness of new and modified code [7]. These reports result in very mixed signals sent to software developers regarding the fault-proneness of modified code as compared to the new one. As we discuss in the "Related work" section, the differences in reports can partially be attributed to the differences in findings. However, partially they are a result of different assumptions or interpretations, or due to a different understanding of the term "fault-proneness".

In this paper we understand the term "fault-prone code" in three different ways:

1. Code is fault-prone when it contains many faults, i.e., there is a large number of faults per class
2. Code is fault-prone when it has a high *fault density*, i.e., a class has a large number of faults per a line of code
3. Code is fault-prone when writing it leads to introducing many faults into the class, i.e., the number of faults per a written or modified line of code in the class is high

The difference between point 2 and point 3 in the list above is visible in the case of modified classes, where the size of the modification is smaller than the size of the entire class. Below, we explain our three ways of understanding the “fault-proneness” term in more detail.

In the literature, the most popular trend is to relate the fault-proneness of a code unit (e.g., class, module, file) to the number of faults in the code unit [6-8], which corresponds to point 1 from our list. If we consider that focusing our fault detection efforts on classes with the larges number of faults is the most efficient, then we assume that the cost of finding all faults in the class is constant, i.e., it is the same for all classes independently of their sizes.

However, some authors (e.g., [6]) admit that it might be considered more correct to consider fault-density as a measure of fault-proneness of the code unit because fault-density measure captures some other cost perspective. It assumes that the cost of performing a fault detection activity on the code unit is proportional to the size of this code unit (such an assumption can be found in [3]). This understanding corresponds to point 2 from our list.

There is also another cost model that can be taken into account. The cost in that model can potentially relate to the amount of code that was actually written (i.e., the size of the modification). In case of new classes it would be the number of code lines in the class. In case of modified classes it would be the number of new or modified lines of code. By dividing the number of faults in the class by such a cost metric we would obtain some kind of *risk* metric, describing the risk of producing a fault when introducing code into a new class as compared to the risk of producing a fault when introducing the same amount of code as a modification of an existing class. This understanding of the term “fault-proneness” corresponds to point 3 from our list.

Even though these three “fault-proneness” definitions are different, they may all be of interest in certain cases. It is so because there is a wide range of fault detection techniques, which have different cost models. For example, one can reasonably argue that the efficiency of extensive code inspections, where the code of an entire class is inspected, is related to the fault-density of the class. On the other hand, the efficiency of a code-walkthrough that focuses on the code that was actually written/modified is more related to the number of faults per written/modified code line, i.e., to our risk metric. Since, as we see it, all three perspectives can be meaningful in some cases, we take all of them into account in our study.

Our study is based on the data collected from two consecutive releases of two large telecommunication systems produced by Ericsson. Both these systems operate in the service layer of a mobile phone network. The systems were developed in object oriented technology using C++. They consist of about 1000 classes and about 500 KLOC each. Each system is divided into a number of components (around 40 components in each of the systems). In each of the releases under study between 60 and 120 new classes were added and between 175 and 320 of existing classes were modified. The staff involved in development of those system can be considered experienced, every person involved had several years of experienced with the respective systems. As these systems are mission critical they undergo expensive and extensive testing (mostly the focus is put on function and system testing).

Based on the data collected from the systems described above we attempt to answer the following specific research questions:

- Q1. Is there a significant difference between the number of faults in new and modified classes?
- Q2. Is there a significant difference between the fault densities of new and of modified classes?
- Q3. Is there a significant difference in the risk of producing a fault when introducing code in the new class as compared to the risk of producing a fault when introducing the same amount of code as the modification of an existing class?

The reminder of this paper is structured as follows. In Section 2 we present the results of related studies. In Section 3 we describe methods we used. In Section 4 we present our results. In Section 5 we discuss our findings. Section 6 contains the most important conclusions from our study.

2. Related work

We failed to find a lot of research that focuses specifically on comparing the fault-proneness of new and modified code. There are some studies that as one of the advantages of code reuse present the lower fault-proneness of the reused code units as compared to the fault proneness of new code units. In [8] Selby quantified the fault proneness of reused components as "74% less than that of newly developed components"[8]. However, as reused components Selby considered those that were both modified and unmodified. In his paper [8] we can see that the components which underwent a major revision (more than 25% of changes) are as fault prone as the new code, the components that underwent minor revision (<25% of changes) are about half as fault prone as the new ones. These are the completely reused components that make the difference, as they have almost no faults. As a measure of fault-proneness, Selby considered the number of faults per component, irrespectively of its size.

The significantly lower fault-proneness of completely reused (unmodified) components as compared to the fault proneness of the modified components is supported not only by a conventional wisdom but also some empirical studies, e.g., [4]. However, in [4], there is no comparison of the fault-proneness of new and modified components, which is the goal of our study.

Such a comparison can be extracted from [6], where Ostrand *at al.* build fault prediction models. Among the variables they used there are two categorical predictors that indicate if a file is new or changed. Based on their models, the authors showed that the modified file is likely to have about "2.9 times more faults than existing unchanged files with otherwise similar characteristics" [6]. For new files this factor is 6.4, i.e., that new files are likely to have "6.4 times more faults than existing unchanged files with otherwise similar characteristics"[6]. These two statements make it possible to conclude that new files have more faults than modified ones. However, since among those "similar characteristics", the authors mention file size we can reasonably assume that the same relation holds truth for fault densities, i.e., new files have higher fault densities than modified files. This conclusion is not based on the statistical analysis of fault proneness but on an observation of the fault prediction model built in this study [6].

The studies mentioned above tend to consider new code units as more fault prone. In [7], Pighin and Marzona came to different conclusions. By comparing an average number of faults per file they concluded that there is no statistically significant difference between new and reused code. The authors quote the average number of faults per new and per old file. The values describing the fault-proneness of new files are about 20%-60% higher compared to the values describing the fault proneness of the old files. Unfortunately, the authors also do not distinguish between modified and unmodified files. They only distinguished between new and old files. In either case, however, their results differ from the results of the other studies presented in this section.

As we can see there is no consensus between different researchers when it comes to the fault-proneness of the new and the modified code. Also we can see that in all cases presented, the authors do not take any information about the size of modification into account. Therefore, based on these studies, it is difficult to assess if e.g., it is actually more risky to modify an existing code than to write a new one. Additionally, unlike our study, most of the studies presented in this section, i.e., [4, 7, 8], were not performed on object-oriented systems and therefore it is not clear if conclusions from them apply to object oriented systems.

3. Methods

3.1 Data collected

We collected data from two consecutive releases of two systems developed by Ericsson. From now on we refer to them as to System A1, System A2, System B1, and System B2, where System A1 and System A2 are two consecutive releases of System A and System B1 and System B2 are two consecutive releases of System B. For each of the system releases we collected the data about classes that were new in each given release as well as classes that were modified in each given release, i.e., classes that were present in the previous release of the system but were modified in the release under study. We did not collect information regarding classes that were fully reused. For each class we collected the following data:

- the number of faults that were found in the class (FAULTS)
- the size of the class (SIZE) measured in thousands of codelines (KLOC)
- the size of the modification (MODSIZE) measured in thousands of codelines (KLOC). For classes that were modified MODSIZE was measured as the number of new or modified lines of code in the class, for the new classes it was equal to SIZE

The measurements concerning size and modification size were collected using the LOCC application [10]. Due to confidentiality reasons we are not allowed to reveal any information concerning the actual number of faults in the system. Therefore, our FAULTS metric has been modified – we multiplied the actual number of faults by a randomly selected value (the same in all cases). Therefore, our data is internally consistent and it makes sense to compare the numbers for the new and the modified code, but it does not make sense to use the data to predict the system quality.

Our first question Q1 (see Section 1) considers the amount of faults per class so the metric that is associated with it is, obviously, the FAULTS metric. The second question (Q2) considers class fault density.

Therefore, we introduce a new metric, called DENSITY, which we define as (1):

$$\text{DENSITY} = \frac{\text{FAULTS}}{\text{SIZE}} \tag{1}$$

Our final, third question is about the risk of introducing a codeline into the class. We associate with it a new metric, which we call RISK. The RISK metric is defined in the following way:

$$\text{RISK} = \frac{\text{FAULTS}}{\text{MODSIZE}} \tag{2}$$

3.2 Analysis methods

To answer all three questions we applied exactly the same procedure. For each variable (i.e., FAULTS, DENSITY, and RISK) in each project we started by calculating the average value and standard deviation for new and modified classes. Obtaining these values made it possible for us to give answers to our three questions. However, we could not say that the differences (or the lack of differences) between the new and the modified classes were statistically significant. Therefore, additionally we performed a statistical analysis.

In our statistical analysis, we for each project tested the hypothesis about the equality of distributions of our respective variables (i.e., FAULTS, DENSITY, and RISK). The statistical tests suggested for that purpose are [11]:

- t-test when variables are normally distributed
- Mann-Whitney U test when variables are not normally distributed

Since our data does not follow the normal distribution we used the Mann-Whitney U test. The computational procedure for Mann-Whitney U test can be found in [1, 11]. In our study we selected a standard significance level of 0.05 [1]. The statistical significance of 0.05 means that we accept 5% chance of rejecting the hypothesis about the equality of the fault-proneness of new and modified code when the hypothesis is actually correct. In other words, selecting 5% significance level means that we are at least 95% confident that the hypothesis regarding the equality of fault-proneness of new and modified code is false before we reject it.

4. Results

4.1 Question 1: FAULTS

The first question regarded the number of faults per class that can be found in the new and in the modified classes. Table 1 summarizes our findings.

Table 1. Average number of faults per modified and per new class in the respective systems. Values in brackets describe standard deviations. *Statistical sig.* says if the difference between New and Modified classes is statistically significant (i.e., if, at 0.05 level, we can reject the hypothesis that, on average, New and Modified classes have the same number of faults).

	System A1	**System A2**	**System B1**	**System B2**
Avg. $\text{FAULTS}_{\text{New}}$	0.36 (0.83)	0.32 (0.74)	0.07 (0.30)	0.47 (1.19)
Avg. $\text{FAULTS}_{\text{Modified}}$	0.62 (1.47)	0.58 (1.44)	0.14 (0.51)	0.46 (0.88)
Statistical sig.	no	no	no	no
$\frac{Avg.FAULTS_{Modified}}{Avg.FAULTS_{New}}$	1.73	1.84	1.93	0.99

From Table 1 we can see that in most of our systems an average modified class had almost twice as many faults as an average new class (see Table 1, last row). The only exception was System B2 in which the average number of faults was similar in both new and modified classes. However, in neither of our systems the difference was statistically significant at the 0.05 level.

4.2 Question 2: DENSITY

The second question regarded the difference in fault densities of new and modified classes. Table 2 summarizes our findings.

From Table 2 we can see that fault densities of new and modified classes were very similar. In the last row of Table 2 all values are rather close to 1, which suggests a similar average fault density of new and modified classes. In all studied releases, the statistical analysis indicated that the hypothesis about the equality of means could not have been rejected.

Table 2. **Average fault densities of new and modified classes in the respective systems. Values in brackets describe standard deviations. *Statistical sig.* says if the difference between New and Modified classes is statistically significant (i.e., if, at 0.05 level, we can reject the hypothesis that New and Modified classes have the same fault densities).**

	System A1	System A2	System B1	System B2
Avg. DENSITY$_{New}$	0.59 (1.65)	1.23 (3.57)	0.24 (1.37)	0.74 (1.95)
Avg. DENSITY$_{Modified}$	0.76 (1.90)	1.33 (4.56)	0.22 (0.93)	0.70 (1.70)
Statistical sig.	no	no	no	no
$\frac{Avg.\text{DENSITY}_{Modified}}{Avg.\text{DENSITY}_{New}}$	1.29	1.08	0.90	0.96

4.3 Question 3: RISK

Our last question regarded the difference in the risk of introducing a fault if writing a line of code in the new class as compared to writing the line of code in the modified class. The results are summarized in Table 3.

Table 3. **Average risk connected with introducing line of code into new and into modified classes in the respective systems. Values in brackets describe standard deviations. *Statistical sig.* says if the difference between New and Modified classes is statistically significant (i.e., if, at 0.05 level, we can reject the hypothesis that writing the same amount of code leads to introducing, on average, the same amount of faults in New and Modified classes).**

	System A1	System A2	System B1	System B2
Avg. RISK$_{New}$	0.59 (1.65)	1.23 (3.57)	0.24 (1.37)	0.74 (1.95)
Avg. RISK$_{Modified}$	23.60 (92.5)	24.31 (106.40)	11.12 (85.28)	17.71 (87.72)
Statistical sig.	yes	yes	no	yes
$\frac{Avg.\text{RISK}_{Modified}}{Avg.\text{RISK}_{New}}$	40.00	19.75	46.50	24.03

Table 3 suggests that the risk associated with introducing a code line into a modified class is significantly higher than the risk associated with introducing a code line into a new class. The last row from Table 3 clearly indicates that a codeline introduced or modified in an already existing class is about 20-40 times more likely to result in a fault, as compared to a codeline written in a new class. In most cases, our statistical analysis supports this finding. Only in System B1 we could not reject the hypothesis that the risk is similar in both cases. It is interesting, because the risk ratio (the last row in Table 3) is actually the highest in case of System B1.

5. Discussion

When comparing our results with the results obtained by other researchers we can observe some differences. One general conclusion from the studies reported in [6-8] (see Section 2 for details) is that new code units are at least as fault-prone as modified ones. In the studies reported in [6, 8] the new code units are explicitly considered more fault-prone than reused ones. In [7], the authors consider the fault-proneness of new code units to be similar to the fault-proneness of code units that existed in the previous releases of the system. However, the results concerning the average number of faults per code unit reported by them actually show that, on average, reused code units have less faults per file.

Since the studies described above consider the number of faults per code unit as a measure of fault-proneness, they should be compared to our results concerning the FAULTS metric (see Section 4.1). Our results are closest to the results obtained by [7], in the sense that we do not consider the number of faults per code unit significantly different in new and modified classes. However, even though the difference between new and modified classes was not statistically significant, the higher average number of faults per class in modified classes in three out of four of our systems indicate that the modified classes are at least as fault-prone as the new ones.

The results concerning the DENSITY metric are hard to compare with other studies, as the other studies do not quote relevant values explicitly. The only study we can refer to is the study described in [6], from which we can deduce that new code units have about twice as high fault densities as the new ones. This was not confirmed by our findings. We found that the average fault-densities in new and in modified classes are very similar.

Our last metric, i.e., the RISK metric, was introduced by us in this study, so we can not compare our findings with any results from the literature. Our results concerning the RISK metric indicate that the risk connected with writing/modifying a line of code in an already existing class is significantly higher compared to the risk connected with writing a line of code in the new class. In the cases we examined in this study the line of code written/modified in an existing class was 20 to 40 times more likely to result in a fault than the line of code written in a new class.

The differences in our results obtained for the DENSITY and for the RISK metrics suggest that, on average, in our systems the modifications of the classes were rather small, i.e., the size of modification was much smaller than the size of the modified class. The average size of modification can be roughly quantified as a couple of percent of the size of class. As a basis for this quantification we use the information that the DENSITY of new and modified classes is roughly the same, while RISK is 20 to 40 times as high in the modified classes as in new ones. Since RISK has the same value as DENSITY for the new classes this relation (20 to 40 times) also roughly describes the average difference in size between the size of modification and the size of an entire class in modified classes. According to our results for FAULTS and DENSITY metrics, such a small modification of the class resulted in as many faults as we would expect if the same class was written from scratch.

Obviously, one can reasonably argue that not all faults found in modified code are the result of a modification itself, some of them can be “inherited” with the code. In general, we agree with such a statement. However, such “inherited faults” should be also present in the fully reused, unmodified code. Since we know that among the reused code the fault-proneness of unmodified code is significantly smaller compared to the fault-proneness of modified code [4, 8] (see Section 2 for details), we see the actual modification as a factor that affects the fault-proneness of modified classes mostly. It is, however, an interesting issue for further investigation.

One practical conclusion from our study is that it seems not useful, from an efficiency perspective, to concentrate fault detection or fault prevention activities specifically on the new or on the modified code if the cost of such activities is either constant or if it is proportional to the size of the class. In both cases the efficiency of fault detection is likely to be rather similar. However, if the cost of such an activity is related to the size of the modification then we clearly suggest focusing on the modified code, as this is likely to lead to detecting more faults within given budget.

In practice, it may be sometimes hard to know exactly what the actual cost model is. Our three cases (constant, related to the size of class, and related to the size of modification) are somewhat extreme. However, as we see it, they can still be useful in a decision making process. For example, we asked our industrial partner about the cost of performing a code-walkthrough. Code-walkthroughs essentially focus on the written/modified code. Our interviewees said that neither of our cost models was fully correct, because it was rarely so that it was enough to look only at modified code. However, they said that in case of code-walkthroughs the RISK metric is much more relevant and closer to reality than DENSITY metric. It is so, because it never happens that an entire modified class is covered during a code-walkthrough. Based on this information, and our findings, we could suggest them focusing their code-walkthroughs mostly on the modified code.

The findings from our study, especially those concerning the RISK metric, may be also used in the project planning phase. Since it seems that it is much more risky to modify classes than to write new ones, one possible preventive measure can be to assign more experienced developers to tasks requiring class modification.

Theoretically, also some architectural decisions can be impacted by our findings concerning the RISK metric, e.g., to avoid class modification when introducing new functionality. On the other hand, we are aware that a line of code written in an already existing class is, on average, likely to deliver more functionality than a line of code in the new class. How to make trade-offs between these two ways of implementing functionality is an interesting research question.

High RISK values obtained for modified classes clearly suggest that class modification is a difficult task. One possible explanation of this phenomenon can be that, in practice, it is difficult to fully understand all interdependencies within the class and, therefore, it is difficult to modify a class without introducing faults. It would be interesting to see to what extent our findings concerning the risk of modifying existing code are affected by the fact that the studied systems were developed in the object-oriented technology. Modifying a class can be considered more risky than e.g., modifying a library of functions, since, potentially, we can violate some internal assumptions in the class and, in this way, cause faulty behavior of the entire class. Modification of class code is a strongly discouraged practice (see e.g., Open-Close principle [5]). However, this high risk connected with modifying a class, as compared to modifying e.g., a library of functions, is only a hypothesis, which we could not verify in this study.

Another interesting research question is connected with the fact that in Table 3 the standard deviation values for RISK for modified classes are much greater than those for new classes. This indicates that the dispersion of RISK measures is large, i.e., that there is a large variability in the risk of modifying particular classes. An interesting topic for further investigation would be finding a method for identifying classes that are especially risky to modify, i.e., with especially high RISK values. We tried to correlate high RISK values with some basic class and modification characteristics, e.g., class size, modification size, relative modification size (size of modification divided by the size of class) but we obtained rather low correlation values (Spearman correlation coefficient was positive but always below 0.3). This indicates low applicability of these measures for predicting RISK metric.

6. Conclusions

The goal of this study was to compare the fault-proneness of new and modified code. We identified three different perspectives on fault-proneness, which we later used to compare the fault proneness of new and modified classes from two consecutive releases of two large systems produced by Ericsson.

The first perspective, which we identified, relates the fault-proneness of the class to the number of faults that were found in the class. The second perspective associates the fault-proneness of the class with its fault density. In the third perspective the fault-proneness is measured as the number of faults found in the class divided by the amount of code that was actually written or modified in the class. Therefore, when comparing the fault-proneness of new and modified classes from this third perspective we compared how risky the class modifications were, as compared to writing new classes.

We found that there is no difference between the number of faults per class in new and in modified classes. On average, the modified classes had more faults, but the difference was not statistically significant. There was also no statistically significant difference between the fault densities in new and modified classes. Our results are different from the results reported by other researchers in this area. Usually they reported that new code had more faults and higher fault densities compared to modified code.

We found, however, a large and statistically significant difference between the risk of introducing a fault into the class when writing a codeline in a new class compared to writing/modifying a codeline in an existing class. We quantified this risk as 20 to 40 times as high in modified classes as in new ones. This shows that class modification is a risky and fault-prone task. This appears to be the most interesting and novel finding of our study, since we could not find any related work for this kind of measure.

Finally, we presented a number of practical conclusions that can be drawn from our study. The major conclusion is that the large risk connected with modifying existing classes suggests that special attention should be put on the tasks that require class modification. For example, the number of quality assurance activities that concentrates on reviewing the modification itself (e.g., code-walkthroughs) can be increased. However, it seems not useful to focus activities that aim at analyzing an entire class, e.g., testing, specifically on the modified code, as it is not likely to lead to finding more faults than if such activities were focused on the new classes. The same applies to focusing testing specifically on new classes – it is also not likely to bring any better results compared to focusing testing on modified classes.

7. Acknowledgements

The authors would like to thank Ericsson for providing us with the data for the study and The Collaborative Software Development Laboratory, University of Hawaii, USA (http://csdl.ics.hawaii.edu/) for the LOCC application.

This work was partly funded by The Knowledge Foundation in Sweden under a research grant for the project "Blekinge - Engineering Software Qualities (BESQ)" (http://www.bth.se/besq).

8. References

[1] A.D. Aczel and J. Sounderpandian, Complete business statistics, McGraw-Hill, Boston, Mass., (2006).

[2] B. Boehm and V.R. Basili, Software Defect Reduction Top 10 List. *Computer*, 34 (2001), 135-137.

[3] L.C. Briand, J. Wust, S.V. Ikonomovski, and L. H., Investigating quality factors in object-oriented designs: an industrial case study. *Proc. of the 1999 Int'l Conf. on Software Eng.*, (1999), 345-354.

[4] T.M. Khoshgoftaar, E.B. Allen, R. Halstead, G.P. Trio, and R.M. Flass, Using process history to predict software quality. *Computer*, 31 (1998), 66-73.

[5] B. Meyer, Object-oriented software construction, Prentice Hall, Upper Saddle River, N.J., (1997).

[6] T.J. Ostrand, E.J. Weyuker, and R.M. Bell, Predicting the Location and Number of Faults in Large Software Systems. *IEEE Transactions on Software Engineering*, 31 (2005), 340-355.

[7] M. Pighin and A. Marzona, An empirical analysis of fault persistence through software releases. *Proceedings of the International Symposium on Empirical Software Engineering*, (2003), 206-212.

[8] R.W. Selby, Empirically based analysis of failures in software systems. *IEEE Transactions on Reliability*, 39 (1990), 444-454.

[9] F. Shull, V. Basili, B. Boehm, A.W. Brown, P. Costa, M. Lindvall, D. Port, I. Rus, R. Tesoriero, and M. Zelkowitz, What we have learned about fighting defects. *Software Metrics, 2002. Proceedings. Eighth IEEE Symposium on*, (2002), 249-258.

[10] U.o.H. The Collaborative Software Development Laboratory, USA, *LOCC Project Homepage, http://csdl.ics.hawaii.edu/Tools/LOCC/*. (2005), The Collaborative Software Development Laboratory, University of Hawaii, USA. The Collaborative Software Development Laboratory, University of Hawaii, USA.

[11] C. Wohlin, P. Runeson, M. Höst, M. Ohlsson, B. Regnell, and A. Wesslen, Experimentation in software engineering: an introduction, Kluwer, Boston, (2000).

Paper IX

From Traditional to Streamline Development – Opportunities and Challenges

Piotr Tomaszewski, Patrik Berander, Lars-Ola Damm

To be submitted to a journal.

Abstract

Traditional software development processes have shown to be inappropriate for markets where it is necessary to quickly respond to changing customer needs. Therefore, a number of modern development processes that attempt to improve customer responsiveness have been developed. One such modern process is Streamline Development, a process developed by and for Ericsson AB. This paper presents an early evaluation of the suitability of Streamline Development for Ericsson. The evaluation was performed by finding positive and negative aspects of introducing Streamline Development, as well as identifying issues to address if implementing the new process. The data regarding the impact of introducing Streamline Development was collected in a series of interviews and then structured using a modification of Force Field Analysis.

1. Introduction

Today’s competitive business environment in the software domain has resulted in a situation where it is no longer enough to develop systems with adequate functionality, quality and price to remain competitive. Nowadays, it is becoming increasingly important to quickly respond to changing customer and market demands. In the last years, it has become evident that traditional software development processes do not handle this new situation very well [1, 3-5, 11, 17]. The reason for this is that traditional processes tend to have rather long life cycles and do not deliver the actual customer value (i.e. the working system) until late in the process [8]. Additionally, traditional processes are not very good at coping with changing requirements and, at the same time, due to their long duration they are especially exposed to changing market demands. Changing requirements are a source of significant amount of rework necessary to adapt the system to the new requirements [15]. Such rework caused by changing requirements is one of the most important productivity bottlenecks in large projects (see Paper I and Paper II). These drawbacks of the traditional processes are a problem in a business environment where changing user needs occur and organizations need to find ways to deal with them.

To address the problems presented above, research and practice within the software domain have switched focus from such traditional processes to processes that favor customer responsiveness [15]. In these emerging processes, the system (or parts of the system) is usually available early in the development process. This makes it possible to meet customer needs faster and deliver value to the customers earlier. Frequent releases enable early customer feedback, which results in a customer getting more involved in the entire development process. This, in turn, makes it possible to detect and address changing needs of customers much earlier and thus improves customer responsiveness and reduces the risk of waste caused by implementing inadequate functionality.

The challenges with traditional processes and the potential in the emerging processes have been recognized by Ericsson AB, one of the major software developers of telecommunication systems in the world. Currently, Ericsson’s systems are developed in a rather traditional style, in large projects that have long life cycles (often more than a year). As an effect of this, Ericsson has experienced similar problems as the ones presented above. To address these problems, a new in-house developed process called Streamline Development (SD) has been suggested.

This paper presents a study performed at Ericsson that was planned as one of the activities aiming at an early evaluation of SD applicability for replacing current development practices. The goal of this study was to identify benefits and drawbacks of SD as the development process to be used at Ericsson, as well as changes that are required to prepare the organization and products for successful implementation of SD. The study was performed in two Product Development Units (PDUs) at Ericsson. The information regarding the impact of introducing SD was collected by interviewing persons representing the roles that would be affected by a change to SD. To structure the data collected in the interviews, a modified version of Force Field Analysis [6] was used.

The paper is structured as follows: Section 2 presents the current development process used at Ericsson, as well as the SD. Section 3 describes related research. Section 4 describes the methods used in this study. Section 5 focuses the results. Section 6 discusses the findings and Section 7 concludes the work.

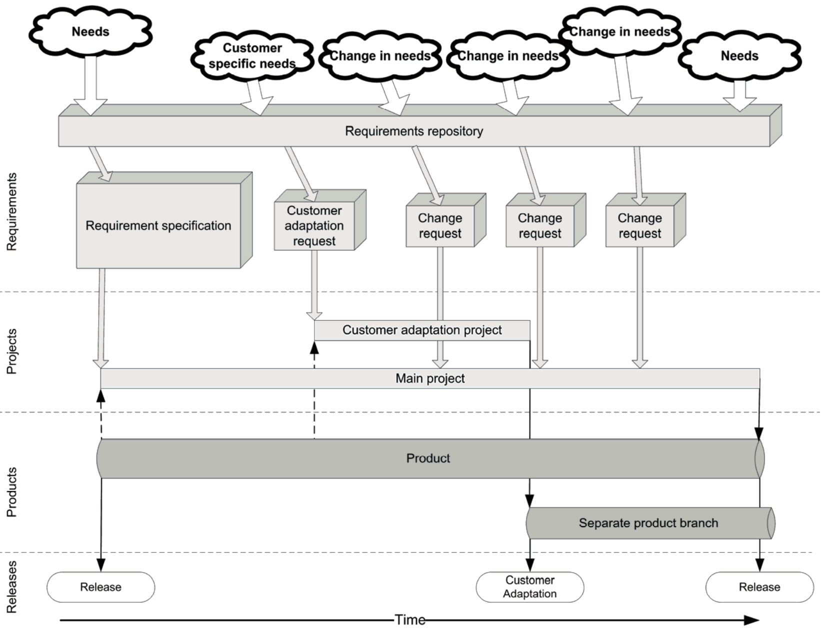
2. Ericsson AB and their development processes

Ericsson AB is an ISO 9001:2000 certified company in Sweden. Ericsson is one of the global market leaders within the telecommunications domain and sells systems to a market with basically all mobile operators as potential customers. In order to give some background regarding the work practices at Ericsson Section 2.1 presents a description of the traditional development process that is currently used at Ericsson. In addition, the major problems Ericsson is facing due to the use of this process are presented. Section 2.2 presents the suggested new way of developing software, i.e., Streamline Development (SD) and describes the ways in which it is supposed to overcome the problems of the traditional development process described in Section 2.1. The process descriptions presented in the following sections are simplified since the main intention is to underline the major differences between the processes rather than describe them in detail.

2.1 Traditional development process

Figure 1 presents a simplified visualization of how the software is developed currently at Ericsson. Normally, products are developed in a number of consecutive releases. The product releases are developed in large, long–lasting (from one to two years), projects with the scope defined upfront. The scope is set by selecting a number of requirements from the requirements repository. These requirements are refined into a requirement specification that should cover an entire project. Based on this requirement specification, a new version of the system is developed and released to the customers.

Figure 1. Traditional development process at Ericsson (one main product release).



Due to the length of the projects, it often happens that customers' needs change when the project is still ongoing. This means that some of the requirements specified in the beginning become obsolete. Therefore, some requirements need to be added, some need to be changed, and some need to be deleted to better match the customer expectations. When a change in needs is identified, a Change Request (CR) is filed (see Figure 1). A CR describes an alteration from the original requirement baseline (the changes are handled similarly to what is described in [7]). This may imply that an "original" requirement that has already been implemented must be either re-implemented or thrown away depending on the kind of CR. In either case, CRs are a source of waste in the project.

The description above primarily concerns projects aiming for a broad market launch. However, it is also common that specific customers initiate requirements that should not be included in the main release (and hence not in a main project) for some reason. Such requirements are denoted as Customer adaptation requests in Figure 1. They are usually urgent and must be addressed quickly. When a decision is made to meet such customer specific requirements a customer adaptation project is started and the requested functionality is implemented into the system. Upon completion of such a project, the product is released to the ordering customer but it is not integrated with the main product and therefore has to be maintained as a separate product branch (see Figure 1).

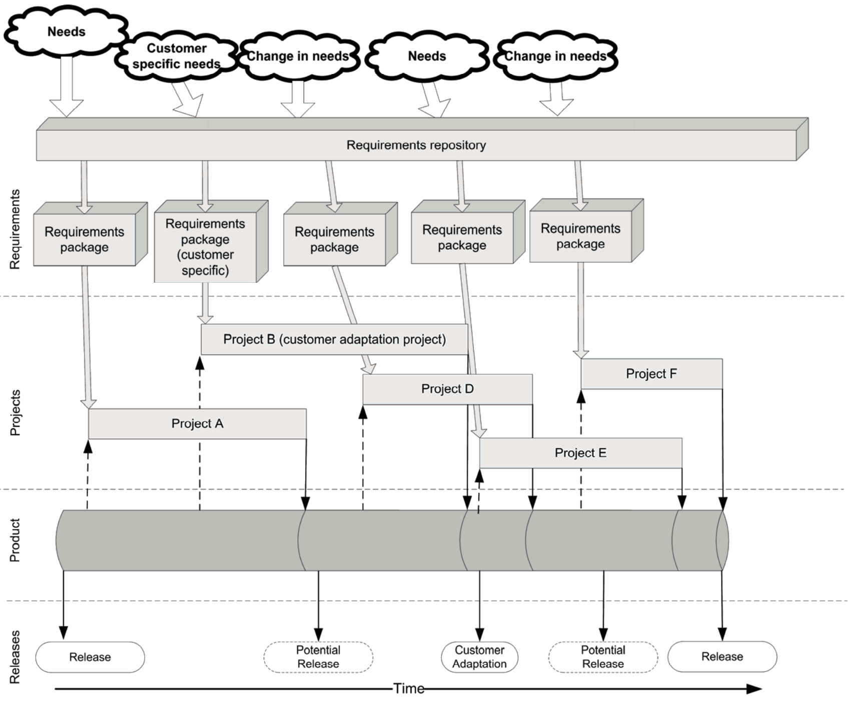
One of the main problems Ericsson has with the process described above is the high cost of handling CRs. As mentioned above, CRs are a source of waste and rework. Due to the long lead-times CRs are relatively common in the current projects at Ericsson. Therefore, CR handling accounts for a significant part of project cost, decreases productivity and increases project lead-time. In addition, long lead-times decrease the company's competitiveness by lowering customer responsiveness. Another problem faced in traditional projects at Ericsson is the cost of handling customer adaptations. Each such adaptation has to be maintained almost as a separate product, which makes them very expensive.

2.2 Streamline Development Process

Streamline Development (SD) is presented in Figure 2. In SD, the projects will be significantly smaller and shorter (at most 3 months long) than in the traditional process described in Section 2.1. This obviously means that the scope of each project will be reduced. A project in SD is initiated when there is a development team available and when there are a suitable number of highly prioritized requirements that can be combined into a requirements package (based on that they fit well together, etc.). The size of such a requirements package is limited by the project length – it should be possible to implement the requirements from the package within the project boundaries, i.e., in less than 3 months.

When developing according to SD, there will always be one (and only one) version of the product at any point of time. When a development project is completed, its outcome is integrated with the current baseline, i.e., the current system version (see Figure 2). Such integration creates a new baseline that the next project will be integrated with. After integration it should be possible to release the system to the customers. Customer adaptation projects are handled in the same as any other projects, i.e., they are integrated into the main product (see Project B in Figure 2). This means that only one latest system version needs to be maintained. However, even though each project produces a new system version that potentially can be released, it does not have to be released to the market. Therefore, in Figure 2, some of the system versions are marked as “potential releases”, which means that they can be released but do not have to be. Such a separation between development and release is a clear difference from the traditional development, where the aim of the project was to release a new version of the system to the market.

Figure 2. Streamline Development process



SD is supposed to address a number of shortcomings of the traditional development described in Section 2.1. Since SD-projects are much shorter than traditional projects, they are less exposed to the risk of changing market demands. Hence, the system developed in such projects is likely to be delivered to the customers before the market demands change. This should reduce the risk of waste due to re-implementation or throw-away. If market demands change, a new project will be started to adapt the current system to the new needs. SD also deals with the problem of the high cost connected with maintaining customer adaptations. In SD, customer adaptations will be integrated into the main product, which should reduce the cost connected with maintaining separate product branches.

3. Related work

Streamline Development aligns well with other modern development practices (e.g., incremental development, evolutionary development, extreme programming). There are a lot of studies [3-5, 11, 13, 17] that discuss advantages and disadvantages of such modern development practices in relation to more traditional software development processes. However, no such studies have reported about SD, for the simple reason that SD is an internally developed process at Ericsson. Nevertheless, several studies reported describe processes that are similar to SD (e.g., processes that focus on customer responsiveness, elimination of rework). Therefore, findings and conclusions from such studies are interesting to investigate and compare with the findings from the study presented in this paper.

One characteristic of SD is the ability to release more often than in traditional processes. The advantages of early and frequent releases are often discussed in the context of incremental software development. An interesting overview of the findings concerning advantages of using incremental approaches is presented by Benediktsson and Dalcher [1]. They argue that incremental software development provides early return on investment to the customers, assures closer fit to the real customer needs, decreases the risk of rework by involving customers early in the development process, decreases the reliance on specialist personnel, lowers the dependency on external deliverables, allows better informed decision making and better understanding of trade-offs, reduces maintenance cost, and enables better resource management.

Another list of advantages of using incremental approach is presented by Graham [4]. The advantages are divided into two groups; advantages for developers and for customers. Among the benefits for developers Graham presents improved team morale, reduced maintenance, reduced risk, easier control of over-engineering, facilitated measurement of productivity, rapid feedback on the correctness of estimations, and a reduced cost of change requests handling. As advantages for customers Graham [4] presents early availability of products, increased confidence in developer, better software quality, longer software lifetime, more flexible options, increased user acceptance, increased system assimilation, increased understanding of requirements and ability of software to meet real, not frozen needs. Apart from advantages, Graham [4] also presents a list of problems connected with incremental development. The list is largely based on [13]. The problems are divided into hardware related problems, life cycle problems, management problems, and financial/contractual problems. In this paper, the life cycle and management problems are of primary interest, and hence focus is put on these. Problems related to the life cycle concern problems with the requirements specification of increments, design integrity, a need for additional testing, and high cost of configuration management. Management problems, on the other hand, regard problems with: co-ordination of teams, controlling releases of increments, and prioritization and scheduling of increments as well as problems with introducing certain "cultural changes" in the organization.

In [9], Middleton discusses the advantages of lean software development. Lean software development is somewhat similar to SD, as it focuses on short cycles between specifying requirements and producing software. By using lean software development, Middleton [9] showed that it has a positive impact on quality of software. However, he also observed that introducing practices like lean software development into organizations that have traditional processes often requires changes in the organization. Such changes are required since traditional "vertical" hierarchy may not apply to the new way of working where closer and more "horizontal" collaboration between roles may be required. Such change may require further changes, e.g., changes in role responsibilities, promotional patterns, etc.

There are also empirical studies that attempt to quantify the gains from using modern development approaches rather than traditional ones. In [3], Dalcher *at al.* present an experiment in which a number of teams developed similar systems using different processes. In this study, the teams used a traditional approach, incremental development, evolutionary development, and extreme programming. The results showed that the use of modern approaches (i.e., incremental development, evolutionary development, and extreme programming) lowered lead-time and improved productivity. Further, there are several "success stories" [5, 11, 17] reported in relation to implementation of modern development practices, similar to SD. These claim that such practices increased customer satisfaction from the product [5, 17], improved product quality [5, 11], and decreased cost [5].

There are also studies reporting about situations where an organization has a working traditional process and considers moving towards a modern process. For example, Boehm and Turner[2] identified barriers, both perceived and actual, that must be considered when introducing agile processes in the place of traditional ones. These barriers are divided into development process conflicts, business process conflicts and people conflicts. When discussing development process conflicts the authors mention that applying agile processes to legacy systems might not always be easy. For example, refactoring, which is a key practice in agile development, is usually very difficult in legacy systems [2]. Among business process issues the authors mention issues connected with human resources (e.g., different positions and different competence may be needed), project progress measurement (e.g., traditional milestones may not always be applicable for agile development), and process standard ratings (e.g., introducing agile processes may affect the rating of a company with respect to certain standards, like ISO or CMMI) [2]. Finally, when investigating people conflicts, the authors [2] mention issues connected with change management (e.g., usually there is some resistance to change among the staff) as well as some logistical issues (e.g., many agile practices require teams to be collocated).[2]

4. Method

This goal of study was an early evaluation of SD applicability for replacing the traditional development process currently used at Ericsson (see Section 2.1 for information concerning the current development practices at Ericsson and Section 2.2 for the description of SD). The main analysis method used in this study was Force Field Analysis (FFA), which is an analysis method used in early evaluations of change suggestions [6]. The FFA method is described in more detail in Section 4.1. The information regarding the impact of SD introduction was collected by performing a number of interviews with persons that represent those roles in the company that will be affected by introducing SD. The findings from the interviews were later post-processed and structured using FFA by the researchers performing this study. More detailed information regarding the data collection and analysis can be found in Section 4.2.

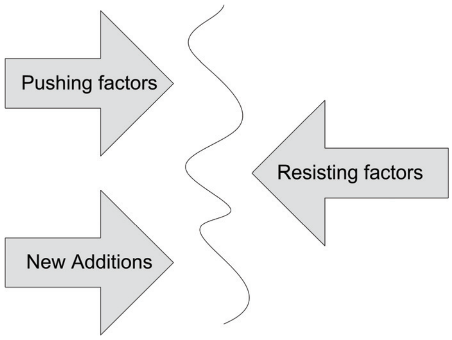
4.1 Force Field Analysis

Force Field Analysis (FFA) is a method for identifying issues that should be taken into account when deciding if to implement a strategic change [6]. FFA is performed by identifying and categorizing data about a change according to the following key factors [6]:

- *Pushing factors*: aspects of the current situation that would aid implementation of the change (e.g., enthusiastic staff)
- *Resisting factors*: aspects of the current situation that would resist the implementation of the change (e.g., lack of management support)
- *New Additions*: additions that must be in place to make the change possible (e.g., new tools must be acquired)

By balancing the Pushing factors and the New Additions with the Resisting factors (as presented in Figure 3), FFA makes it possible to evaluate how successful the implementation of the change is likely to be.

Figure 3. Force Field Analysis



In the study presented in this paper, FFA had to be modified slightly since the aim with the study was not to evaluate the probability of success of the change implementation but rather the change (i.e., the SD introduction) itself. Therefore, the definitions of the three factors taken into account in FFA were slightly modified:

- *Pushing factors*: the advantages of SD. Things that would improve after introducing SD (e.g., customer responsiveness would improve).
- *Resisting factors*: threats connected with introducing SD, i.e., things that would worsen if introducing SD (e.g., the quality of the architecture would deteriorate).
- *Required changes*: Issues that must be resolved and problems that must be overcome before the SD can be implemented (e.g., new competence must be acquired).

By balancing Pushing and Resisting factors it is possible to make an informed decision if the change is worth introducing or not. The information about the Required changes makes it possible to assess the cost of introducing the new processes and to identify issues that must be resolved before the new process can be introduced.

4.2 Data collection and analysis

Before the study, a meeting with two department managers from two Product Development Units (PDUs) at Ericsson was arranged. One of these managers was responsible for the evaluation and potential implementation of SD at his PDU, while the other manager was a representative from the management team at the other PDU. During this meeting it was decided that the opinions regarding benefits and drawbacks of SD as well as opinions about changes that are required for successful implementation of SD (i.e., data necessary for FFA) should be collected by performing interviews. Further, the following questions were addressed during the meeting to clarify the scope of the evaluation:

- *From which perspectives should the evaluation be done?* – a number of different evaluation perspectives can potentially be identified, e.g., product perspective, organization perspective but also some role-related perspective like development or marketing perspective. The reason for discussing this issue was to find out which perspectives are of interest for people making decisions about introducing SD.
- *Who should be involved in the evaluation?* – it was important to identify roles that should be interviewed at each PDU in order to capture the aspects of interest. Persons representing these roles were the main source of information in this study.

During the meeting the following perspectives were found important by the managers:

- *Organizational perspective* – when introducing a new development process (such as SD), it is common that the organization must change. This means that it is not only the size of the project, the order of things done, etc. that must change but also work responsibilities, resource utilization, and so forth. Further, it might be so that SD is more applicable for some PDUs and less for other. To capture these issues the interviews were performed at two different PDUs and questions regarding applicability of SD for each of them were asked.
- *Product perspective* - products may have to change due to the different way of developing the software. Since Ericsson has a product portfolio with products with different characteristics, potentially some of these products are more suited than others for development with SD. To capture such issues, questions concerning the applicability of SD for different products were included.

The managers suggested interviewing all roles that will be affected by introducing SD since this was the only way to provide the holistic picture of the change. This means that a sample representing the major roles in the two PDUs was needed. The following roles were recommended to be included in the interviews:

- *Operative Product Managers* – persons that can be considered as customers' spokespersons in a project. Their major role is to decide what should be done to meet customer needs.
- *System Managers* – persons responsible for refining requirements and deciding how the requirements can be implemented in the system.
- *Designers* – persons responsible for the designing and implementing the system.
- *Testers* – persons responsible for verifying that the product works properly and according to its specification [12].
- *Project Managers* – persons holding the overall responsibility for a project. Project Managers plan, direct, and integrate the work efforts in order to achieve the project's goals [10].
- *Configuration Managers* – persons responsible for build environments, product and document storage systems, creation of deliverable products from developed code and documents, and for managing changes.

In total, 12 interviews were performed (six roles from each of the two PDUs) during which 27 persons were interviewed. Each interview was scheduled for two hours. At each of these interviews, the following persons participated:

- persons representing one role at one of the PDUs (usually two to three persons)
- two researchers, one led the interview and one acted as a secretary

By using this setting, it was possible to get role and PDU specific discussions at the same time. Further, by having more than one person from each role, it was possible to get discussions between the interviewees. The discussions were facilitated by the fact that the interviewees were familiar with each other. By having two researchers present, (one interview leader and one secretary responsible for taking notes), it was possible to keep a good flow of the conversations at the same time as the necessary information was collected.

As the study was meant as exploratory, a certain degree of freedom was given to the interviewers to enable them to ask follow-up questions when some new issues popped-up, to reformulate some of the questions, or to change the order in which questions were asked. Hence, the interviews can be classified as semi-structured [14]. Another option was to use unstructured interviews, which are very common in exploratory studies [14]. However, at the time of the study the topic was largely discussed in the company and some people were very opinionated. By using purely unstructured interviews, there could be a risk that the conversations would focus only on positive or negative aspects of the case. This could have led to collecting only partial information about the studied situation. By adding some structure to the interviews and asking questions that forced the interviewees to focus on different aspects, this risk was reduced.

After each interview, the secretary post-processed the information collected and prepared the final result of the interview. During this process, the information was validated and potential repetitions and ambiguities were removed. As a final validation, the person leading the interview went trough the result and discussed potential discrepancies in views with the secretary. If any discrepancies were found, the issue was discussed until consensus was reached. To facilitate the analysis, the statements describing the opinions of the interviewees were collected in a spreadsheet. This spreadsheet was later used to make the classification of the data easier in the analysis part explained below.

The final grouping of all the identified factors according to FFA (i.e., into Pushing factors, Resisting factors, and Required changes) was done jointly by the researchers involved in the study during a consensus meeting. The major purpose of this activity was to present the data in a usable form so that the findings can be useful for the decision makers at Ericsson. However, some problems were experienced when doing this. First of all, it was problematic to get a good picture of the factors due to the size of the data set (about 300 statements representing opinions of the interviewees regarding the introduction of SD were collected). Another problem was that the opinions were stated at different levels of abstraction, which made them hard to compare. To address these problems the similar statements were grouped and presented on the comparable abstraction level during a consensus meeting.

5. Results

This section presents the most important results of the study. It must be clear that the statements presented in this section are based on the opinions of the interviewees about the potential introduction and usage of SD, i.e., they are not any actual experiences with SD as SD has not yet been implemented. The results presented in this section represent the main findings and are presented according to the classification presented in Section 4.1, i.e., as Pushing factors (Section 5.1), Resisting factors (Section 5.2), and Required changes (Section 5.3).

5.1 *Pushing factors*

The Pushing factors are the positive effects of introducing SD. They are summarized in Table 1.

Table 1. Summary of Pushing factors of SD. See Section 5.1 for details regarding each of the factors.

Factor	Description
Increased motivation of the staff	Shorter projects and immediate feedback will let everyone involved in the development process see the results of their work faster.
Improved productivity	Shorter projects tend to have more stable scope because they are less exposed to the risk of changing market demands. Stable scope minimizes the waste since not that much rework must be done to adapt the system to new requirements.
Increased customer responsiveness	SD makes it possible to release new versions of the system more frequently and hence respond to new customer demands quickly.
Simplified maintenance	In SD there will be only one version of the system produced with no branching for customer adaptations. That will minimize the number of maintained system versions and, therefore, minimize the cost of maintenance.
Improved communication	Projects will be performed by smaller teams. Small team size makes it possible to have more frequent and efficient communication between all team members.
Increased competence level	Fast feedback will facilitate personal development. Project-related competence will improve due to frequent projects.
Improved controllability of the project	Due to smaller project size and scope it will be easier to control the project but also to perform estimations and predictions concerning the project.

A common view among the interviewees was that introducing SD would result in *increased motivation of the staff* involved in the software development. As major reason for that, the interviewees mentioned the positive impact of short projects. In short projects, the end of a project is more “tangible” and thereby people will be more motivated. Another motivating aspect of short projects, according to the interviewees, is that short projects provide almost immediate feedback and make it possible to quickly see the results of their own work.

The interviewees shared an opinion that smaller projects will lead to *higher productivity.* The reason for increased productivity was the fact that short projects are less exposed to the risk of changing requirements. The change of requirements commonly involves waste because already performed work has to be re-made (or removed) to meet changed requirements. According to the interviewees, the more stable scope in small projects minimizes the risk of waste (and hence unnecessary work) and therefore, leads to higher productivity.

Interviewees also mentioned that SD can give a competitive advantage, because it will *increase the customer responsiveness* by delivering the products to the customers fast. The short time between "ordering" functionality and getting it should be very attractive from Ericsson's customers' perspective. The customers are interested in getting new systems with new features fast because these systems often give them an advantage over their competitors.

Another positive aspect of introducing SD is connected with the idea of producing only one version of the system and incorporating customer adaptations into the main product. The interviewees shared the opinion that this should *simplify the maintenance* and reduce its cost. Currently, many product branches have to be maintained (e.g., for each customer adaptation), which is very expensive.

Further, SD should also *improve the communication* within the projects. The development teams will be smaller and the communication within the teams will be easier and more frequent. Therefore, relatively more time can be spent on producing the system than on control and synchronization between development team members. According to the interviewees, the risk of costly misunderstandings (e.g., misunderstandings concerning requirements) will also decrease since the collaboration between different roles will be closer as a result of improved communication.

Due to the improved communication within projects and closer co-operation between different roles, the overall *competence level should increase* since team members will have more chances to learn from each other. The learning process will be further facilitated by small projects. In small projects the time between requirement specification, implementation and testing will be short, which will provide instant feedback to all roles. This will promote and facilitate personal improvement; the competence of individuals should increase because of quick feedback on the quality of their work. Additionally, the projects will be more frequent and the personnel will do their tasks more often. Thus the overall competence related to performing a project should also increase.

The respondents also believed that SD should *improve the controllability* of the projects, mainly because of the reduced size of the projects and better communication within them. Small projects make it possible to make more correct predictions and estimations regarding their lead-time and cost. It is also easier to obtain and maintain an overall picture of what is happening in the project and, therefore, it is also easier to monitor progress.

5.2 Resisting factors

The Resisting factors are the effects of introducing SD that, according to the interviewees, will have a negative impact on software development at Ericsson. They are summarized in Table 2.

The interviewees often mentioned *problems with assuring the quality* of the systems as possible drawbacks if introducing SD. Since SD puts a lot of pressure on keeping the projects short, the interviewees saw a risk that quality assurance may suffer. The interviewees also perceived SD as a very “feature oriented” process – the small projects’ primary goal will be to deliver the functionality as quickly as possible. This attitude may promote short term thinking because of which long term goals, like maintaining the quality of the systems, may suffer.

According to the interviewees, one of the major undesired consequences of the “feature orientation” of SD may be *architecture deterioration.* Since SD is feature oriented, architectural improvements may be neglected when planning which requirements to implement thus causing architecture deterioration. Another problem related to architecture deterioration is that small projects are not appropriate for implementing large architectural improvements. This may make it hard to address architectural deterioration problem after introducing SD.

Some interviewees mentioned *high maintenance cost* as one of the problems. This may sound as a paradox, because maintenance cost decrease was mentioned as one of the positive aspects of introducing SD (see Section 5.1 for details). However, some interviewees noticed that another feature of SD, the ability to release very often, may lead to a situation that there will be a large number of system releases present in the market. Today a new version of the system is available, on average, every year and a half. With 3 months projects, that additionally can be overlapping, a new version of a system can appear much more often. Since customers are not always willing to update their systems so frequently, there is a clear risk of having a large number of releases out in the market that have to be maintained and supported.

Table 2. The summary of Resisting factors of SD. See Section 5.2 for details regarding each of the factors

Factor	Description
Difficulty in assuring quality	SD suggests keeping projects short. Quality assurance activities often tend to suffer when cutting costs in the project.
Long-term architecture deterioration	The main focus of short projects is on adding functionality in an efficient manner. That causes a risk of taking short-cuts and forgetting about issues like system architecture. Furthermore, such architecture deterioration may be hard to address in SD as large architectural changes will be hard to introduce in small projects.
Increased maintenance cost	Frequent releases and lack of possibility to enforce their installation on customers causes the risk of having a number of releases in the market that must be maintained.
Increased backward compatibility cost	Doing frequent releases means that each new release must support data migration from a large number of previously released systems
Dependence on individuals	In small projects the dependence on the skills and knowledge of individuals is large – there is a high risk of loosing competence when loosing a team member.
Applicability to legacy systems	SD must be supported by the architecture of a system. Legacy systems were not architected with SD in mind.
Old (legacy) processes are still used	Even though SD requires rather dramatic changes of current processes, there might be a risk that personnel is used to old ways of working, and continues working in the old way with only slight modifications.

Another problem with having many releases of systems present in the market is the need of keeping *backward compatibility* when releasing a new system version. For example, it happens that the database format changes between different releases. In such a case, an installation program of the new release of the system has to perform data migration to the new data format. Having many releases on the market would mean that an installation program would need to support data migration from a large number of database formats.

The interviewees shared an opinion that the *dependence on individual persons* is a much larger problem in small projects compared to the current large ones. The reason for that is that developer teams will be smaller in SD and therefore, the consequences of someone dropping out from a project are likely to be more severe. Such a loss would be very difficult to compensate in a time of a short project.

Some of the interviewees were concerned about the *applicability of SD to already existing, large software systems*. They argue that the system architecture must support software development in a number of small concurrent projects. Also the dependencies within the system must be well understood to enable efficient project planning (projects should not collide with each other). Such good knowledge of system structure is often a problem in the case of legacy systems. Additionally, some SD concepts also require certain support from the architecture. One example can be the idea of integrating customer adaptations with the main product. Such adaptations are usually made for one particular customer and should not be accessible for other customers. This implies that the architecture should provide means of switching on/off certain functionalities. Since legacy systems were not architected with this SD specific requirements in mind, there is a risk that their architecture is not suitable for SD.

Apart from legacy systems the respondents also discussed *legacy processes*. When changing to a new way of working, new processes that will fit this new way of working must be developed as a replacement for the old processes. Even though everyone agreed that this is the way to introduce the change, several interviewees saw a clear risk that it will be tempting to try to use currently existing processes in SD. The old processes are not meant to be used in an environment like with SD, which means that the usage of old processes may affect SD negatively.

5.3 Required changes

Required changes are issues that, according to our interviewees, must be tackled before introducing SD in order to make the introduction of SD successful. The Required changes are summarized in Table 3.

Table 3. The summary of Required changes connected with introducing SD. See Section 5.3 for details regarding each of the factors.

Suggestion	Description
Introduce continuous requirements management	The inflow of new requirements is constant. The requirements must be continuously prioritized because project planning depends on that. Also the requirements prioritization depends on the current baseline (system version). Therefore it must be updated every time the baseline changes, i.e., every time a new system version is released
Assure effective pre-project planning	A number of concurrent projects must be coordinated so that there will be no collisions between them. Also many things (e.g., allocation of some resources) that can now be planned in the project will have to be planned before project starts.
Increase the efficiency of the installation procedure	Efficient installation procedure will make testing more efficient. Also frequent releasing and deployment will make the efficiency of the installation process more important.
Increase testing efficiency	Testing is likely to be one of the productivity bottlenecks in SD. Therefore new methods for improving the efficiency of testing will be necessary.
Keep people in the same product line	It will be important not to move people between different products as in short projects the learning effect will have large impact on productivity.
Assure understanding of dependencies within the systems	To enable coordination of small projects it is important to fully understand the dependencies between all components of the system. That will make it possible to avoid collisions between projects.
Improve architecture	Some required architecture improvement work needs to be performed before SD is introduced to make the system architecture fit SD. It also seems that it will be harder to address large issues like architecture changes within SD.

When discussing the required changes connected with introducing SD, most of the interviewees stressed the need of introducing *continuous requirements management*. This means that requirements must be packaged and prioritized so that it is always clear what a new small project should do. Packaging requirements means that it is necessary to find chunks of requirements that fit well together and, at the same time, are of suitable size to implement in the small projects. When having such packages, it is important to continuously prioritize these in order to always choose the requirements packages most suitable for implementation (considering e.g. dependencies, cost, and market window). This activity must be continuous as new requirements are continuously incoming. Additionally, since requirements are usually prioritized towards the current baseline (the characteristics of the current system version) their prioritization should be updated every time the baseline changes (i.e., when some new project is integrated). It is important to do so because changing the baseline may actually change the importance of a requirement.

The interviewees also pointed out that the *effectiveness of pre-project planning must be assured*. When planning a new project it is necessary to assure that there will be no clashes with other ongoing projects. This planning must also take into account that some decisions must also be made upfront, before the project starts. For example, if the project requires an access to a customer site (e.g., for testing purposes) then appropriate arrangements usually must be made well before starting the project.

SD enables frequent releases of the new system versions. According to the interviewees in their large telecommunication systems it will be very important to assure an *efficient installation procedure*. If the deployment on customer site is to be done more often, an inefficient installation procedure may become a major cost. SD will also require frequent installations in the test plant. Therefore, an efficient installation will facilitate and decrease the cost connected with testing of the systems.

In general, the respondents found it very important to *increase testing efficiency*. They considered testing efficiency as one of the possible productivity bottlenecks in SD. One major reason for that is that small and frequent projects will mean that testing will be done more often but there will be shorter time to do it. Since there is some overhead connected with testing (e.g., regression testing must be repeated for each system increment) the impact of testing on project cost is likely to increase. Therefore, the requirements for testing efficiency will be much higher in SD.

Since individual capabilities are likely to play larger role in small projects (see Section 5.2), the interviewees found it very important to *keep people in the same product line*. Moving people continuously between different products may result in that their learning process will affect the overall project productivity negatively. One of the goals with SD is to increase the overall productivity of software development by having it done in small, efficient, and specialized teams of developers. This advantage may be lost if people will be continuously moved from one team to another, as it is done at the moment.

The interviewees also found it important to fully *understand the dependencies between system components.* They suggested creating an anatomy plan [16] in which the interdependencies between different system components would be described. Without the knowledge about these interdependencies it is impossible to coordinate a number of concurrent projects and ensure that they will not collide with each other. The knowledge about interdependencies in the system is also necessary for requirements prioritization and packaging, i.e., requirements affecting the same parts of the system should in most cases be implemented together.

It is also very important to focus on *architecture improvements* before implementing SD. Two reasons why this issue is important were identified. First, one of the common opinions of our interviewees was that SD requires support from the system architecture. The architecture of the system must make it possible to perform concurrent development projects and maintain a single system version. A second reason why architectural issues are important to address before implementing SD is that it apparently will be very hard to address these issues after SD is introduced. “Feature orientation” and limited scope of short projects, together with the necessity of maintaining one product version only, will make it very difficult to address some large and fundamental issues, like for example architecture change or improvement.

6. Discussion

In this section, a discussion concerning the study is presented. Section 6.1 focuses on the discussion of the major findings from this study. In Section 6.2 the findings from this study are compared with the findings of other researchers. Section 6.3 discusses different validity issues connected with the study.

6.1 Discussion regarding the results

The goal of the study was to evaluate the applicability of SD at Ericsson. Unfortunately, all details of this evaluation cannot be revealed due to confidentiality reasons. However, as can be noticed in the discussions in previous sections, the overall attitude towards SD was positive as there was a rather large agreement between the interviewees when it comes to the positive effects of introducing SD. These positive effects were very similar to the intentions of SD (see Section 2.2), which indicates that SD is likely to achieve the goals it was designed to achieve (e.g. improved customer responsiveness and productivity). The fact that the interviewees were positive about the new process is a good forecast for the success of the SD implementation as staff's resistance to change is often considered a problem when new processes are introduced [2]. Additionally, by looking at Resisting factors and Required changes, it can be seen that many of the Required changes actually either address or at least reduce the importance of some of the problems classified as Resisting factors. For example, insufficient knowledge about legacy systems and their architecture deficiencies were considered as one problem with applying SD to legacy systems (see Section 5.2 for details). By suggesting an anatomy plan and architecture improvement work before introducing SD, the interviewees addressed this problem and minimized its impact. Therefore, it seems that by taking some preventive actions the risks described as Resisting factors can be reduced.

The analysis presented in this paper can be considered an early evaluation or the first step in the evaluation of SD. The main focus was to identify and classify factors that should be taken into account when making the final decision regarding the introduction of SD. Some general conclusions regarding the usefulness of SD can be drawn from this study. They are, however, by no means sufficient for making the final decision. Nevertheless, the study was considered useful from an SD evaluation perspective by decision makers at Ericsson. It made it possible to check if, in general terms, SD seems promising and if there are no obvious obstacles for introducing it at Ericsson. Based on this information, it was possible to decide if it makes sense to put more time and effort into further development and evaluation of the SD idea. One of the good things about performing evaluations in the way presented in this study (see Section 4 for details regarding the method) is the limited amount of time necessary to perform it. For the case study presented in this paper, it took about 2 working weeks for 3 persons to perform interviews, collect and analyze the data, and present the findings in a usable form to the decision makers. Therefore, the low cost combined with the effectiveness makes this evaluation method a useful tool in early evaluations of new development processes.

As a natural next step in the evaluation of SD we suggest performing a detailed cost-benefit analysis of introducing SD, in which the impact of each of the Pushing factors, Resisting factors, and Required changes should be quantified. By balancing the benefits from introducing Streamline Development (i.e., Pushing factors) with the costs associated with risks (i.e., Resisting factors) and changes (i.e., Required changes) it will be possible to make the final decision regarding the introduction of SD. As it can be noticed, further analysis can be still performed in the frame of Force Field Analysis. This indicates that Force Field Analysis can be a very helpful analysis method in evaluations of different change suggestions. In this study we successfully used it in a process change evaluation but it seems to be also applicable for an evaluation of any other changes e.g., technology change.

6.2 Comparison with other studies

As indicated in Section 3, SD has many similarities with many other modern software development practices. Therefore, it is interesting to compare the results obtained in this study with the findings of other researchers that evaluated such modern development processes and compared them with more traditional ones. Not surprisingly, like many others (e.g., [1, 5, 17]) the interviewees in this study also observed the positive impact of increased customer responsiveness obtained by introducing practices like SD. Similarly to Graham [4] the interviewees also noticed that the cost connected with change request handling should be reduced, mainly because the number of change requests should decrease due to shorter projects. Another observation from Graham [4] that was confirmed is the positive impact of shorter projects on the motivation of the staff. A common conclusion in many studies is that all these positive effects lead to higher productivity and shorter lead time [3, 5]. The majority of our interviewees expressed that they expect this also from SD. Also the negative effects of introducing SD described in this study have been recognized in other studies. For example, coordination problems and requirements prioritization challenges are mentioned in [4]. Another problem recognized by other researchers is the problem with applying new processes to legacy systems [2], which was also identified in this study. The difficulty to implement large and complex features in small projects, mentioned by the interviewees, was also recognized in [11].

There are also findings reported by other researchers that were not fully confirmed in this study. For example, many studies [5, 9, 11] discuss the positive impact of modern practices on system's quality. In this study, on the other hand, the interviewees at Ericsson were concerned about some quality issues, e.g., they stressed that the "feature orientation" of SD may lead to the degradation of the architecture. However, the differences may be explained by the fact that quality can be understood in different ways. In [5] the quality is defined in terms of cost and customer satisfaction. Our interviewees actually also expected this to improve. In [9, 11] the authors define quality in terms of defects. This quality view, i.e., the impact of SD on the number of defects in systems, was not mentioned by the interviewees and it is hence impossible to say if this finding was verified or not.

Another issue reported by other researchers [1, 4] that is not fully supported by our findings, is the reduction of maintenance cost. The interviewees of this study argued that releasing the software frequently, which is necessary to achieve customer responsiveness, can have negative impact on the maintenance cost. This is because frequent releases may result in many different system versions on the market that have to be maintained (see Section 5.2 for details). On the other hand, some interviewees actually agreed that maintenance costs would decrease because only one branch of the product will exist. Therefore, since it is hard to say which of these two issues will have a predominant impact on the cost of maintenance, it is not possible to determine if the maintenance cost will increase or decrease as a result of introducing SD.
It can be observed that most studies in the area (see Section 3 for examples) focus on a situation, in which a company can simply choose between traditional and some new development method. In practice, however, organizations most often have a traditional process already in place. Such perspective may change the importance of different factors. It also adds a new angle, the change implementation process, which normally is not accounted for when simply comparing two different development processes. At the same time, it is an important factor in making an informed decision regarding a process change. In this study, this factor was taken into account (in FFA this factor is represented by Required changes). This is one of the strengths of the presented study.

6.3 Validity

This case study was performed in a concrete industrial setting. Therefore, it may seem hard to generalize the findings to other contexts. However, the actual findings from this study, i.e., Pushing factors, Resisting factors, and Required changes, may be of interest for other companies that consider moving from a traditional to a modern approach where projects are shorter and releases are more frequent. For example, it is highly likely that large fundamental changes, like architectural changes, will be hard to address in small projects. Also things like continuous management of requirements must always be in place in order to make the best use of short projects and to reach a high level of customer responsiveness. Even though the individual issues are likely to be of different importance for different companies, they can still help them when identifying threats, opportunities and costs of changing the development process. Also, it seems that the method used for evaluation (see Section 4) seems to be useful in evaluations of new development processes. Therefore, it can be seen as one of the contributions of this paper that can be interesting for broader audience. Another problem with this kind of studies is that statements, opinions, judgments, etc. may be misinterpreted. However, all three researchers have a long history of collaboration with Ericsson which means that all three are knowledgeable in processes, company culture, etc. This fact reduces the threat that things have been misinterpreted and hence increases the likelihood that the correct issues actually are reported. This experience from Ericsson was also very helpful when performing interviews since the previous knowledge and experiences made conversations easier and more effective.

7. Conclusions

The goal of this study was to perform an early evaluation of Streamline Development's applicability as a replacement of the current development practices at Ericsson. Streamline Development is a new process created at Ericsson AB with the main purpose of improving customer responsiveness. The evaluation was conducted by interviewing 27 persons from two Product Development Units at Ericsson. The interviewees represented roles in the company that will be affected by changing the development process. To analyze the findings from these interviews, an adaptation of the Force Field Analysis [6] method was suggested. In this method all the opinions were classified as Pushing factors (issues that would improve if a new process was introduced), Resisting factors (issues that would worsen if a new process was introduced), or Required changes (changes that must be made to prepare organization and products for a new development process).

The overall conclusion from the study was that Streamline Development seems promising. Its main goals (i.e., improvement of customer responsiveness and productivity) were recognized as Pushing factors, which indicates that they are likely to be achieved. On the other hand, some issues were identified and classified as Resisting factors. However, many of these were addressed by suggestions of Required changes. This indicates that certain actions can be taken in order to decrease the negative impact of those Resisting factors.

An additional conclusion from this study concerned the method used to evaluate Streamline Development (i.e., collect the opinions regarding the new process by performing interviews and structure them using Force Field Analysis). This method was found to be a very useful tool for evaluating the applicability of Streamline Development. It provided information that was considered valuable by decision makers at Ericsson. Due to the relatively low cost of performing such evaluation, it seems to be especially useful as a method for early evaluations of new process ideas.

8. Acknowledgments

We would like to thank Wayne Strong for guiding us through the world of strategy evaluations. We would also like to thank the managers and the interviewees at Ericsson for their active participation in the study.

This work was partly funded by The Knowledge Foundation in Sweden under a research grant for the project "Blekinge - Engineering Software Qualities (BESQ)" (http://www.bth.se/besq).

9. References

[1] O. Benediktsson and D. Dalcher, Effort estimation in incremental software development. *Software, IEE Proceedings-*, 150 (2003), 351-358.

[2] B. Boehm and R. Turner, Management challenges to implementing agile processes in traditional development organizations. *IEEE Software*, 22 (2005), 30-40.

[3] D. Dalcher, O. Benediktsson, and H. Thorbergsson, Development life cycle management: a multiproject experiment. *Fibres and Optical Passive Components, 2005. Proceedings of 2005 IEEE/LEOS Workshop on*, (2005), 289-296.

[4] D.R. Graham, Incremental development and delivery for large software systems. *Software Prototyping and Evolutionary Development, IEE Colloquium on*, (1992), 2/1-2/9.

[5] B.D. Jensen, A software reliability engineering success story. AT&T's Definity PBX. *Software Reliability Engineering, 1995. Proceedings., Sixth International Symposium on*, (1995), 338-343.

[6] G. Johnson, K. Scholes, and R. Whittington, Exploring corporate strategy, Financial Times Prentice Hall, Harlow, (2005).

[7] G. Kotonya and I. Sommerville, Requirements engineering: processes and techniques, John Wiley, Chichester, (1998).

[8] A. MacCormack, C.F. Kemerer, M. Cusumano, and B. Crandall, Trade-offs between productivity and quality in selecting software development practices. *IEEE Software*, 20 (2003), 78-85.

[9] P. Middleton, Lean software development: two case studies. *Software Quality Journal*, 9 (2001), 241-252.

[10] J.M. Nicholas and J.M. Nicholas, Project management for business and technology: principles and practice, Prentice Hall, Upper Saddle River, N.J.; London, (2001).

[11] J.r. Niels, Putting it all in the trunk: incremental software development in the FreeBSD open source project. *Information Systems Journal*, 11 (2001), 321-336.

[12] S.L. Pfleeger, Software engineering: theory and practice, Prentice Hall, Upper Saddle River, (2001).

[13] F. Redmill, Incremental delivery-not all plain sailing (software development). *Software Prototyping and Evolutionary Development, IEE Colloquium on*, (1992), 6/1-6/6.

[14] C. Robson, Real world research: a resource for social scientists and practitioner-researchers, Blackwell Publishers, Oxford, UK; Madden, Mass., (2002).

[15] I. Sommerville, Software engineering, Addison-Wesley, Boston, Mass., (2004).

[16] L. Taxén and J. Lilliesköld, Manifesting shared affordances in system development: The system anatomy, *The 3rd International Conference on Action in Language, Organisations and Information Systems*, Limerick, Ireland, (2005), 28-47.

[17] P. Tran and R. Galka, On incremental delivery with functionality. *Computers and Communications, 1991. Conference Proceedings., Tenth Annual International Phoenix Conference on*, (1991), 369-375.

ABSTRACT

Software development productivity can be improved by introducing improvements in many areas. In this thesis we investigate technology and process driven productivity improvements, i.e., productivity improvements that have sources in changes of technologies or in changes in development processes. The technology driven productivity improvement discussed in this thesis is the change of server platform from a standard general purpose platform to a specialized fault-tolerant platform. We discuss productivity implications of introducing such a platform as well as suggest ways of making the platform introduction process cost efficient. The process changes, which we discuss in this thesis, include improvements of fault detection processes as well as changes of the entire development process.

We analyze the implications of introducing new technology by performing case studies, in which we describe, analyse, and quantify the impact of the new platform on software development productivity. We show that there is a significant productivity decrease connected with introducing a new platform. We also show that the initial low productivity can be overcome by experience and maturity. We suggest a number of improvements for both the platform introduction process and the mature development on the specialized platform. Since some productivity decrease after introducing new technology is to a large extent unavoidable, we look for ways of minimizing it. We show that it is possible to minimize it by introducing the specialized platform gradually. We present an example of a hybrid architecture, which combines the specialized and the standard platforms. We show that such architecture is able to provide good technical characteristics for a significantly lower cost as compared to developing the entire application on the specialized platform.

As a process improvement suggestion we propose introducing fault prediction models with the goal of increasing the efficiency of fault detection. We suggest and evaluate several such models that are available at different stages of a software development process. The models are evaluated using data from a number of large software systems. Their predictions are also compared with the predictions made by human experts. We show that introducing our fault prediction models is likely to result in an improvement of fault detection efficiency. Another process related productivity improvement suggestion evaluated by us is the change of the development process. We present a case study in which we evaluate a new process concept. One of the goals of that process is to improve the company's productivity.

ISSN 1653-2090
ISBN 91-7295-090-0

2006:05