The Impact of Software Reuse and Incremental Development on the Quality of Large Systems

Parastoo Mohagheghi

Doctoral Thesis

Submitted for the Partial Fulfillment of the Requirements for the Degree of

Philosophiae Doctor

Department of Computer and Information Science
Faculty of Information Technology, Mathematics and Electrical Engineering
Norwegian University of Science and Technology

July 2004

ISBN printed 82-471-6408-6
ISBN electronic 82-471-6407-8
ISSN 1503-8181
NTNU 2004:95

Printed in Norway by NTNU Trykk, Trondheim

Abstract

Incremental development, software reuse, product families and component-based development seem to be the potent technologies to achieve benefits in productivity, quality and maintainability, and to reduce the risks of changes. These approaches have multiple and crosscutting impacts on development practices and quality attributes. Empirical studies in industry answer questions about why and when certain approaches are chosen, how these are applied with impact on single instances and how to generalize over classes or systems. Large, long-lived systems place more demands on software engineering approaches. Complexity is increased, systems should have the correct subset of functionality and be maintainable for several years to return the investment.

The research in this thesis is based on several empirical studies performed at Ericsson in Grimstad, Norway and in the context of the Norwegian INCO project (INcremental and COmponent-Based Software Development). A product family with two large-scale products that have been developed incrementally is described. The work aimed to assess the impact of development approaches on quality and improve the practice in some aspects. The research has been a mixed-method design and the studies use qualitative data collected from sources such as web pages, text documents and own studies, as well as quantitative data from company's data repositories for several releases of one product. The thesis contains five main novel contributions:

C1. Empirical verification of reuse benefits. Quantitative analyses of defect reports, change requests and component size showed reuse benefits in terms of lower defect-density, higher stability between releases, and no significant difference in change-proneness between reused and non-reused components.

C2. Increased understanding of the origin and type of changes in requirements in each release and changes of software between releases. A quantitative analysis of change requests showed that most changes are initiated by the organization. Perfective changes to functionality and quality attributes are most common. Functionality is enhanced and improved in each release, while quality attributes are mostly improved and have fewer changes in form of new requirements.

C3. Developing an effort estimation method using use case specifications and the distribution of effort in different phases of incremental software development. The estimation method is tailored for complex use case specifications, incremental changes in these and reuse of software from previous releases. Historical data on effort spent in two releases are used to calibrate and validate the method.

C4. Identifying metrics for a combination of reuse of software components and incremental development. Results of quantitative and qualitative studies are used to relate quality attributes to development practices and approaches, and to identify metrics for a combination of software reuse and incremental development.

C5. Developing a data mining method for exploring industrial data repositories based on experience from the quantitative studies.

This thesis also proposes how to improve the software processes for incremental development of product families. These are considered minor contributions:

C6a. Adaptation of the Rational Unified Process for reuse to improve consistency between practice and the software process model.

C6b. Improving techniques for incremental inspection of UML models to improve the quality of components. A controlled industrial experiment is performed.

Acknowledgements

This thesis is part of the INCO project (INcremental and COmponent-based development) done jointly by the University of Oslo (UiO) and the Norwegian University of Science and Technology (NTNU). The doctoral work is financed by INCO via the UiO for two years, the Simula Research Laboratory in Oslo for one year and NTNU for 6 months for duties as a research assistant. The fieldwork was done at Ericsson in Grimstad, Norway.

First of all, I thank my supervisor, Reidar Conradi, for his continuous support and advice during this doctoral work, and his comments on the papers and drafts of this thesis. His engagement and knowledge has inspired me a lot. I also thank other members of the INCO project and the software engineering group at the Department of Computer and Information Science at NTNU for their support and comments on various papers, especially Dag Sjøberg and Magne Jørgensen at the Simula Research Laboratory, and Tor Stålhane and Letizia Jaccheri at NTNU.

I acknowledge the editing work by Stewart Clark at NTNU and the comments by Magne Syrstad at NTNU on Part I of this thesis. I have collaborated with my supervisor, other researchers and several students during the work on this thesis, as reflected in the author list of the papers in Part II. I acknowledge their contributions and thank all of them.

I have been a part-time employee at Ericsson during this work and the organization has given me the privilege of performing empirical studies and sharing the results. I am deeply grateful for this opportunity and thank all the former colleagues who have supported me.

Finally, I thank my husband, Esmaeil, and our son, Arash, for their love and support.

NTNU, July 7, 2004

Parastoo Mohagheghi

Contents

List of Figures

List of Tables

Abbreviations

AF	Adaptation Factor (COCOMO)
AOP	Aspect-Oriented Programming
ARS	Application Requirement Specification (Ericsson)
CASE	Computer-Aided Software Engineering
CBD	Component-Based Development
CBSE	Component-based Software Engineering
CCM	CORBA Component Model
COCOMO	Constructive Cost Model
CORBA	Common Object Request Broker Architecture
COM	Component Object Model (Microsoft)
COTS	Commercial-Off-The-Shelf
CM	Configuration Management
CR	Change Request (Ericsson)
DCOM	Distributed Component Object Model (Microsoft)
EJB	Enterprise Java Beans (Sun)
FIS	Feature Impact Study (Ericsson)
FODA	Feature-Oriented Domain Analysis
GPSN	Gateway GPRS Support Node
GPRS	General Packet Radio Service
GQM	Goal/Question/Metric
GSM	Global System for Mobile communications
GSN	GPRS Support Node
GUI	Graphical User Interface
HiA	Agder University College
IDL	Interface Definition Language
IP	Internet Protocol
KLOC	Kilo Lines of Code
LOC	Lines of Code
MDA	Model Driven Architecture (OMG)
MS	Mobile Station (Ericsson)
NTNU	Norwegian University of Science and Technology
OMG	Object Management Group
OORTs	Object-Oriented Reading Techniques
ORB	Object Request Broker
OS	Operative System
OSS	Open Source Software
PH	Person-Hours
QoS	Quality of Service
R&I	Review & Inspections (Ericsson)
ROI	Return On Investment
RUP	Rational Unified Process
SEI	The Software Engineering Institute (SEI) at Carnegie Mellon University
SGSN	Serving GPRS Support Node

SPI	Software Process Improvement
SoC	Statement of Compliance (Ericsson)
SQL	Structured Query Language
TG	TollGate (Ericsson)
TR	Trouble Report (Ericsson)
UCP	Use Case Points
UCS	Use Case Specification (Ericsson)
UiO	University of Oslo
UML	Unified Modeling Language
UMTS	Universal Mobile Telecommunications System
W-CDMA	Wideband Code Division Multiple Access
WPP	Wireless Packet Platform (Ericsson)
XP	eXtreme Programming

1 Introduction

In this chapter, the background to the research and the research context is briefly presented. The chapter also describes research questions, research design and the claimed contributions. Also, the list of papers and the thesis outline are presented.

1.1 Problem Outline

As a considerable portion of software projects miss schedules, exceed their budgets, deliver software with poor quality and even wrong functionality, researchers and industry are seeking methods to improve productivity and software quality. Software reuse has been proposed as a remedy for decades. Reuse is an umbrella concept and the reusable assets can take many forms: component libraries, free-standing COTS (Commercial-Off-The-Shelf) or OSS (Open Source Software) components, modules in a domain-specific framework, or entire software architectures and their components forming a product family. Component-Based Development (CBD) provides techniques for the decomposition of a system into independent parts conforming to a component model, thereafter composition of systems from pre-built components (either COTS or developed in-house). CBD advocates the acquisition and integration of reusable components. Components are more coarse-grained than objects, which may be an advantage in retrieving and assembly, and they conform to a component model, which facilitates composition. Incremental development is chosen to reduce the risks of changing requirements or environments. The basic idea is to allow the developers to take advantage of what was being learned during the development of earlier, deliverable versions of the system and to enhance the system in accordance with the demands of users or the market.

While several technologies for software reuse, CBD and incremental development have emerged in recent years, there are still many open questions. The impact of these technologies on software quality, schedule or cost should be analyzed. The risks associated with single technologies and their combinations should be identified. Case studies in industry play an important role in all these steps, since technologies should be studied in a real context, combined with industrial practices and tuned to fit the context. Incremental development, CBD and product family engineering are especially relevant for developing large, long-lived software systems. In these systems, the scope is gradually covered (and discovered), complexity is handled by decomposition into independent units and thereafter composition, and systems may share software architecture and some core assets to reduce cost and increase productivity. Empirical studies on large systems may answer questions on how certain technologies are applied and adapted for large-scale development.

1.2 Research Context

The research in this thesis uses the results of quantitative and qualitative empirical studies of a large-scale telecom system developed by Ericsson in Grimstad, Norway. The General Packet Radio Service (GPRS) support nodes enable high-speed wireless Internet and data communications using packet-based technology. The two main nodes are the Serving GPRS Support Node (SGSN) and the Gateway GPRS Support Node (GGSN). Ericsson has developed two products to deliver GPRS to the GSM (Global System for Mobile communication) and UMTS (Universal Mobile Telecommunications System) networks. The SGSN nodes in the two networks share software architecture, a component framework, many other core assets and development environment in a product family approach. The systems are developed incrementally and several releases are delivered in over five years. The software process has been an adaptation of the Rational Unified Process (RUP). Quantitative data from SGSN for GSM releases, as well as qualitative data gathered from web sites, project documents and own studies of the software process model and the practice of software development are used in the analysis and interpretation phases. Some of the results are used to build a model of the impact of development approaches on quality attributes.

I had over three years of experience working with software architecture, design, programming and software process adaptation in the GPRS projects prior to this doctoral work. Ericsson has supported us in collecting data and performing studies. However, Ericsson stopped development in Grimstad in 2002, and the organizational noise around reorganizations and outsourcing has influenced the study in the sense that the original focus on software process improvement for reuse could not be followed.

Some characteristics of the development projects and the system in the telecom domain are:

- Personnel turnover has traditionally been small and Ericsson has had access to experienced staff with domain knowledge in the development phase. However, Ericsson has reduced its staff by almost 60% in the last three years.
- Quality (or non-functional) requirements such as performance, reliability, availability, maintainability and evolvability are of great importance for nodes in a network. On the other hand, there are no direct user interfaces and no requirements related to these. Other requirements such as safety and security are defined differently in each domain.
- The system is decomposed at the highest level into subsystems of large granularity. Subsystems are units of reuse. Each subsystem contains a number of function blocks, which are tightly coupled inside and are mapped to components. Components are mostly developed in-house.

Other characteristics of the system as a large-scale system are:

- Large systems face challenges in all phases of development that small systems do not, such as difficulties in iteration planning, complexity of design,

integration and test in large, and maintenance costs. Therefore, development technologies should be verified for development in large.
- Companies are increasingly using mainstream development methods, tools, standards, programming languages and software processes.
- Outsourcing is a new trend in industry, but the success depends on the task and the competence of the company taking over. Ericsson could outsource maintenance of previous releases since after re-organizations, experienced personnel were hired by another company based on an agreement with Ericsson.
- Large companies start joint projects for developing new standards, tools and processes.

Outsourcing, joint projects and mainstream development environment lead to a more standardized view of software development. Thus, assessing common development approaches in case studies are interesting for a broader audience than before, both for generalization and to understand variations and adaptations in single instances.

The work for the thesis is done in the context of the INCO (INcremental and COmponent-based Software Development) project, which is a Norwegian research project from 2001 to 2004. INCO defines the following four project goals:

G1. Advancing the state-of-the-art of software engineering, focusing on technologies for incremental and component-based software development.
G2. Advancing the state-of-the-practice in software-intensive industry and for own students, focusing on technologies for incremental and component-based software development.
G3. Building up a national competence base around these themes.
G4. Disseminating and exchanging the knowledge gained.

The purpose of this thesis is:

- Advancing the state-of-the-art of software engineering by assessing existing theories, exploring aspects that are insufficiently empirically studied before and generalizing the results when possible.
- Advancing the state-of-the-practice of software reuse in incremental development of a large telecom system by proposing improvements to the development processes.

1.3 Research Questions

The goal of the research is to explore the impact of software reuse and incremental development on quality, where quality refers to both software process quality and software product quality, for a large-scale (telecom) system, and to improve the practice based on the gained knowledge. The research questions are defined to be:

RQ1. *Why a reuse program is initiated and how is it implemented?*
RQ2. *What is the impact of software reuse, CBD and incremental development on the quality? The impact of development approaches on product quality metrics and on project attributes such as schedule or effort are sought.*
RQ3. *How to improve the practice of incremental development of product families in some aspects?*

1.4 Research Design

Empirical studies may be performed quantitatively, qualitatively or in combination. The choice of approach affects data collection, data analysis and discussions of validity. This study has been a combination of qualitative and quantitative studies. The first phase as shown in Figure 1-1 is dominated by qualitative studies of the software process model and the development practice. A controlled experiment on inspection techniques and a survey on software reuse are also performed. The second phase is dominated by quantitative studies of Trouble Reports (TRs), Change Requests (CRs), effort distribution and Use Case Specifications (UCSs). In the third phase, the results of qualitative and quantitative studies and internal measures gathered by the company are integrated in three aspects: metrics, developing a data mining method for exploring industrial data repositories and assessing development approaches. The mix of quantitative and qualitative methods has several purposes:

- Expanding our understanding when moving from one study to the other.
- Triangulation or confirming the results of one study by other studies.
- Answering questions that are not possible to answer by a single-method design, such as the impact of development approaches in several dimensions.
- Performing studies that are both exploratory and confirmatory.
- Taking benefit of all available data; both quantitative data such as TRs, as well as qualitative data such as process descriptions and project reports.

Figure 1-1 shows the studies performed, their date and sequence, type of studies and the relations to papers and contributions. The papers numbered from P1 to P13 are listed in Section 1.5 and the contributions are described in Section 1.6.

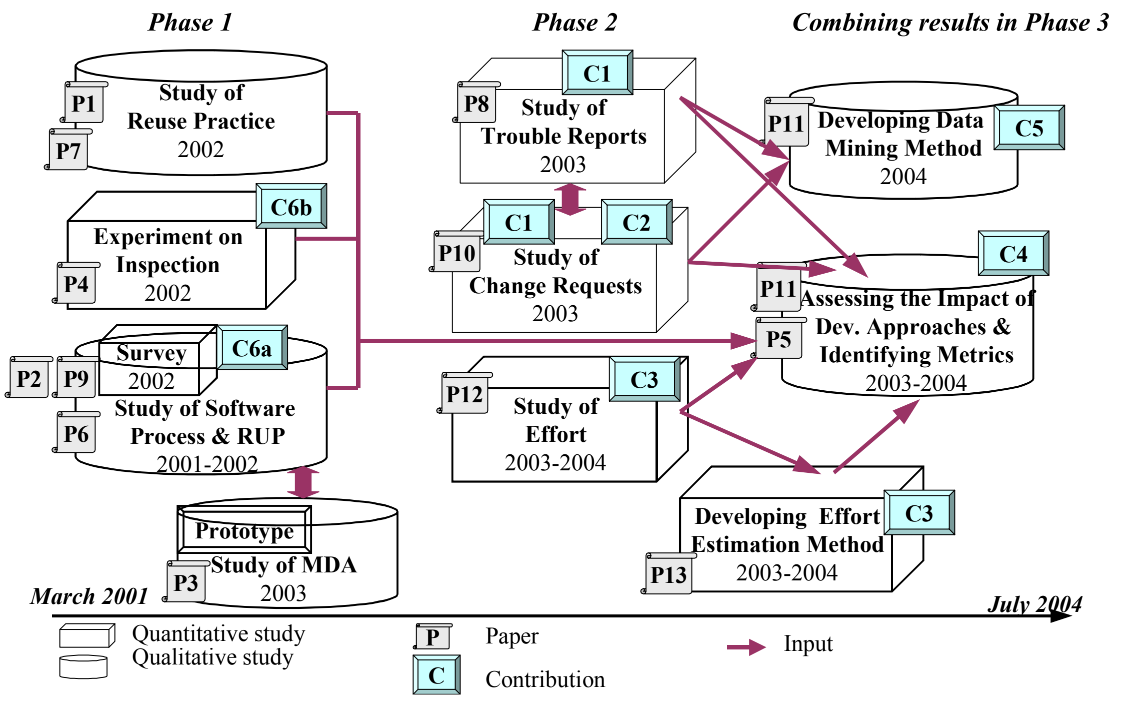

Phase 1
Phase 2
Combining results in Phase 3
P1
P7
Study of Reuse Practice 2002
C6b
Experiment on Inspection 2002
P4
P2
P9
Survey 2002
C6a
Study of Software Process & RUP 2001-2002
P6
Prototype
P3
Study of MDA 2003
March 2001
C1
P8
Study of Trouble Reports 2003
C1
C2
P10
Study of Change Requests 2003
P12
C3
Study of Effort 2003-2004
P11
Developing Data Mining Method 2004
C5
C4
P11
Assessing the Impact of Dev. Approaches & Identifying Metrics 2003-2004
P5
Developing Effort Estimation Method 2003-2004
C3
P13
July 2004
Quantitative study
Qualitative study
P
Paper
C
Contribution
Input

Figure 1-1 Studies and their contributions

The research methods for each research question have been:

RQ1 is answered by *qualitative analysis* of the practice and the software development process, a small survey and other knowledge gained from *quantitative studies*.

RQ2 is answered by *mining and quantitative analysis* of data stored in different company data repositories, the company's internal measures and reports, and qualitative observations. A model has been developed of the impact of development approaches on some quality attributes.

RQ3 is answered by *combining results* of **RQ1** and **RQ2,** and by proposing improvements in RUP, estimation method, inspection techniques and metrics. A research method for mining industrial data repositories is also proposed.

1.5 Papers

[P1] Mohagheghi, P., Conradi, R.: Experiences with Certification of Reusable Components in the GSN Project in Ericsson. In Judith Stafford et al. (Eds.): Proc. 4th ICSE Workshop on Component-Based Software Engineering: Component Certification and System Prediction (ICSE'2001), Toronto, May 14-15, 2001, pp. 27-31. *SU-report 6/2001.* Main author.

[P2] Mohagheghi, P., Conradi, R., Naalsund, E., Walseth, O.A.: Reuse in Theory and Practice: A Survey of Developer Attitudes at Ericsson. NTNU- Department of Computer and Information Science (IDI) PhD Seminar, May 2003. Main author.

[P3] Mohagheghi, P., Nytun, J.P., Selo, Warsun Najib: MDA and Integration of Legacy Systems: An Industrial Case Study. Proc. of the Workshop on Model Driven Architecture: Foundations and Applications, June 26-27, 2003, University of Twente, Enschede, The Netherlands (MDAFA'03). Mehmet Aksit (Ed.), 2003, *CTIT Technical Report TR-CTIT-03-27*, University of Twente, pp. 85-90. Main author.

[P4] Conradi, R., Mohagheghi, P., Arif, T., Hegde, L.C., Bunde, G.A., Pedersen, A.: Object-Oriented Reading Techniques for Inspection of UML Models -- An Industrial Experiment. In Luca Cardelli (Ed.): Proc. European Conference on Object-Oriented Programming (ECOOP'03), Darmstadt, July 21-25, 2003, *Springer LNCS 2743*, pp. 483-501, ISSN 0302-9743, ISBN 3-540-40531-3. Co-author.

[P5] Mohagheghi, P., Conradi, R.: Using Empirical Studies to Assess Software Development Approaches and Measurement Programs. *Proc. the ESEIW 2003 Workshop on Empirical Software Engineering (WSESE'03) - The Future of Empirical Studies in Software Engineering.* Rome, September 29, 2003, Andreas Jedlitschka and Marcus Ciolkowski (Eds.), pp. 65-76. Main author.

[P6] Mohagheghi, P., Conradi, R.: Different Aspects of Product Family Adoption. Proc. the 5th International Workshop on Product Family Engineering (PFE-5), Siena, Italy, November 4-6, 2003, F. van der Linden (Ed.), *Springer LNCS 3014,* pp. 429-434, 2004. Main author.

[P7] Mohagheghi, P., Conradi, R.: An Industrial Case Study of Product Family Development Using a Component Framework. *Proc. the Sixteenth International Conference on Software & Systems Engineering and their Applications (ICSSEA'2003)*, December 2-4, 2003, Paris, Volume 2, Session 9: Reuse & Components, ISSN: 1637-5033, 6 p. Main author.

[P8] Mohagheghi, P., Conradi, R., Killi, O.M., Schwarz, H.: An Empirical Study of Software Reuse vs. Defect-Density and Stability. Proc. the 26th International Conference on Software Engineering (ICSE'04), May 23-28, 2004, Edinburgh, Scotland, pp. 282-292. *IEEE Computer Society* Order Number P2163. The paper received one of the five Distinguished Paper Awards at the conference. Main author.

[P9] Li, J., Conradi, R., Mohagheghi, P., Sæhle, O.A., Wang, Ø., Naalsund, E., Walseth, O.A.: A Study of Developer Attitude to Component Reuse in Three IT Companies. Proc. the 5th International Conference on Product Focused Software Process Improvement (PROFES 2004), April 5-8, 2004, Kansai Science City, Japan, *Springer LNCS 3009*, pp. 538-552. Co-author.

[P10] Mohagheghi, P., Conradi, R: An Empirical Study of Software Change: Origin, Acceptance Rate, and Functionality vs. Quality Attributes. Accepted at the ACM-IEEE International Symposium on Empirical Software Engineering (ISESE 2004), August 19-20, 2004, Redondo Beach CA, USA, 10 p. Main author.

[P11] Mohagheghi, P., Conradi, R.: Exploring Industrial Data Repositories: Where Software Development Approaches Meet. *Proc. the 8th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering* (QAOOSE'04), Olso, Norway, June 15, 2004, Coral Calero, Fernando Brito e Abreu, Geert Poels and Houari A. Sahraoui (Eds.), pp. 61-77. Main author.

[P12] Mohagheghi, P., Conradi, R.: A Study of Effort Breakdown Profile in Incremental Large-Scale Software Development. To be submitted to IEEE Computer. Main author.

[P13] Mohagheghi, P., Anda, B., Conradi, R.: Use Case Points for Effort Estimation - Adaptation for Incremental large-Scale Development and Reuse Using Historical Data. To be submitted. Main author.

The published papers can be downloaded from the software engineering group's publication list at the Department of Computer and Information Science, NTNU [IDI-SU04].

1.6 Contributions

The contributions are integrated in two observations:

- Several aspects of software development must be revised when introducing a development approach such as reuse or incremental development. This work investigated the software process model, inspection techniques, estimation method, effort distribution and metrics.
- The above aspects should also be analyzed and adapted for a combination of development approaches and the context. Some other research has also

identified this fact. Further evidence is provided and some improvements are proposed.

In addition to a descriptive study of incremental development of a product family in [P1], [P3] and [P7], this thesis contains five main novel contributions and two minor contributions which will now be presented.

1.6.1 Contributions in Software Reuse

C1. Empirical verification of reuse benefits
This is the first empirical study of a large industrial system. The main contributions are:

C1-1. A quantitative analysis of TRs showed that reusable components have significantly lower defect-density than non-reused ones. Reused components have, however, more severe defects but fewer defects after delivery, which shows that defects of these components are given higher priority to correct. The study did not show any relation between component size and defect-density for all components. Only for non-reused components, an increase in the number of defects with the component size was observed.

C1-2. A quantitative analysis of the amount of modified code between releases showed that reused components are more stable (are less modified) between successive releases.

C1-3. A quantitative analysis of CRs did not show any difference in change-proneness (the number of CRs/Component size) between reused and non-reused components.

Some metrics are defined (and the related data is collected) at the level of blocks, while others are defined at the level of subsystems. The statistical analysis is done at both levels when possible and sometimes with only subsystems.

C6a. Adaptation of the Rational Unified Process (RUP) regarding reuse
The approach to initiating a product family was an extractive one. Software architecture has evolved to support reuse, while the software process model (an adaptation of RUP) has not adapted for reuse to the same degree. Adapting the process model beforehand was not considered critical for initiating reuse or reuse success. However, it is likely that the company will gain in the long term from adapting RUP explicitly regarding reuse with proposed changes in workflows and activities, for both development *for* and *with* reuse. No empirical studies have been found on evaluating or adapting RUP in this aspect.

1.6.2 Contributions in Incremental Development

C2. Increased understanding of the origin and type of changes in requirements or artifacts in incremental development
A quantitative analysis of CRs showed that perfective changes (enhancements and optimizations) are most common. Of these, non-functional changes to improve "quality attributes" are more frequent than are pure functional changes. The results also show that earlier releases of the system are no longer evolved. Functionality is enhanced and improved in each release, while quality attributes are mostly improved and have fewer changes in forms of new requirements. The share of adaptive/preventive changes is lower, but still not as low as reported in some previous studies. There is no previous

literature on the share of non-functional changes. Other literature claims that the origin of changes is usually outside the organization, while this study showed that most changes are initiated by the project organization itself. The results contribute to the understanding of how software evolves in incremental development of large systems and have generated hypotheses for future empirical studies in the same domain or other domains.

C3. Developing an effort estimation method using use case specifications and effort distribution in different phases of incremental software development

The Use Case Points (UCP) effort estimation method is adapted for complex use cases (described in UCSs) with many main and alternative flows, incremental changes in these and reuse of software from previous releases. The complex UCSs have been broken down into smaller ones, points have been calculated for all the steps in UCSs and the modified ones, and the method is simplified by assuming an average project when it comes to assigning values to several factors. The study also contributed in adding an Adaptation Factor (AF, set to 0.55) to count for reuse of software from a previous release (contra developing from scratch), which is borrowed from COCOMO 2.0. The estimation method is calibrated by using historical data on effort from one release and is verified by using data from the successive release. The UCP method was earlier only tested on small projects and without incremental development.

A quantitative analysis of distribution of effort over activities using historical data from two releases showed that approximately half the effort is spent on activities before system test. The other half is spent on project management, system test, software process development, Configuration Management (CM) and some minor effort-consuming activities (documentation, inspections, travel etc.). No similar study on effort consumption in new development approaches and for large systems has been found. The results have been used in adapting the estimation method. They are also useful in breaking down total effort between activities in a top-down estimation method.

C6b. Improving techniques for inspection of UML models

Data from 38 earlier inspections were used to develop a baseline for the cost-efficiency of the company's existing inspection technique. A controlled experiment with two teams of developers was performed, comparing the company's existing inspection technique with a tailored version of the Object-Oriented Reading Techniques (OORTs) developed by the University of Maryland. The results showed no significant difference in the cost-efficiency of the two techniques, but there was a difference in the types of detected defects. The OORTs have previously been the subject of several student experiments and one industrial case study, but no controlled experiment with incremental development of UML models and with models that sometimes cover entire walls. The method fitted (unexpectedly) well into the development process, but needs further improvements and adjustment to the context.

1.6.3 Contributions in Software Reuse and Incremental Development

C4. Metrics for a combination of reuse of software components and incremental development
Results of qualitative and quantitative analysis are used to assess the company's measurement program and the relations between quality metrics and development practices (and the underlying development approaches). Other literature discusses metrics for CBD. However, these metrics should be adapted for incremental development and for reuse. Metrics have been identified for a combination of reuse of software components and incremental development, intended for assessing development approaches and building more complex models.

1.6.4 Research Method

C5. A data mining method for exploring industrial data repositories
A data mining method is developed that is based on experience from the quantitative studies. It shows steps in such studies, combines top-down theory search with bottom-up hypotheses generation and uses the traditional data mining steps in the execution phase. Challenges in performing data mining studies and integrating the results of several studies with one another are classified into physical and conceptual ones.

1.6.5 Summary of Contributions

Table 1-1 shows benefits in terms of improved process quality or improved single component quality. It also shows the papers that describe the contributions.

Table 1-1 The relations of contributions to the quality of process and single components

Contribution	Process quality	Single component quality	Paper
C1-1, C1-2, C1-3. Reuse benefits		Reused components are more stable and have less defect-density. No relations between defect-density or the number of defects, and size of components are observed.	P8 P10
C2. Software evolution	Generating theory on evolution in incremental development.		P10
C3. Effort estimation method	Improved effort estimations. Generating theory on effort distribution in large-scale incremental development.		P12 P13
C4. Metrics	Assessing company's measurement program. Useful in assessing development approaches.	Useful in assessing quality of single components.	P5 P11
C5. Data mining method	Developing research method.		P11
C6a. Software process model	More consistency between practice and process model is advised. Developers use process web pages and supplement them with expert knowledge and previous work.	Better documentation of reused components is needed.	P2 P6 P9
C6b. Inspection techniques	Consistency between UML models, and between UML models and requirements can be improved.	Certification of components.	P4

1.7 Thesis Structure

This thesis consists of two parts:

- Part I aims to provide an introduction to the field, put the research into the context, integrate the results, and evaluate the overall research design and the results. It covers Chapters 1 to 8.
- Part II contains 13 papers that provide detailed results and discussions of the individual studies. Part II covers Chapter 9.

Figure 1-2 shows the structure of Part I of this thesis.

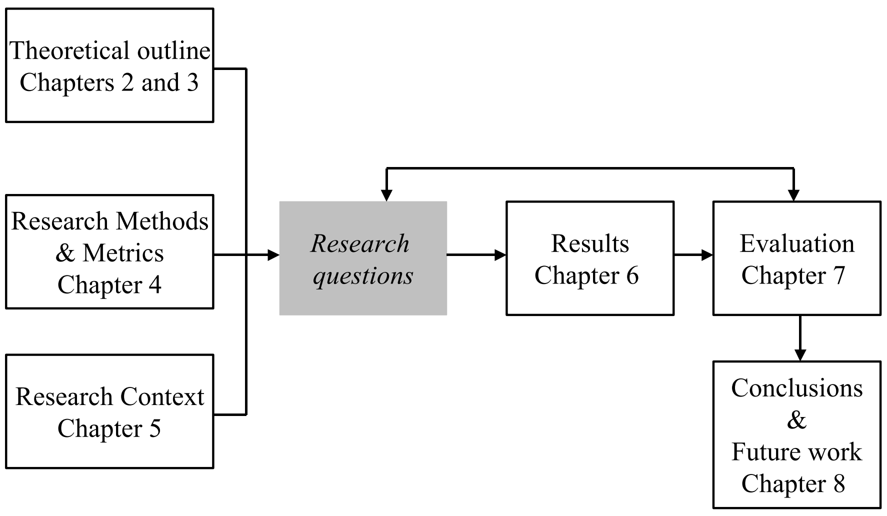

Theoretical outline Chapters 2 and 3
Research Methods & Metrics Chapter 4
Research Context Chapter 5
Research questions
Results Chapter 6
Evaluation Chapter 7
Conclusions & Future work Chapter 8

Figure 1-2 The structure of Part I of this thesis

Chapters 2 and 3 are an introduction to the field and introduce challenges that are faced in this thesis. Chapter 4 presents an overview of research methods and metrics, and challenges in selecting research methods. The research context, describing the company context, challenges in large-scale software development, relations to the INCO goals and the research design in this study are subjects of Chapter 5. Research questions, which are derived from previous work reported in the literature and the research context, are already presented in Section 1.3. All the papers P1-P13 and their contributions are presented in Chapter 6, in addition to some results and discussions that are yet not published. The research questions are further answered in Chapter 7. The relations between the research questions, papers, contributions and INCO goals are presented, and the experience from working in the field is also discussed in Chapter 7. The thesis is summarized in Chapter 8 and future work is proposed. All the papers are given in Chapter 9.

In the both parts of this thesis, I have generally used the term "we" to present the work, either my reflections in Part I or the studies done jointly as reported in Part II. Research is a collaborative process and I have received valuable feedback on all the parts, especially from my supervisor.

A second note is on the format of the papers in Part II that had to be changed to fit the format of Part I. I have also provided more information on the context and the results of some studies in Part I to justify the conclusions and facilitate interpretation.

2 *Software Reuse and Component-Based Development*

This chapter describes the challenges in software engineering that are the motivation behind reuse, incremental and Component-Based Development (CBD) approaches. Then, there is a classification of literature related to software reuse, CBD and product family engineering. The definitions of these subjects are discussed and research challenges are described for each of them. Two side effects of CBD are briefly presented that are reason behind proposing alternative approaches. Finally, the whole chapter is summarized and the research challenges are described related to the studies in this thesis.

2.1 Software Engineering Definitions and Challenges

Software engineering describes the collection of technologies that apply an engineering approach to the construction and support of software products. *Technology* is used here in a broad sense, meaning concepts, principles, development approaches, methods, tools, techniques and even software processes. Endres et al. define a *project* as an organizational effort over a limited period of time, staffed by people and equipped by with the other resources required to produce a certain result [Endres03]. For a development project, the result to be achieved is a *product* (same place). In this thesis, a (software development) project refers to the effort required to develop one *release* of a software product in incremental development as well.

A *system* according to Endres et al. consists of hardware and software that is used by other people [Endres03]. This thesis uses the term “system” both when the software part is in consideration (meaning a software product) and for the working system as a whole, for instance the GPRS system. A *software process* is a set of activities and methods that gives a software product. A software process tells which activities to do, in what order and what to produce (artifacts) to have a software product. A *software process model* is a representation of a software process.

Software engineering activities or phases include managing, estimation, planning, modeling, analyzing, specifying, designing, implementing, testing and maintaining [Fenton97]. Software organizations have always been looking for effective strategies to develop software faster, cheaper and better. The term *software crisis* was first used in 1968 to describe the ever-increasing burden and frustration that software development and maintenance placed on otherwise happy and productive organizations [Griss93].

Many different remedies have been proposed, such as object-oriented analysis, Computer-Aided Software Engineering (CASE) tools, formal methods, Component-Based Software Engineering (CBSE), automatic testing, and recently Model Driven Architecture (MDA) and Aspect-Oriented Programming (AOP). After decades of software development, the software industry has realized that there is no “silver bullet”; despite arguments of promoters of new technologies that there is. There are several factors that limit the success of technologies among these immature processes, immature methods and tools, unsatisfactory training, organizational resistance to change, immaturity of technologies and inappropriate use of them, ands the inherent difficulty of developing software, especially for large and complex software products.

Philippe Kruchten discusses why software engineering differs from structural, mechanical and electrical engineering due to the soft, but unkind nature of software. He suggests four key differentiating characteristics [Kruchten01]:

- Absence of fundamental theories or at least practically applicable theories makes is difficult to reason about the software without building it.
- Ease of change encourages changes the software, but it is hard to predict the impact.
- Rapid evolution of technologies does not allow proper assessment, and makes it difficult to maintain and evolve legacy systems.
- Very low manufacturing costs combined with ease of change have led the software industry into a fairly complex mess. Kruchten refers to the continuous bug-fixings, new updates and redesign.

It can also be added that:

- Almost every software project is unique and collecting context-independent knowledge is difficult.
- Markets are in a constant state of flux, encouraging changes in requirements and systems.
- Software development is inherently complex, especially for large systems.

How have software engineers tried to solve the crisis in their discipline? Krutchen’s answer is by *iterative development* and *CBD*. Iterative development seeks to find an *emergent solution* to a problem that is discovered gradually. CBD seeks to reduce complexity by offering high-level abstractions, separation of concerns, and encapsulating complexity in components or hiding it. This thesis considers how these solutions are combined and work in large system development.

The characteristics mentioned above have even become more extreme due to Internet-speed development. Internet-speed development involves rapid requirement changes and unpredictable product complexity [Baskerville03]. In this process, quality becomes negotiable, while rapid development becomes more important. The strategy is to acquire, integrate and assemble components. Companies developing products for these markets have less time to develop for reuse, but maximize development with reuse.

2.2 Literature Overview

A classification of literature on software reuse and CBD is given in order to place this thesis and related work in this landscape. The following groups are identified and examples of literature are provided for each group:

1. *Software reuse*. In his book on the results of the ESPRIT Project REBOOT, Karlsson gives a good overview of all aspects of software reuse (such as organizational aspects, metrics for measuring reuse, development for and with reuse, managing a reuse repository, the Cleanroom adaptation, object-oriented design for reuse and documenting reuse) [Karlsson95]. Jacobson et al.'s book describes the *reuse-driven software engineering business* to manage business, architecture, process and organization for large-scale software reuse [Jacobson97]. They focus on software architecture, and the three distinct activities of *component system engineering*, *application system engineering*, and *application family engineering*. Notations are UML-based, with use cases to specify both the super-ordinate system and subordinate component systems. Morisio et al. and Rine et al. summarize many reuse cases and discuss reuse success factors [Morisio02] [Rine98]. One of the recent books on software reuse is [Mili02], describing technological, organizational, and management or control aspects.
2. *CBD and CBSE*. A classical book on this subject is written by Szyperski [Szyperski97]. The second edition also discusses new approaches such as MDA (Model Driven Architecture), .NET, EJB and others [Szyperski02]. SEI published two reports on the state of CBSE in 2000, one on market assessment [Bass00] and the other on technical aspects [Bachmann00]. Heineman and Council are editors of a handbook on all aspects of CBSE [Heineman01]. Crnkovic and Larsson are editors of a similar book but with more focus on reliable CBD [Crnkovic02]. Atkinson et al.'s book on the KobrA approach supports a model-driven, UML-based representation of components and a product family approach to software development using components [Atkinson02]. Some of the best-known CBSE processes are Catalysis ([D'Souza98] and [Wills in Chapter 17, Heineman01]), Select [Allen98], UML components [Cheesman00], the Rational Unified Process (RUP) [Kruchten00], and OPEN (Object-oriented, Process, Environment and Notation) being a more generic framework for development ([Graham97] [Henderson-Sellers in Chapter 18, Heineman01]). Atkinson et al. provide a brief overview of these processes in [Atkinson02].
3. *Product families/product lines/system family*. Jan Bosch discusses software architecture, quality attributes and software architecture for product families (or product lines) in his book [Bosch00]. His article on maturity levels of software product families gives a framework to discuss different cases [Bosch02]. Jazayeri et al.'s book also discusses software architecture for product families [Jazayeri00]. Another book, which is often cited in relation with product families, is Clements and Northrop's book [Clements01]. Both authors have many articles on the subject as well. Jacobson et al. discuss application family engineering [Jacobson97], and Atkinson et al.'s KobrA process supports product family engineering with components [Atkinson02]. One research actor

on product family development is the Software Engineering Institute (SEI) at Carnegie Mellon University, which has published several technical reports on the subject [SEI04a]. A good comparison of several domain analysis methods is given in [Mili02].

4. *COTS-related*. Commercial-Off-The-Shelf (COTS) software is software that is not developed inside the project, but acquired from a vendor and used "as-is", or with minor modifications. There is extensive literature on definitions of COTS and COTS-based systems, selection of COTS products and processes (e.g. [Torchiano04], [Vigder98a], [Ncube and Maiden in Chapter 25, Heineman01], [Brownsword00], [Morisio03], [Wallnau98], [Carney00], [Basili01]), but less on integration and certification of COTS software (e.g. [Voas98]).
5. *Component technologies* such as CORBA, .NET and EJB. These technologies are best described by their providers, but are compared in various parts of the literature. Longshow compares COM+, EJB and CCM [Heineman01, Chapter 35]. Estublier and Favre also compare Microsoft component technologies (COM, DCOM, MTS, COM+ and .NET) with CCM, JavaBeans and EJB [Crnkovic02, Chaper 4]. Szyperski classifies these technologies in 3 groups [Szyperski02]: the OMG way (CORBA, CCM, OMA and MDA), the SUN way (Java, JavaBeans, EJB and Java 2 editions) and the Microsoft way (COM, OLE/ActiveX, COM+, .NET CLR), and gives an overview of each group and compares them.

The rest of this chapter gives a brief overview on points 1 to 3 in the above list are given and challenges relevant for this thesis are discussed. When considering software reuse and CBD, there are general issues that are also relevant for COTS-based development. However, the specific challenges for COTS-based development are not discussed, since they are not relevant for this study.

2.3 Software Reuse

While literature on CBD is almost all written in recent years, discussion on reuse started in 1969. Doug McIlroy first introduced the idea of systematic reuse as the planned development and widespread use of software components in 1968 [McIlroy69]. Many software organizations around the world have reported successful reuse programs such as IBM, Hewlett-Packard, Hitachi and many others [Griss93]. The reports show that reuse actually works and refer to improved productivity, decreased time-to-market and/or decreased cost.

Reuse is an umbrella concept, encompassing a variety of approaches and situations [Morisio02]. The reusable components or assets can take several forms: subroutines in library, free-standing COTS (Commercial-Off-The-Shelf) or OSS (Open Source Software) components, modules in a domain-specific framework (e.g. Smalltalk MVC classes), or entire software architectures and their components forming a product line or a product family.

Mili et al. define *reusability* as a combination of two characteristics [Mili02]:

1. *Usefulness*, which is the extent to which an asset is often needed
2. *Usability*, which is the extent to which an asset is packaged for reuse.

They add that there is a trade-off between usefulness (generality) and immediate usability (with no adaptation).

Morisio et al. define reuse as [Morisio02]:

> *Software reuse* is the systematic practice of developing software from a stock of building blocks, so that similarities in requirements and/or architecture between applications can be exploited to achieve substantial benefits in productivity, quality and business performance.

The benefits should be quantified and empirically assessed. The above definition excludes ad-hoc reuse, reuse of knowledge, or internal reuse within a project. Frakes et al. define software reuse as "The use of existing software knowledge or artifacts to build new software artifacts", a definition that includes reuse of software knowledge. Morisio's definition is closer to what is meant by "software reuse" in this thesis; i.e. reuse of building blocks in more than one system. Reuse of software knowledge such as domain knowledge, or patterns may happen without reuse of building blocks and is captured in domain engineering.

Developing for reuse has its price, which is the reason for analyzing the success of reuse programs to improve the chances of succeeding. Morisio et al. have performed structured interviews of project managers of 32 Process Improvement Experiments funded by the European Commission, in addition to collecting various data about the projects [Morisio02]. These projects vary a lot in size, approach, type etc. and few of them have defined reuse metrics. The study found that:

- Top management commitment is the prerequisite for success.
- Product family practice, common architecture and domain engineering increase reuse capability.
- Size, development approach (object-oriented or not), rewards, repository and reuse measurement are not decisive factors, while training is.
- The other three factors that are considered to be success factors are reuse process introduced, non-reuse process modified and human factors.
- Successful cases tried to minimize change, to retain their existing development approach, choosing reuse technology to fit that.

Morisio et al. concluded that reuse approaches vary and it is important that they fit the context. However, this work emphasizes the reuse process. Griss writes that reuse needs [Griss95]:

- Management support, since reuse involves more than one project.
- Common wisdom. There is no evidence that object technologies or libraries give improvement in reuse. Some people also say that process understanding is nothing or all. **Introduce reuse to drive process improvements**. Domain stability and experience are often more important for successful reuse than general process maturity.
- Incremental adoption.

Frakes et al. have investigated 16 questions about software reuse using a survey in 29 organizations in 1991-1992 [Frakes95]. They report that most software engineers prefer to reuse rather than to build from scratch. They also did not find any evidence that use of certain programming languages, CASE tools or software repositories promote reuse. On the other hand, reuse education and a software process that promotes reuse have positive impact on reuse. They also found that the telecom industry has higher levels of reuse than some other fields.

Some challenges in research on software reuse are:

- Verifying Return On Investment (ROI) either in reduced time-to-market, increased productivity or in improved quality.
- Identifying the preconditions to start a reuse program.
- Developing processes for software reuse, roles, steps and adapting existing processes.

2.4 Component-Based Development

Component-Based Development (CBD) and Component-Based Software Engineering (CBSE) are often used indistinguishably, but some literature distinguishes between these two. Bass et al. write that CBD involves technical aspects for designing and implementing software components, assembling systems from pre-built components and deploying system into target environment. CBSE involves practices needed to perform CBD in a repeatable way to build systems that have predictable properties [Bass00]. It has been decided to use CBD in the remainder of this thesis to cover all the aspects of engineering systems from pre-built components.

CBD is an approach to the old problem of handling the complexity of a system by decomposing it. Already in 1972, David Parnas wrote about the benefits of decomposing a system into modules, such as shorter development time since modules can be developed by separate groups, increased product flexibility and ease of change, and increased comprehensibility since modules can be studied one at a time [Parnas72]. He also wrote that the main criteria for modular decomposition should be *information hiding*. Modules such as in Ada and procedural languages and objects in object-oriented design are example of previous attempts at decomposition. Software reuse has also been discussed for decades. So what is new in CBD? The answer is the focus on software architecture as a guideline to put pieces together, viewing components as independent units of development and deployment, and on component models. Developing components so that they become reusable is called developing *for reuse*, while developing systems of reusable components is called developing *with reuse* [Karlsson95]. CBD facilitates reuse by providing logical units for assembly and makes systematic reuse possible by demanding that components should adhere to a component model.

There are two distinct activities in CBD [Ghosh02]:

1. Development of components *for* component-based development.
2. The component-based development process itself, which includes assembly.

Bachman et al. list advantages of CBD as [Bachman00]:

- Reduced time to market: even if component families are not available in the application domain, the uniform component abstractions will reduce overall development and maintenance costs.
- Independent extensions: components are units of extension and component models prescribe how extensions are made.
- Component markets to acquire components.
- Improved predictability: component frameworks can be designed to support the quality attributes that are most important.

And advantages added by Bass et al. are [Bass00]:

- Improved productivity, which can lead to shorter time-to-market.
- Separation of skills: complexity is packaged into the component framework and new roles are added such as developer, assembler and deployer.
- Components provide a base for reuse, since components are a convenient way to package value. They have direct usability (may be used to build systems directly), while other approaches such as design patterns are more abstract.

Others mention that extended maintainability and evolvability and fast access to new technology are reasons for choosing CBD for developing systems when the main concern is change (see for instance [Cheesman00]). The growing use of OSS (Open Source Software) is also a new trend to build systems rather fast and cheaply.

Use of components is a clear trend in industry even though the technologies are far from mature. Bass et al. mention that today's technology consumers have accepted improved productivity and shorter time-to-market in exchange for a vague trust to components and component frameworks [Bass00]. This picture may have changed.

Components are a convenient way to package value: they provide a flexible boundary for economy of scope and they can be easily distributed. With economy of scope it is meant that a component can be fine-grained or coarse-grained and the scope can be changed. In contrast 4GL and object-oriented frameworks are more rigid. Components are designed to be a unit of distribution [Bass00].

Components are defined and classified in multiple ways. Definitions vary based on the life cycle phase for component identification (e.g. logical abstractions vs. implementation units), origin (in-house, bought or free software), or roles a component can play in a system (e.g. process components, data components etc.). A few of these definitions are given here, before discussing what is important for reuse.

In the SEI's report on technical aspects of CBSE, a component is defined as [Bachmann00]:

- An opaque implementation of functionality.
- Subject to third-party composition.
- Conformant to component model. This is mentioned as a difference with other COTS software with no constraints on conformance to an architecture.

Heineman and Council define a software component as "A software element that conforms to a component model, can be independently deployed and can be composed without modification according to a composition standard" [Heineman01]. And finally:

> A software component is an executable unit of independent production, acquisition, and deployment that can be composed into a functioning system. To enable composition, a software component adheres to a particular component model, and targets a particular component platform [Szyperski02].

What these three definitions have in common are:

- Components are units of independent development and acquisition.
- Components adhere to a component model that enables composition of components. *Composition* is the term used for components, instead of integration.

None of these two aspects are to be found in object-oriented design. Some other differences with object-oriented design are:

- Instantiation: components may be instantiated or not, and if instantiated there are usually not many instances of them [Atkinson02] [UML2.0].
- Components may have state (e.g. KobrA) or not (in order to be replaceable they should not have state).
- Granularity: components are generally considerably larger than individual classes [Bosch00].

Currently, CBD is mainly carried out using UML for modeling, object-oriented languages for design, and component technologies such as EJB, .NET and CORBA for implementation. All these component technologies are especially developed for distributed systems, which shows that the complexity of these systems and the need for autonomous units promote the use of components.

The terms *component model* and *component framework* are often intermixed. However, it becomes more common to use component model for standards and conventions, and component framework for an implementation of a component model that also gives the infrastructure support [Heineman01] [Bachman00] [Crnkovic02]. The concept of frameworks was initially used for object-oriented frameworks, consisting of a set of related classes with extension points. What is new with component frameworks is that they provide run-time services for components and are part of the final system [Bachman00]. Two aspects are important in component frameworks:

1. Component frameworks define how components interact and thus are part of the software architecture.
2. Component frameworks affect quality attributes either by defining rules, or by providing services. A component framework handles several quality requirements either by [Bosch00]: specific component in the framework, or design patterns for application developers, or a combination of both approaches above.

Developing component frameworks is demanding. Some commercial component frameworks (also called component technologies) are EJB and .NET, while an example of a domain-specific component framework is described in Sections 5.1.3 and 5.1.5. Domain-specific frameworks provide reusable design for a domain in addition to run-time services and are developed for a set of products. They may be implemented on top of other component frameworks.

CBD is about building composable components and building systems from these components. Important aspects are therefore **reuse**, **autonomy** of components and **composition**. Challenges or inhibitors are due to immaturity or lack of software engineering methods, processes and tools in all these aspects. Bass et al. mention inhibitors in CBD as lack of available components, lack of standards for component technology, lack of certified components and lack of engineering methods [Bass00]. Crnkovic lists the challenges of CBSE as: specification, component models, life cycle, composition, certification and tools [Crnkovic02]. The present focus is on software engineering methods. Some challenges in each development phase and for a project as a whole are discussed here, using [Crnkovic02], [Ghosh02], [Jacobson97] and other various sources, and putting them together as:

1. **Management**: decision-making on build vs. reuse vs. buy, initiating product families, ROI, vendor interactions for COTS and cost estimates. Although component markets have grown in the recent years, there are few empirical studies that can verify increased productivity or shorter time to market due to acquiring components.
2. **Requirement engineering**:
   - Selection of components and evaluating these for functional, quality or business requirements, and possible trade-offs between requirements and selected components. Selection is mostly important for COTS components, but also when a company has a reuse repository to choose components from.
   - Traceability between requirements and components.
3. **Analysis and design**:
   - Software architectures such as components and connectors, pipes and filters [Zave98], agent-based [Zave98], blackboard style [Crnkovic02] [Bosch00] and layering [Jacobson97] [Bosch00]. These architecture styles emphasize components being developed independently of one another. Layering is on a higher level of abstraction, applicable to different architecture styles. Architectures for component-based systems should allow building systems by composing components, allow plug-and-play style and allow reuse of components across products. Bosch and Bruin et al. define a similar approach to architecture design [Bosch00] [Bruin02]: derivation of an architecture that meets functional requirements and optimizing it for quality requirements step-wise.
   - Decomposing a system into components, modeling, understanding components, various design decisions on handling concurrency, binding[1] and control (processes or threads).

[1] *Binding* means that resources provided by one component become accessible to another component or bound to the client. Component models talk of early or late binding [Bachman00]: In early binding, the developer must make some decisions and is

  - Implementation: selecting component model or framework, developing glue code or wrapper, component configuration or adaptation and composition.
4. **Prediction and Verification**:
   - Predicting and verifying functional and quality requirements. Components and frameworks should have certified properties, and these certified properties provide the basis for predicting the properties of systems built from components [Bachman00].
   - Validating component assemblies (testing, modular reasoning) and checking the correctness of compositions.
   - Testing: black-box testing without access to source code becomes frequent, vendor's response to trouble reports and isolating faults. Hissam et al. give an overview of techniques for component and system observation [Hissam98].
   - Quality assurance techniques such as inspections. Mark Vigder in [Vigder98b] provides a list to check, for instance connectors, architecture style, interfaces, tailoring and component substitution for evolution.
   - Metrics for component-based systems.
5. **Configuration Management (CM)**: CM is the discipline of managing the evolution of software systems. CM becomes more important because of possible different versions at each site, history of updates, handling licenses and compatibility issues.
6. **Relations between CBD and other approaches** such as incremental development [Atkinson02].
7. **Software processes** that meet the above challenges for component-based systems.

Services of a component are defined by its interfaces and are therefore easier to verify. On the other hand, specification, implementation and assessment of quality attributes are more demanding. Crnkovic et al. mention that CBSE faces two types of problems when dealing with extra-functional properties (extra-functional properties, quality requirements and non-functional requirements refer all to the same properties) [Crnkovic02]:

a) The first problem is common to all software development and concerns imprecise definition of these properties.
b) The second problem is specific to CBSE and concerns the difficulty of relating overall system properties to individual component properties.

Voas further mentions the difficulty of composing quality attributes. Functional composability, even if it were a solved problem (using formal methods, modern design approaches, model checking etc.) is still not mature enough to compose *itilities* [Voas01]. He mentions that the question is which *itilities*, if any, are easy to compose.

also called development time binding like in EJB. Late binding is run-time binding, e.g. JavaBeans. Late binding requires early binding of design decisions on how components will coordinate their activities. This is consistent with the overall philosophy of component-based software engineering: **architecture first** and leads to prediction prior to assembly.

He answers that none of *itilities* are easy to compose and some are much harder to compose than others. A component model defines how components interact and hence embraces aspects that have impact on many quality attributes such as scalability or security. These quality attributes may thus be easier to predict, while others are still left to application systems built on component models.

2.5 Product Families

Many organizations are using a product family engineering approach for software development by exploiting commonalities between software systems, and by reusing software architecture and a set of core assets. Product family engineering is reuse at the largest level of granularity [Atkinson02]. The terms *product family engineering, product line engineering, system family engineering* and *application family engineering* are used for a wide range of approaches to develop a set of products with reuse. The main idea is to increase the granularity of the reused parts and define a common architecture for a family of systems. The use of terminology is sometimes confusing. Frank van der Linden explains it as, " Certain European companies use product line to indicate a set of related, commercial products that have appear similar to users but often are built with different technologies. For example, product lines in consumer electronics include televisions, VCRs, DVD players, audio receivers, CD players, audio amplifiers and so on. We use product family to describe a collection of products that are based on the same technology- for instance a collection of TVs based on the same software architecture. Often products in the same product line are in different product families and vice versa" [Linden02]. The European community uses *product family* for software products that are built using the same technology, which is the same as a *product line* in USA. In this thesis, definitions are provided as they originally are in the references, but the term *product family* is used for the Ericsson case study, in discussions, or when indirectly referring to a paper.

Parnas wrote the first paper on development of systems with common properties in 1976. He wrote: "We consider a set of programs to constitute a *family*, whenever it is worthwhile to study programs from the set by *first* studying the common properties of the set and *then* determining the special properties of the individual family members" [Parnas76].

SEI has conducted research on product families for a few years and has published several technical reports and notes, results of a survey and of several case studies in companies having a product family (see [SEI04a]). SEI defines a software product family/product line as:

> A software product line is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission, and that are developed from a common set of core assets in a prescribed way.
>
> SEI's Product Line Practices initiative [SEI04b]

The recent literature on CBD discusses developing components for product families, e.g.:

- With increasing frequency, components are not sold alone but rather as a family of related and interacting components [Bass00].
- When combining software architecture with component-based development, the result is the notion of software product lines [Bosch00].

Ommering and Bosch summarize the driving forces of proactive, systematic, planned and organized approaches towards software reuse for sets of products, i.e. product lines, as shown in Figure 2-1 [Crnkovic02]. The dashed line is added here: Size and complexity of a problem promotes CBD.

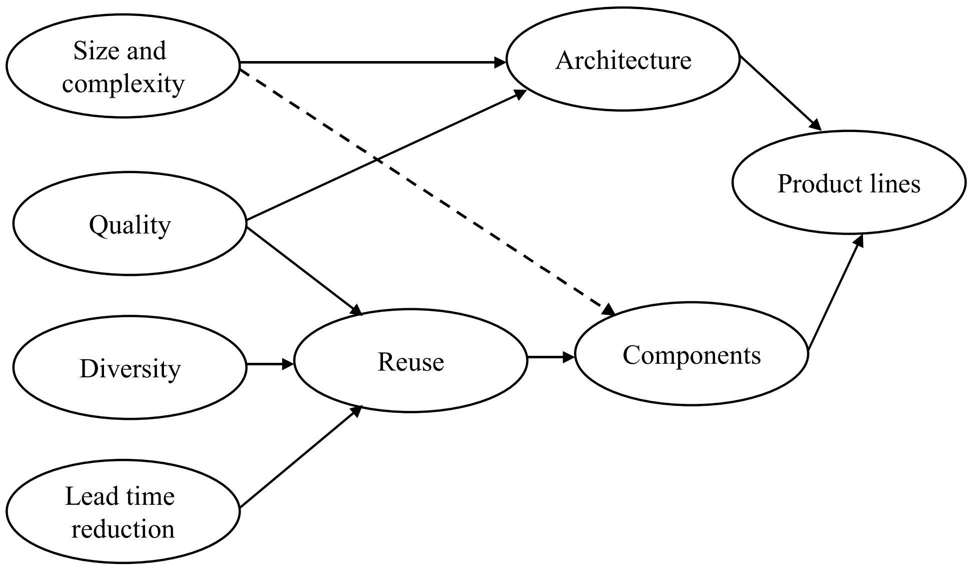


Figure 2-1 Basic arguments for software product lines [Ommering and Bosch in Crnkovic02, Chapter 11]

Product families face the same challenges as other reuse-based approaches as discussed in Section 2.3 such as:

1. How to initiate a product family?
2. How ROI can be assessed? What is the economical impact?
3. What are the organizational or quality impacts?

For product families, we also ask:

4. How is variability or diversity managed? What is the impact on software architecture?
5. How the scope is defined?

Reviewing literature that handles these questions is outside the scope of this thesis. This section discusses questions 1 and 5 that are relevant for the discussion of the Ericsson case study. Quality impacts are later discussed in combination with reuse.

Each product in a product family is developed by taking applicable components from a common asset base, tailoring them through preplanned variation mechanisms, adding new components as necessary and assembling the collection according to the rules of a common, product-family-wide architecture [Northrop02].

A basic concept in this discussion is the concept of a domain. Mili et al. define a *domain* as "An area of knowledge or activity characterized by a family of related systems [Mili02]. A domain is characterized by a set of concepts and terminology understood by practitioners in that specific area of knowledge". Further, they characterize a domain by one of the three criteria: common expertise (producer-focused), common design (solution related), or common market (business related). It is also usual to differ between *problem domains* and *solution domains* [Mili02] [Bosch00]. The core activity in domain engineering is *domain analysis*, which handles the process of eliciting, classifying and modeling domain-related information. Sometimes domain analysis is not performed as a distinct activity, for example when an organization has solid knowledge of the domain [Northrop02].

SEI defines three essential product line activities [Northrop02]:

1. D*omain engineering* for developing the architecture and the reusable assets (or development *for* reuse as called in [Karlson95]).
2. *Application engineering* to build the individual products (or development *with* reuse as called in [Karlson95]).
3. *Management* at the technical and organizational level.

McGregor et al. divide approaches for introducing a product family into *heavyweight* and *lightweigh*t [McGregor02]. In the heavyweight approach, commonalities are identified *first* by domain engineering and product variations are foreseen. In the lightweight approach, a first product is developed and the organization then uses mining efforts to extract commonalities. The choice of approach also affects cost and the organization structure. Charles Krueger claims that the lightweight approach can reduce the barrier to large-scale reuse, as it is a low-risk strategy with lower upfront cost [Krueger02]. Johnson and Foote write that useful abstractions are usually designed from the bottom up; i.e. they are discovered and not invented [Johnson98].

Krueger defines another classification of strategies or adoption models [Krueger02]. The three prominent adoption models are:

- *Proactive*. When organizations can predict their product line requirements and have time or resources, they can design all product variations up front. This is like waterfall development approach to conventional software.
- *Reactive*. This is more like spiral or extreme programming approaches to software development. One or several product variations are developed on each spiral. This approach is best suited when product line requirements are not predictable, or there are not enough resources or time during transition.
- *Extractive*. This approach reuses one or several products for the product line initial baseline. This approach is effective for an organization that wants a quick transition from single product development to a product line approach.

While being proactive can pay off [Clements02a], lightweight approaches have lower risk when products cannot be foreseen. The discussion above on adoption models (or initiation approaches) shows that organizations start and maintain product families in multiple ways. Bosch identifies two factors in deciding which approach is best suited for an organization when adopting a product line [Bosch02]:

a) Maturity of the organization in terms of domain understanding, project organization, management and degree of geographical distribution (less distribution promotes the product line approach in his view).
b) Maturity and stability of the domain. For stable domains it is easier to maximize domain engineering.

Scoping is the selection of features that are to be included in the product family architecture. Bosch answers this question by identifying two approaches [Bosch00]: the minimalist approach that only incorporates those features in the product family that are used by all products. The maximalist approach incorporates all and products should exclude what they do not need. Commonalities and variations in product family requirements and implementations are often defined by abstracting these in *features*. Feature-Oriented Domain Analysis (FODA), first introduced by SEI in 1990 [Kang90], appeals therefore to many organizations. FODA assumes forward engineering and a dedicated product family organization. The KobrA process has also guidelines for forward engineering in product family, and defines activities Framework Engineering and Application Engineering in product family development [Atkinson02].

2.6 Alternatives to Component-Based Development

There are two special side effects in CBD that are tried to be answered by alternative approaches:

- Components are structural and not behavioral units. Therefore, there is only a vague connection between requirements and the structure of the system (this problem is not limited to CBD). The difficulty of traceability between requirements and components is also important for composition and verification. Decomposition into components has two well-known effects called *tangling* and *scattering* [Tarr99]. Tangling means that a given component contains code coming from implementation of several use cases (or any requirement in general). Scattering means that a set of components is required to implement a use case (crosscutting property).
- Traceability between requirements and components and assessment is even more challenging for quality attributes or non-functional requirements. These requirements are related to the whole system and not to a single component, and cannot be specified by interfaces.

The alternative approaches propose either to remove the structural units, or to be more precise with non-functional requirements and add these to a component specification. Two alternative approaches are discussed here: Aspect-Oriented Programming (AOP) that can be combined with CBD or be performed without components, and generative techniques.

AOP is seen by some people as a way to overcome the above problems [Jacobson03] [Pawlak04]. With AOP, partial implementations will be developed, each addressing one single quality aspect of the component. These will be woven together by especially designed aspect weavers to a complete component, complying with a certain contract. AOP can be combined with CBD to support composition of components. Example is Aspect-Oriented Component Engineering (AOCE), in which a component specifies provided and required aspects in addition to business functions [Grundy00]. AOCE avoids "code weaving" that makes it difficult to reuse components. Instead each component inherits from special classes that provide functions to access and modify aspects. Ivar Jacobson aims for use case modularity, and defines *a use case module* as a module containing a use case realization and a set of slices of components participating in realizing the use case [Jacobson03]. He also sees the possibility to remove components totally and have two steps: specify each use case and code it. Pawlak et al. also propose behavioral decomposition and composition of aspects [Pawlak04].

Atkinson et al. consider that the weaver-based approaches and the AOP community have so far been unable to fully resolve the superimposition problems [Atkinson03]: these approaches completely separate aspect code from the base code. This strength is also a weakness: when several aspects and the base code interfere at some join points, issues of priority, nesting and mutual execution arise. However AOP can lead to develop domain-specific environments and domain-specific languages that can ease software development and automatic generation of code.

Another approach to CBD is reflected in generation techniques; i.e. the specification of a component in some component-specific language is taken to a generator that translates the specification into code. For example, Bruin et al. propose generating components from functional and non-functional requirements, instead of composing these, close to aspect weaving in AOP [Bruin02].

One weakness of both AOP and generation techniques is the reuse difficulty. Domain-specific solutions may reduce the complexity of these techniques, but also limit the potential market. Both these techniques are still in early infancy stage, while commercial component models have been in the market for a while and have achieved some success.

2.7 Summary and the Challenges of this Thesis

This chapter has shown that software reuse is the systematic practice of developing software from a stock of building blocks. When combining with CBD, these building blocks are components developed according to a component model. Product family development is reuse and CBD in the large; i.e. developing a set of products that reuse some core assets, combined with a software architecture that can handle commonalities and variabilities between these products. One common architectural solution is a layered architecture to group pieces that have similar change characteristics; e.g. in FODA and [Jacobson97].

Some challenges in software reuse, CBD and product family development were discussed in the previous sections. This section describes which of these challenges are the subjects of this thesis in the context of incremental development of a large system. These Research Challenges (RCs) can be defined as:

RC1. Adaptation for reuse. Software processes, software architecture and organizations should be adapted for reuse, CBD and product family development. This adaptation can (and should preferably) happen gradually. The issue is how this adaptation happens or should happen.

RC2. Combination of approaches. Software development approaches are seldom used in isolation, but in combination with one another and existing practices in companies. Incremental development is combined in this case with reuse, CBD and product family development. Therefore, the attempt is made to provide a holistic view of the system and the development approaches, combining the approaches and seeking for their mutual impact.

RC3. Reuse benefits. The issue is whether we can quantify reuse benefits (if any benefits are achieved) as far as data is available. If these benefits are not observed, the case will be a falsifying case.

RC4. Exploring. There are few case studies on large-scale systems and most empirical work is performed in form of surveys. Exploring case studies may provide new insight or new research questions on the impact of complexity or scale on software development methods. For some software engineering methods, the question is whether these methods scale up. An example is the UCP estimation method, which is the subject of [P13], and was earlier tested only on systems with few use cases. On the other hand, software engineering methods for product families are designed for large systems and several products, and the question is whether these methods scale down when the number of products is low or they are of small size.

3 *Incremental Development*

This chapter begins by defining incremental development and the motivations behind choosing it. Then, it describes variations in incremental approaches. It also reports results of a few empirical studies on the impact of incremental development, prototyping or incremental testing on product and project metrics. Since RUP is the software process in Ericsson, a brief introduction to this is given. Finally, challenges facing this study regarding incremental development are described.

3.1 Definitions

Incremental development is known as an alternative to the waterfall software development method with its strict sequence of requirements, analysis, design and development phases. However, incremental approaches vary in aspects such as the recommended iteration length, the amount of up-front specification work, or emphasis on feedback and adaptation-driven development. There is also a confusion of terminology in this area and iterative development, incremental development, time boxing, spiral development and versioned development are used inconsistently in the literature.

Larman and Basili provide a brief history of iterative and incremental development in [Larman03]. According to them, the history of incremental development goes back to the 1930s when Walter Shewhart, a quality expert at Bell Labs, proposed a series of Plan-Do-Study-Act cycles for quality improvement. In 1975, Basili and Turner defined Iterative and Incremental Development (IID) as:

> The basic idea behind *iterative enhancement* is to develop a software system incrementally, allowing the developer to take advantage of what was being learned during the development of earlier, incremental, deliverable versions of the system…At each iteration, design modifications are made along with adding new functional capabilities [Basili75].

This definition emphasizes learning and feedback as the main motivation behind iterative development and does not distinguish between incremental and iterative development. And from RUP:

> *Iteration*: A distinct sequence of activities with a base-lined plan and valuation criteria resulting in a release (internal or external).
> *Increment*: The difference (delta) between two releases at the end of subsequent releases.
>
> [Bergström03]

Studying different sources have led to the conclusion that *incremental development* is often used for delivering a subset of requirements (in a working system) in each increment, while *iterative development* is used for recursive application of development activities and recursive elaboration of artifacts. *Time boxing* is increments of fixed length. What distinguishes incremental development from *prototyping* is that increments are not thrown away, but are supposed to deliver a complete system, while prototypes are usually thrown away. Incremental development is also used for development methods with major up-front specification and pre-planned product improvements, while in an *evolutionary approach* product improvements are not preplanned and requirements are gradually discovered. An important fact about increments is that they **accumulate functionality**; e.g. release 2.0 builds on release 1.0.

Other aspects of incremental or iterative development are:

- User participation and user feedback [Mills76],
- The need to do risk assessment in each iteration (Gilb and others had previously applied variations of this idea) [Boehm85],
- Gilb emphasizes non-functional requirements in each increment and having measurable goals (for example performance goals) [Gilb88],
- Short iterations as in eXtreme Programming (XP) [Kent99].

The Cleanroom approach to software development also has incremental development as one of its core practices. The others are recursive development (recursive application of common abstraction techniques), non-zero defect software, rigorous specification and design, and usage testing (testing expected system usage in terms of system states, their transitions and dependencies) [Atkinson02]. In 1994, the Standish Group issued its widely cited “CHAOS: Charting the Seas of Information Technology”, which is followed by several reports after that. The Standish Group has analyzed several thousand projects to determine success and failure factors [Standish04]. The top reason for success is user involvement, while firm requirements is also a success factor that is in contrast with incremental development.

3.2 Variations in Incremental Approaches

What increments mean in practice? Even-André Karlsson gives examples such as [Karlsson02]:

a) Each product family release is an increment. These increments are delivered to the customer.

b) Within a project there can be several increments, each adding to the functionality of the previous one.
c) Each sub-project or team can divide the work in increments that can be tested in a simulated environment.

Karlsson asks several questions that should be answered when applying incremental development. Three of these questions that are relevant for this study are presented and answered by getting help from Karlsson and others.

QI. What is the functionality of an increment?

Increments can be:

- *Feature increments*: distinct user functions or features are added in each increment.
- *Normal/error increments*: simple normal cases are developed first. More complexity is added to normal cases in the successive iteration, for example adding error handling.
- *Separate system function increments*: for example, in the telecom domain start and restart is developed first. Commands, traffic handling and other user functionality are added later.
- *Component-oriented increments*: KobrA assigns components and stubs of their children to increments (in order to deliver an executable version) and gradually goes in-depth in component realizations [Atkinson02].

The major difference is between feature-oriented and component-oriented approaches. Both can be combined with normal/error increments. The system function increments is considered as a variant of feature increments.

Atkinson et al. write that software architectures do not lend themselves to incremental development [Atkinson02]. One reason is that architectures should be bearer of non-functional requirements and thus a total view of the system should be developed. As a remedy, they propose component-oriented increments. The disadvantage of this is that the approach is even more dependent on excessive up-front requirement and design work. Furthermore, not all non-functional requirements are possible to assign to single components and no functionality is completely built in early increments.

When developing an entire system from scratch, a sufficiently small first increment is usually difficult to find. For example, in product family development some reusable assets should be developed first, since all components rely on the services of a component framework. In this case a combination of features and normal/error increments may be useful. Feature increments have the advantage of testing all parts of the system early, but the disadvantage of reopening some design items several times. It increases the need for inspections and regression testing to ensure consistency and compliance with earlier deliveries.

QII. How long are the increments and how are they scheduled?

The question has three aspects: the duration, whether the duration is fixed or variable, and whether they are done sequentially with all personnel or in parallel:

- Short increments keep up the focus, but can result in a focus on the code and neglecting other documentation. Long increments become like the waterfall model.
- Fixed duration or time boxes make planning easier, but splitting the functionality into increments of equal size may be difficult. Variable increments require more planning.
- Sequential increments need no coordination between increments, while parallel increments allow better use of scarce resources e.g. test environments.

QIII. How is work allocated?

There are basically two different strategies:

- *Design item responsibility*. People are assigned to design items and deliver the functionality necessary for each increment. This is more natural for normal/error increments or component-oriented increments.
- *Increment responsibility*. People are assigned to increments and do the necessary functionality in each item affected by an increment. This is more natural for feature increments.

Design item responsibility is better for complex design items and has the advantage of better knowledge about the item, no cost to open or understand the design, and better consistency in design. Increment responsibility gives better system understanding and no handover of intermediate results (less communication). It is possible to add the role "design item coordinator" in connection with increment responsibility to get the advantages of both approaches, as Ericsson does.

3.3 Incremental Development and Quality

Incremental development is chosen to reduce the risks of changing requirements and environments, and to learn from experience or user feedback. Other risks associated with the "big bang" approach are also reduced such as risks associated with new technologies. It also allows companies to enter the market with an early version of a system. However, the impact on cost, effort, organization and software quality should be further assessed.

Jalote et al. suggest that development time may be reduced by time boxing and parallel iterations [Jalote04]. Each time box is divided into stages of almost equal duration. They argue that the total development time is not reduced in a sequential approach and in fact it can take more time than a waterfall model if all requirements were known. But fixed time boxes reduce the turnaround time and allow better utilization of resources. The constraints are that the method needs a good feature estimation method, tight CM as teams work in parallel and is best fitted to medium-sized projects that have a lot of features to deliver. In other words, other aspects of software development such as estimation method or CM are important for a software development method to work.

Two papers that relate development practices to measurable attributes of product or project are mentioned in [P5]. The first paper reports results of a survey in 17 organizations on correlation between defect-density and probability of on-time delivery, and practices [Nuefelder00]. Incremental development is not among the practices mentioned. Incremental testing as opposed to big bang testing had strong negative correlation with defect-density, meaning that quality is improved by incremental testing. Having a life cycle model also decreases defect-density.

The second paper reports results of a survey among managers in Hewlett-Packard [MacCormack03]. A total of 29 projects were analyzed, and quantitative data such as size of projects and defect-density are also collected. The authors correlate 8 practices with defect-density and productivity. The results show that different practices are associated with multiple dimensions of performance. For example, the use of regression tests has an impact on the defect rate, but not on productivity. Conversely, the use of daily builds increases productivity, but does not affect the defect rate. The practice that has an impact on multiple dimensions is early prototyping. They found a weak relation between dividing the project in sub-cycles (delivering functionality in pieces) and the defect rate, but little effect on productivity. They argue that early prototyping allows getting feedback from customers and developing what they want. They conclude that practices should be considered as coherent systems of practices, be chosen depending on the attribute that should be optimized and may trade-off for other practices.

3.4 The Rational Unified Process (RUP)

This section provides a brief introduction to RUP, since Ericsson uses an adaptation of it. This relies on [Arlow02] for the history of UP and RUP, [Kruchten00] for an introduction to RUP and [Bergström03] for adopting RUP.

The history of the Unified Process (UP) goes back to 1967 when Ericsson (with Jacobson working there) took the step of modeling a complex system as a set of interconnected blocks and also defined traffic cases; i.e. how the system is used. Later SDL (Specification Description Language) was defined and became a standard for specifying telecom systems. Together with Booch and Rambaugh, Jacobson developed Unified Modeling Language (UML), which has gradually replaced SDL. RUP is introduced in 2001 and the version used by Ericsson at the time of this study was RUP 5.5.

RUP is a software engineering process and also a process product. RUP is the most widely commercial variant of UP. It has added a lot of features to UP that both extends and overwrites UP. RUP can and should be adapted (tailored) to suit the needs of an organization and a concrete project. RUP is based on the six best practices as shown in Figure 3-1.

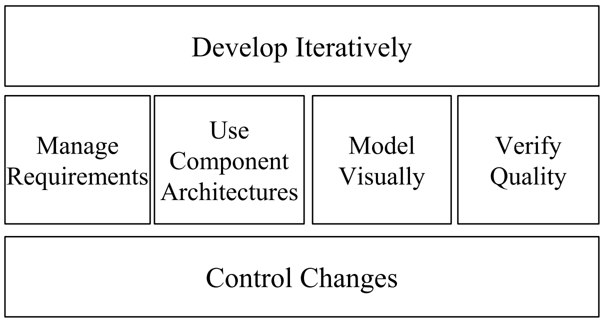


Figure 3-1 Best practices of RUP

RUP is iterative, use-case-driven (creation of other models as well as test artifacts will take off from the use case model) and architecture centric. One core practice in RUP is developing a software architecture in early iterations.

Figure 3-2 shows the four phases and nine workflows or disciplines in RUP. The phases and the goals for each phase are:

1. *Inception* - Define the scope of a project and identify all the actors and use cases. Draft the most essential use cases (almost 20%) and allocate resources.
2. *Elaboration* - Plan project, specify features and develop a stable software architecture. The focus is on two aspects: a good grasp of the requirements (almost 90%) and establishing the architecture baseline.
3. *Construction* - Build the product in several iterations up to a beta release.
4. *Transition* - The product is delivered into end-user community. The focus is on installation, training, support and maintenance.

Each phase may be executed in one or more iterations. There is a milestone at the end of each phase. The milestone at the end of the elaboration phase is the *architecture milestone*. Bergström et al. emphasize that this milestone is the most important one and it can only be passed when the vision, architecture and requirements are stable, the testing approach is proven and an executable prototype is built [Bergström03]. RUP emphasizes:

- Up-front requirement specification to assign requirements to increments,
- Early stable software architecture,
- Variable length increments where final stages of an iteration can overlap with initial stages of the next one,
- Assignment of use cases to increments (a variation of feature increments).

The concepts of role, activity and artifact are central in RUP. A *role* performs an *activity* to produce or update an *artifact*.

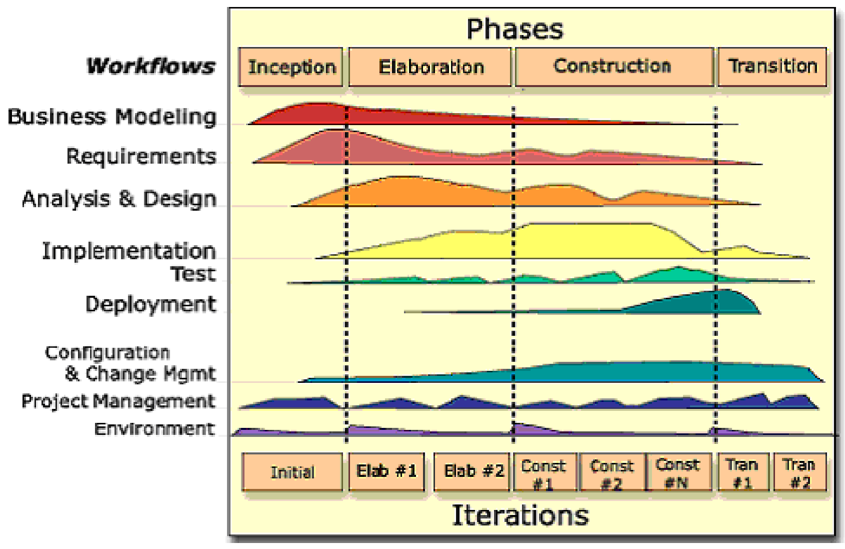


Figure 3-2 Phases, workflows (disciplines) and iterations in RUP

Adapting RUP can be done by selecting parts, for example, workflows of it (Bergström et al. discuss *adopting RUP*, which normally means selecting, but not changing a method. This work talks of *adapting RUP*; i.e. selection and changing). Many companies start with use cases when choosing RUP. However, each workflow is very large and one should exactly decide what to choose. Probably the easiest approach to adaptation is selecting artifacts, related activities and roles [Bergström03]. Some changes are easier than others, e.g. adding templates or guidelines, while adding or removing roles or artifacts may introduce inconsistencies. RUP also comes with a tool called "RUP Builder" which allows the selection of three variants of RUP depending on the size of the project: Small, Medium and Large (Classic). Bergström et al. emphasize that the one practice that should not be excluded is the **architecture-first approach** (the architecture milestone) [Bergström03]. Many consider RUP as a heavyweight process (i.e. many rules, practices and documents), compared to lightweight processes with few rules and practices. There are two points in this discussion:

a) The difference between RUP and processes such as XP is not only the amount of produced artifacts, but also the stability of requirements, up-front requirement specification and the architecture-centric approach of RUP.
b) RUP can also be used lightweight.

RUP is applicable for development of several types of systems such as component-based or real-time systems. Components in RUP are defined in the implementation view and are executable units.

RUP defines a *component* as a non-trivial, nearly independent, and replaceable part of a system that fulfils a clear function in the context of a well-defined architecture. A component conforms to and provides the physical realization of a set of interfaces.

Although RUP is widely used, there is a lack of empirical studies on RUP. A study of introducing RUP in a few Norwegian companies shows that although the motivation was improving practices such as requirement specification, these improvements are not later assessed [Holmen01]. One important advantage was, however, achieving a uniform process in different units of an organization.

RUP is very rich in notation, contains best practices in software development, is claimed to be appropriate for a wide range of systems and is continuously evolved. However, being generic means that it lacks guidelines for specific domains or types of projects. For example, RUP does not have guidelines for developing *for* and *with* reuse. Rational has started a forum to develop RAS (the Reusable Asset Specification), which is a set of concepts, notations and guidelines for describing reusable assets of business systems, thus improving later search. In a project report at NTNU in 2002, the students gathered a list of tools that supported RAS [Schwarz02]. RAS may be useful when a company plans to start a searchable database for reusable assets; i.e. a reuse repository. As discussed in Section 2.3, having a reuse repository is not proven to be a success factor for reuse. On the other hand, introducing a reuse process or adapting a non-reuse process is important for the success of reuse.

3.5 Summary and the Challenges of this Thesis

This chapter has discussed approaches to incremental development and the questions that should be answered when selecting an approach. It has also briefly discussed how RUP answers these questions. The research challenges related to incremental development in this study are presented here. RC1 and RC2 are specific cases of RC1 and RC2 defined in Section 2.7. The numeration of research challenges is continued from Section 2.7:

RC1. RUP Adaptation. Since RUP is the process of Ericsson, we ask whether and how it may be adapted regarding reuse.

RC2. Combination. Incremental development is combined with a product family approach. As in Section 2.7, the case is studied for understanding how these approaches are combined and what the impact of this combination is.

RC5. Quality impact. The impact of incremental development on effort, product quality or organization is not studied sufficiently. There is a lack of empirical studies, especially case studies.

RC6. Software evolution and maintenance. There are empirical studies on software maintenance, but maintenance and evolution in the context of incremental development is not studied empirically in the literature. Evolution is an inherent characteristic of incremental development. Software companies need to understand how software evolves and have processes to manage it, such as CM and requirement change handling. This work aims to empirically study the first aspect; i.e. how software evolves in incremental development.

4 Research Methods and Metrics

This chapter provides a brief introduction to research approaches and strategies. It also discusses advantages and challenges in the case study approach as a research approach used in several studies in this thesis. Validity threats for all types of studies and in particular how to overcome these for case studies are further discussed. Goals and criteria for defining metrics and types of metrics are described. Finally, the challenges are discussed, facing empirical studies in general and in this thesis in particular in selecting research methods.

Comprehensive introductions to this field can be found in [Wohlin00] [Creswell03] [Cooper01] [Juristo01]. Kitchenham et al. also provide a first attempt to define explicit guidelines for performing and reporting empirical software engineering research [Kitchenham02].

4.1 Research Strategies in Empirical Research

Empirical research is research based on the scientific paradigm of observation, reflection and experimentation as a vehicle for the advancement of knowledge [Endres03]. Empirical studies may have different purposes, being *exploratory* (investigating parameters or doing a pre-study to decide whether all parameters of a study are foreseen), *descriptive* (finding distributions of a certain characteristics) or *explanatory* (why certain methods are chosen and how they are applied).

There are three types of research paradigms that have different approaches to empirical studies and may be used for all the above-mentioned purposes [Wohlin00] [Creswell94] [Creswell03] [Seaman99]:

- *Quantitative approach* is mainly concerned with quantifying a relationship or comparing two or more groups. The aim is to identify a cause-effect relationship, verify hypotheses or test theories.
- *Qualitative approach* is concerned with studying objects in their natural environment. A qualitative researcher attempts to interpret a phenomenon based on explanations that people bring to them. Developing software is a human intensive activity and in the recent years the community has increasingly used qualitative methods from the social sciences in empirical software engineering research. The primary intent is to develop theory or make interpretations of data. Qualitative data is usually subjective, unstructured and non-numeric.

– The *mixed-method approach* is evolved to compensate for limitations and biases in each of the above strategies, seeking convergence across other methods or triangulation[2] of data and combining advantages of both strategies. Both quantitative and qualitative data are collected sequentially or in parallel, based on the assumption that collecting diverse type of data provides a better understanding of a research problem.

An overview of research approaches and examples of strategies used in each is shown in Table 4-1, which relies on [Creswell03].

Table 4-1 Alternative research approaches

Approaches	Quantitative	Qualitative	Mixed methods
Strategies	- Experimental design - Non-experiment designs such as surveys - Case studies	- Ethnographies - Grounded theory - Case studies - Surveys	- Sequential - Concurrent - Transformative
Methods	- Predetermined - Instrument based questions - Numeric data - Statistical analysis	- Emerging methods - Open-ended questions - Interview data - Observation data - Document data - Text and image analysis	- Both predetermined and emerging methods - Multiple forms of data drawing on all possibilities - Statistical and text analysis
Knowledge claims	Postpositivism: - Theory test or verification - Empirical observation and measurement	Constructivism: - Theory generation - Understanding - Interpretations of data	Pragmatism: - Consequences of action - Problem-centered - Pluralistic

Note that the boundaries between approaches are not sharp. For example, surveys can be open-ended or explanatory, being considered as a qualitative study, and case studies can combine quantitative and qualitative studies [Wohlin00]. Yin also warns against considering case study equal to qualitative research [Yin03]. Flyvbjerg writes that good research should be problem-driven, not methodology-driven [Flyvbjerg04]. More often, a mixed-method approach will provide the best answer.

A brief definition of some of the strategies that are used in the studies in this thesis is given here:

[2] Triangulation may be of data sources (*data triangulation*), among different evaluators (*investigator triangulation*), of perspectives of the same data (*theory triangulation*), and of methods (*methodological triangulation*) [Yin04]. The present discussion covers the first type; i.e. collecting data from multiple sources to address the same fact or phenomenon.

- Quantitative strategies:
    - *Experiments* include true experiments with random assignment of subjects to treatments, as well as quasi-experiments with non-randomized design and single-subject experiments. Experiments with students as subjects are more common in universities, while industrial experiments or experiments with professionals as subjects are very few. The Simula Research Laboratory in Oslo has developed a web-based environment for conducting experiments and surveys, which they have been used in several empirical studies [Simula04]. An example of using professionals in laboratory experiments is a controlled experiment to compare the effect of a delegated versus centralized control style on the maintainability of object-oriented software. A total of 99 junior, intermediate and senior professional consultants from several international consultancy companies and 59 students participated in the controlled experiment [Arisholm04].
    - *Surveys* include cross-sectional and longitudinal studies using questionnaires or structured interviews, with the intent of generalizing from a sample to the population. An example is a recent survey on the state of practice in CBSE was performed by the CBSEnet project among 109 industrial organizations [CBSEnet04] [Escalante03]. One problem of large-scale surveys is the low response rate. SEI reports a response rate of 20% in their survey on the state of practice in product family development [Cohen02]. Some other surveys report even lower response rates.
    - *Case studies as a quantitative strategy* are conducted to investigate quantitatively a single phenomenon within a specific time frame.
- Qualitative strategies:
    - In *grounded theory*, the researcher attempts to derive a general, abstract theory of a process grounded in empirical data. Two characteristics of this design are the constant comparison of data with emerging categories and theoretical sampling of different groups to maximize the similarities and the differences of information.
    - *Case studies as a qualitative strategy* explore in depth a program, an activity or process over a period of time. An example is a case study on product family development in Salion, Inc. [Clements02b]. Clements et al. refer to the case as unique in the sense that the company did not have substantial experience in its application area. Salion pursued a reactive approach to its product family.
- Mixed-method strategies:
    - *Sequential procedures*, in which the researcher seeks to elaborate on or expand the findings of one method with another method.
    - *Concurrent procedures*, in which the researcher converges quantitative and qualitative data to provide a comprehensive analysis of the research problem. Data is collected concurrently and results are integrated in the interpretation phase.
    - *Transformative procedures*, in which the researcher uses a theoretical lens within a design that contains both quantitative and qualitative data. Creswell mentions a feministic or racial lens as examples.

The important question in research design is when to use each strategy. If the problem is identifying factors that influence an outcome or test the effect of some manipulation, quantitative approaches are chosen. If the problem is to understand why the results are as they are or to identify causes, a qualitative approach is best. The mixed method approach uses different methods in the various phases of a study.

Yin answers the question of choosing an approach by listing three conditions [Yin03]:

a) The type of research question posed. *How* and *why* questions are explanatory, and usually should be studied over time in replicated experiments or case studies. *What, who*, *where*, *how many* or *how much* questions ask about the frequency or describe the incidence of a phenomenon. *What* questions can also be exploratory in which case any of the strategies may be used.
b) The extent of control an investigator has over actual behavioral events. Only in experiments, can the researcher control treatments or behavioral events. In a case study, the researcher cannot control treatment, but may control the measures to be collected.
c) The degree of focus on contemporary as opposed to historical events.

Other factors distinguishing approaches from one another are:

d) The ease of replication: lowest in case study and highest in experiments according to [Wohlin00].
e) The risk of intervening: highest for case studies and lowest for surveys.
f) Scale: experiments are "research-in-the-small", case studies are "research-in-the-typical" and surveys that try to capture a larger group are "research-in-the-large" [Kitchenham95].
g) Cost: formal experiments are costly, have limited scope and are usually performed in academic environments. Industry does not have time or money to spend on experiments.

While each research strategy has limitations, most research strategies can be applied for exploratory, descriptive or explanatory reasons. For example:

- In grounded theory, cases are selected for their value to refine existing or exploring new classifications.
- A history or archival analysis may also be applied to answer which method or tool is better in a given context.
- Surveys measure people's opinion about a phenomenon, which in cases may not reflect the real distribution, or may be affected by contemporary events.
- Case studies can be applied as a comparative research strategy, comparing the results with a company baseline or a sister project [Wohlin00].

Two strategies are in general applicable for overcoming limitations of research strategies:

a) Replication of studies over time and in multiple contexts.
b) Combination of strategies. For example surveys can be combined with open-ended interviews, and case studies can include analysis of archival records, quasi-experiments and interviews.

Ramesh et al. analyzed a sample of 628 papers published in 13 major computer science journals between 1995 and 1999 for topics, research methods, the level of analysis (the object that is studied) and the theoretical foundation of the research [Ramesh04]. They classify research approach as being

- Formulative (formulating processes, methods, guidelines etc.), covering 79.15% of papers,
- Descriptive (describing development or a product), covering 9.88% of papers,
- Evaluative, covering 10.98% of papers.

When it comes to research methods:

- Over 88% use conceptual analysis (either mathematical or not).
- Case study and field study represent 0.16% each.
- Experiments represent 3.6% (either human subjects or software).
- Research methods such as grounded theory, ethnography and descriptive/exploratory surveys are not represented at all, as well as field experiments.

Not surprisingly, Computer Science (CS) itself is the reference discipline in 89.33% of papers, followed by mathematics (8.60%). The results show that CS research is relatively focused when it comes to research approaches and seldom relies on work outside the discipline for its theoretical foundation.

4.2 The Case Study Approach

Case studies are very suitable for industrial evaluation of software engineering methods and tools because they can avoid scale-up problems observed in small experiments [Kitchenham95]. The difference between case studies and experiments is that experiments sample the variables that are being manipulated, while case studies sample the variables representing the typical situation. Formal experiments also need appropriate levels of replication, and random assignment of subjects and objects.
Yin identifies the situation when the case study has an advantage as [Yin03]:

A ”how” or ”why” question is being asked about a contemporary set of events, over which the investigator has little or no control.

Yin further define a case study as [Yin03]:

A case study is an empirical inquiry that

- Investigates a contemporary phenomenon within its real-life context, especially when,
- The boundaries between phenomenon and context are not clearly evident.

During the performance of a case study, a variety of different data collection procedures may be applied [Creswell94]. In fact, a case study relies on multiple sources of evidence, with data needing to converge [Yin03].

Flyvbjerg summarizes the wide extent critical remarks against case studies to five points [Flyvbjerg04]:

1. General theoretical (context independent) knowledge is more valuable than concrete, practical, context-dependent knowledge.
2. One cannot generalize on the basis of an individual case (one data point). Therefore, the case study cannot contribute to scientific development.
3. The case study is most useful for generating hypotheses; i.e. the first step of research, while other methods are more suitable for hypotheses testing and theory building.
4. The case study contains a bias towards verification; i.e. to support the researcher's pre-assumption.
5. It is often difficult to summarize and develop general propositions and theories on the basis of specific case studies.

Flyvbjerg then argues against these same points:

1. In the areas of his interest (environment, policy and planning), context-independent knowledge is not available. Context-independent theories are for novices during learning, while professionals have intuitive approach based on case knowledge and experience. This argument is also valid for software engineering.
2. Generalization can often be done on the background of cases, but normally the possibility of formal generalization is overestimated - even though case studies are brilliant to falsification tests. Formal generalization is overvalued as a source of scientific development, whereas the force of example is underestimated. Yin comments that the analogy to samples and universes is incorrect in case studies [Yin03]. Survey research relies on *statistical generalization*, whereas case studies rely on *analytical generalization*. In analytical generalization, the researcher strives to generalize a particular set of results to some broader theory or to a broader application of a theory.
3. This misunderstanding derives from the previous one. Generalizability of case studies can be increased by strategic selection of cases. For example *atypical* or *extreme cases* (e.g. especially problematic or especially good projects) often reveal more information than typical ones. Another example is that *most likely cases* are suited for falsification of propositions, while *least likely cases* are appropriate to test verification. Yin adds *critical cases* (see below), *revelatory cases* (when an investigator has an opportunity to observe and analyze a phenomenon inaccessible before) and *longitudinal cases* (study a case over time) to the spectrum of valuable cases.
4. Typically case studies report that their pre-assumptions and concepts were wrong and hypotheses must be revised. The case study contains no greater bias than any other method of inquiry.
5. It is true that summarizing case studies is difficult, but the problem is more often due to the properties of the studied reality than to case study as a research approach. Many good studies should be read as narratives in their entirety.

While choosing multiple cases increases the reliability and generalizability, a single case study is interesting when the rationale is one of those mentioned in point 3 above:

- A critical case is important in testing a well-formulated theory: if it is(not) valid for this case, it is(not) valid for many cases (or any case). A case study can challenge the theory, test it, or extend it. Critical cases allow logical deduction [Flyvbjerg04].
- A representative case can be a typical project that can be informative about the average projects.

Most theories in software are developed based on studies in a defined context. Formal experimentation is over-emphasized, often not possible, the results cannot scale-up and are therefore not convincing for the industry. In software engineering, industrial case studies are rare due to several reasons:

- Companies do not allow outsiders to access critical information or publish the results either due to the confidentiality of results or the risk of intervening with the on-going project.
- Performing a case study may need observation and collection of data over months or even years.
- Wohlin et al. write that case studies are easier to plan, but the results are difficult to generalize and are harder to interpret [Wohlin00]. However, there are issues that make planning difficult: it takes time to gain the necessary permissions, overcome the communication barrier and understand the context. The results are harder to interpret and generalize due to the impact of the context.
- Finally, a case study may take another turn than planned; projects may be stopped, or changes in personnel or environment may happen that affect data collection.

On the other hand, good case studies are as rare as they are powerful and informative [Kitchenham95].

4.3 Validity Threats

A fundamental discussion concerning results of a study is how valid they are. Empirical research usually uses definitions of validity threats that originate from statistics and not all the threats are relevant for all types of studies. Wohlin et al. define four categories of validity threats [Wohlin00]:

- *Conclusion validity* (for statistical analysis)- "*right analysis*": this validity is concerned with the relationship between the treatment (the independent variable in a study) and outcome (the dependent variable). We want to make sure that there is a statistical relationship of significance. Threats are related to choice of statistical tests, sample sizes, reliability of measures etc.
- *Internal validity* (for explanatory and causal studies, not for exploratory or descriptive studies)- "*right data*": we must make sure that there is a causal relationship between treatment and outcome and that is not a result of factors that are not measured. Threats are related to history, maturation, selection of subjects, unpredicted events and interactions, ambiguity about the direction of causal influence etc. Yin adds (*Experimental) Reliability* to this: demonstrating

that a study's operations can be repeated with the same results such as data collecting [Yin03].

- *Construct validity- "right metrics"*: we must ensure that the treatment reflects the cause and the outcome reflects the effect. Threats are mono-operation bias (a single case may not reflect the constructs), mono-method bias (a single type of measure may be misleading), hypotheses guessing etc.
- *External validity- "right context"*: this validity is concerned with generalization of results outside the scope of a study. Three types of interactions with the treatment may happen: people (the subjects are not representative for the population), place (the setting is not representative) and time (the experiment is conducted in a special time for example right after a big software crash). Yin terms this establishing the domain to which a study's findings can be generalized [Yin03].

Different threats have different priorities based on type of research. For example, in theory testing, internal validity is most important, while generalization is not usually an issue. For a case study, Yin identifies tactics to improve validity as:

- Use multiple of sources in data collection and have key informants to review the report in composition to improve construct validity.
- Perform pattern matching (comparing en empirically based pattern with a predicted one especially for explanatory studies) and address rival explanations in data analysis to improve internal validity.
- Use theory in research design in single case studies to improve external validity.

4.4 Measurement and Metrics

Measurement is central in any empirical study, especially for benchmarking (collecting and analyzing data for comparison) and to evaluate the effectiveness of specific software engineering methods, tools and technologies [Fenton00a]. Benchmarking can also be used to calibrate tools such as estimation tools [Heires01].

Measurement is mapping from the empirical world to the formal, relational world. Consequently, a *measure* is the number or symbol assigned to an entity by this mapping, in order to characterize an attribute [Fenton97] [Wohlin00]. The term *metrics* is used either to denote the field of measurement or to the measured attribute of an entity and related data collection procedures. In this thesis, *measurement* is used for the activity of measuring and *metrics* for an attribute that is measured such as software size.

The first dedicated book on software metrics is published in 1976 [Gilb76], while the history of software metrics dates back to the mid-1960s when the Lines of Code (LOC) metric was used as the basis for measuring programming productivity and effort [Fenton00a]. Recent work emphasizes:

- Building causal models that are more complex. To do so, Fenton et al. suggest using Bayesian Belief Nets that can handle uncertainty, causality and combining different (often subjective) evidence [Fenton00a]. Jørgensen et al. discuss the fact that theory building is generally neglected in empirical studies [Jørgensen04].

- Combining results of different studies [Kitchenham01] and different methods. For example, Briand et al. combine scenario-based and measurement-based product assessment [Briand01].

Some attributes are directly measurable (e.g. size of a program in LOC), while others are derived from other measurements and are called indirect measures (e.g. productivity in LOC/effort). Measures can also be divided into objective and subjective measures: an *objective measure* is a measure where there is no judgment in the measurement value, such as LOC. A *subjective measure* depends on both the object and the viewpoint, such as personal skill [Wohlin00].

In software engineering, entities we wish to measure are usually divided into three classes [Fenton97]:

- Processes: such as effort or duration Sommerville calls these metrics *control metrics* [Sommerville00].
- Products: artifacts that result from process activities. Sommerville calls these metrics *predictor metrics,* such as LOC or number of defects.
- Resources: entities needed by process activities, such as developers or tools.

Sedigh-Ali et al. [Sedigh-Ali01b] mention the importance of *quality metrics* in the early stages of software development. In contrast to quality attributes that are *user-oriented* (such as reliability or Quality of Service), quality metrics are *developer-oriented*, because developers can use them to estimate quality at a very early stage of development (such as defect-density). Later in the development lifecycle, the purpose of measurement is to assess whether a quality attribute is achieved and to predict the future values or trends.

Measures are classified into five major types: nominal, ordinal, interval, ratio and absolute scales. Definitions and proper types of statistics and statistical tests for each type are described in [Wohlin00] [Cooper01] [Fenton97]. Usually qualitative research is mostly concerned with measurement on the nominal and ordinal scales, while quantitative research mostly treats measurement on the interval and ratio scales. Hypotheses with nominal and ordinal data are tested with *non-parametric tests*, while *parametric tests* are used for data derived from interval and ratio measurements and are more powerful. The choice of test depends also on whether one sample or more than one sample of data are available, and whether the distribution of variables are known and is normal. Parametric tests are more powerful when the distribution of variables is known. However, if the distribution is unknown, non-parametric tests are more appropriate, which are also effective with small sample sizes [Kitchenham02]. Examples of parametric tests are *Z test* and *t-test* for one sample case or two independent samples. Example of a non-parametric test is the *chi-square* one-sample test. The chi-square test can also be used for ordinal data and with several samples. Cooper et al. have useful examples on these tests [Cooper01].

When we measure something, we either want to assess something or to predict some attribute that does not yet exist. The second goal is achieved by making a *prediction* *system*; i.e. a mathematical model for determining unknown parameters from known ones. A software metric should be validated to make sure that the metric is a proper (numerical) characterization of the claimed attribute (i.e. assessing construct validity). Prediction systems should be validated as well: the accuracy of the prediction system should be studied by comparing model performance with known data. For example, Boehm specifies that the COCOMO effort-prediction system will be accurate to within 20% under certain conditions [Boehm95].

Defining metrics and collecting related measures in an organization need resources and is costly. Determining what to measure is not a trivial task.

The Goal/Question/Metric (GQM) approach is based upon the assumption that an organization must define goals for itself and its projects and trace these goals to metrics by defining a set of questions [Basili84]. GQM has gained much respect since it emphasizes the role of metrics; i.e. metrics should be goal-driven and relevant.

Others emphasize that for researchers it is also important to relate goals to theories and models. Kitchenham et al. write, "Although GQM ensures that measures are useful, simple and direct, it cannot ensure that they are trustworthy (or repeatable) and timely (since it is not concerned with how data collection maps to the software process in a manner that it ensures timely extraction and analysis of measures)" [Kitchenham01]. Another approach to defining metrics is the process-oriented one, defining when data should be collected (for example appropriate metrics for a workflow in RUP). It seems that a measurement program should combine a goal-driven approach with a process-driven one.

Pfleeger describes some lessons learned in building a corporate metrics program [Pfleeger93]. The author writes:

- Software engineers need tools and techniques to minimize their metrics duties.
- Engineers would collect and analyze metrics thoroughly and accurately only when the metrics met a specific need or answer an important question.

Paul writes that the selection criteria for metrics should include *usefulness*, *clarity* and *cost-effectiveness* [Paul96].

One challenge in data analysis is combining data from multiple sources, either in collection or analysis. A data set consists of data from all projects in a company or different data within a product. Kitchenham et al. warn about combining all data that happens to be available as it may result in invalid conclusions [Kitchenham01].

4.5 Summary and the Challenges of this Thesis

When planning a thesis like this, several questions should be answered. The numeration of research challenges from Sections 2.7 and 3.5 is followed:

RC7. Defining research questions. What are the research questions and how well are they formulated? Sometimes the research question is well defined, making it easier to decide research method. In most cases, however, the research question is emerging and so is the strategy. In this thesis, RQ1 was defined as a pre-study of software process improvement work, RQ2 was derived from bottom-up analysis of data, while RQ3 was originally defined to focus on improving GSN RUP, but was gradually revised to focus on other aspects.

RC8. Choosing research strategies. What research strategy should be chosen to answer the research question(s)? The quantitative, qualitative and mixed-method strategies are discussed. Case studies are valuable in answering how development approaches are implemented, what the results are and why the results are as they are. A mixed-method research approach allows emerging research design and collecting different types of data. Therefore, a mixed-method design is chosen that combines results of surveys, experiments, quantitative analysis of industrial databases, and qualitative study of software processes and development practices.

RC9. Collecting and analyzing data. How should data be collected and analyzed? The selected metrics and statistical tests are described in the papers.

RC10. Interpretation. How useful, innovative and valid are the results? The validity threats that are relevant for individual studies are discussed in the papers. In Section 7.4, validity threats for all the studies are also discussed. Usefulness and innovation is addressed when discussing results.

Empirical research in software engineering meets several challenges in general:

- As a field with a few decades of history, most research methods are borrowed from other disciplines. It started with statistics, while in the recent years the community has increasingly used methods from the social sciences in empirical software engineering research. These methods should be adapted for software engineering.
- Data is scarce in software engineering, it is very context-dependent and therefore is hard to analyze. Mcgarry emphasizes, “When it comes to measuring software, every project is unique” [McGarry01].
- Quick changes in technologies do not allow proper evaluations before use and feedback after use.

Performing case studies in industry is useful to meet all these challenges; i.e. to evaluate methods when facing the context, to gather useful data for researchers, and to evaluate technologies for researchers and practitioners.

5 *Research Context*

In this chapter, the Ericsson context is presented with more details on the GPRS system, GSN RUP, the component framework and the development environment than what is presented in the papers in Part II of this thesis. Software engineering challenges in developing large systems and some characteristics of the telecom domain are discussed. Furthermore, the research in this thesis is described in the context of the INCO goals. Finally, the research design is presented, which combines quantitative and qualitative studies, and top-down confirmatory studies with bottom-up explorative approach.

5.1 The Ericsson Context

5.1.1 About the Company

Ericsson is an international telecom company with development and sales units all over the world. It has approximately 40 000 employees at present. Ericsson has developed software for many years. It has sound traditions and long experience in development, quality assurance and how to launch complex networks. This study has used data from the GPRS (General Packet Radio Service) system, which is developed and tested in Ericsson organizations in Norway, Sweden, Germany and Canada. Currently 288 operators around the world have commercial GPRS services. Ericsson is the supplier to over 110 of these. Having provided more GPRS networks worldwide than any competitor, Ericsson is the world's leading GPRS supplier [Ericsson04a]. The development organization in Grimstad has been involved in developing software for GPRS from 1997 to 2003.

5.1.2 The GPRS System

Telecommunication and data communications are converging, and the introduction of GPRS in the cellular networks is a step towards this convergence. GPRS is a new non-voice value added service that allows information to be sent and received across a mobile telephone network. It supplements today's circuit switched data and Short Message Service (SMS) [GSM04]. GPRS provides a solution for end-to-end Internet Protocol (IP) communication between a Mobile Station (MS) and an Internet Service Provider (ISP) or a corporate Local Area Network (LAN). It is also expected that GPRS combined with the Internet Protocol version 6 (IPv6) will initiate a large growth trend within machine-to-machine (m2m) communication. Theoretical maximum speeds of up to 171.2 kilobits per second (kbps) are achievable, but the actual speed is lower (115 kilobit per second). The information in this section on the GPRS system is from Ekeroth et al. [Ekeroth00].

The GPRS Support Nodes (GSNs) are parts of the Ericsson cellular system core network that switch packet data. The two main nodes are the Serving GPRS Support Node (SGSN), and the Gateway GPRS Support Node (GGSN). The generic term GSNs is applicable to both SGSN and GGSN, which pertain to the commonalties and strong functional relation between the two nodes.

Figure 5-1 shows an example of the GPRS solution in a GSM network. GSNs are also used for GPRS domains within a Universal Mobile Telecommunications System (UMTS, using Wideband Code Division Multiple Access or W-CDMA) or Time Division Multiple Access (TDMA) system. SGSNs can be delivered for a pure GSN network, a pure UMTS network or combined for both.

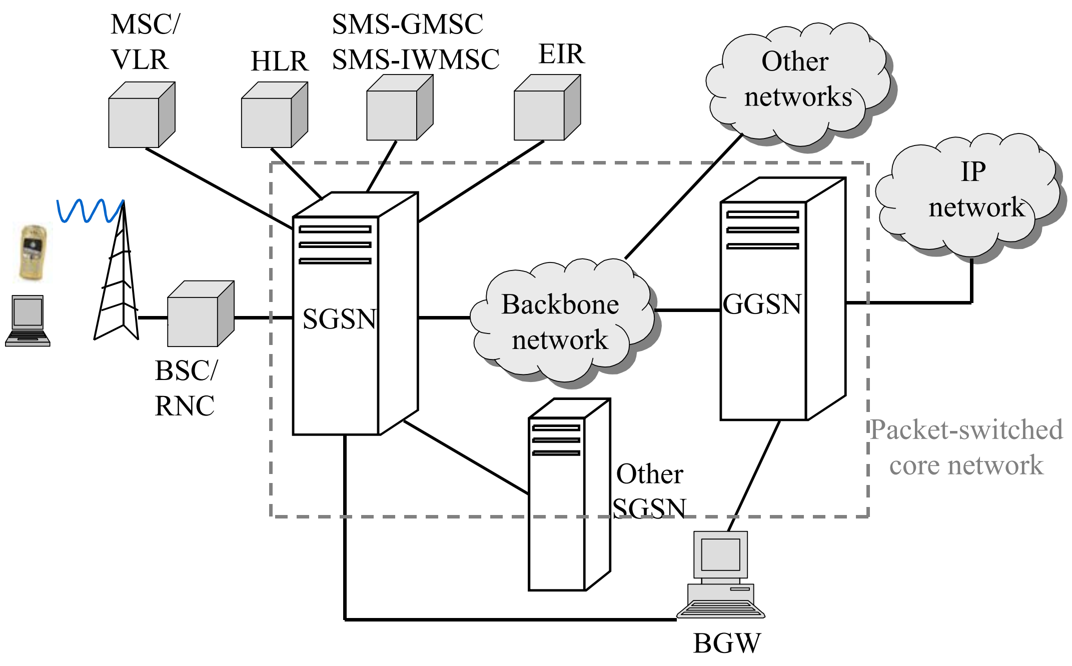


Figure 5-1 The Ericsson GPRS solution in a GSM network

The SGSN node keeps track of the individual MS's location and performs security functions and access control. The SGSN is connected to the GSM base station system through the G_b interface (and/or to the UMTS Radio Access Network through the I_u interface). The SGSN also interfaces other nodes in the network as shown in Figure 5-1 and the GGSN node.

The GGSN node provides inter-working with external packet-switched network. GGSN is connected with SGSNs via an IP-based backbone network. The other nodes in Figure 5-1 are:

– Home Location Register (HLR) that contains subscriber information.

- SMS-GMSCs (Short Message Service Gateway MSC) and SMS-IWMSCs (Short Message Service InterWorking MSC) supports transmission of SMS towards the MS via the SGSN.
- Mobile Service Switching Center/Visitor Location Register (MSC/VLR).
- Equipment Identity Register (EIR) contains a list of e.g. stolen mobile phones.

Standards from European Telecommunications Standards Institute (ETSI) and Third Generation Partnership Project (3GPP) specify interfaces between these nodes and the GSNs. The Ericsson implementation of GPRS is compliant with the generic GPRS architecture as specified by ETSI and 3GPP. Statement of Compliance documents (SoC) gives information on which parts of the respective standards that are supported by Ericsson and which parts that are not supported or just partly supported.

The GPRS system is required to be highly available, reliable and secure. It should handle defined Quality of Service (QoS) classes and enable hardware and software upgrades. It should also handle a high number of subscribers (several hundred thousands) and offer them real-time services. Another important requirement is scalability; i.e. to be configurable for different networks with high or low numbers of subscribers. The system has a distributed architecture consisting of several processors to meet the reliability and scalability requirements. The internal software bus is replicated and so are several interfaces.

5.1.3 Software Architecture Definition and Evolution

Software architecture is described at different abstraction levels using several UML models and views from RUP: logical view, dynamical view, implementation view, process view, physical view and deployment view. Only a simplified model of the logical view is presented here.

Software for the GSNs run on the Wireless Packet Platform (WPP), which is a platform developed in parallel with the GSNs by Ericsson. WPP includes several processors that the software is running on and also interface boards that connect the nodes to other nodes in the network. Figure 5-2 shows an overview of the initial software architecture of GSNs for the GSM network. The system is decomposed into a number of *subsystems* based on a functional requirements and interfaces, as well as optimization for non-functional requirements.

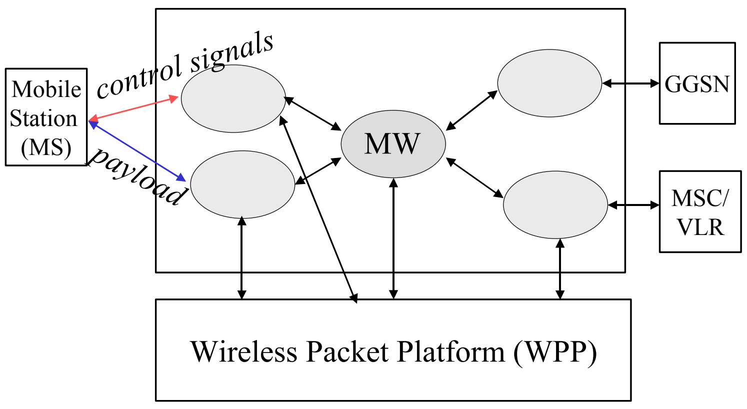


Figure 5-2 The initial software architecture of GPRS

For example, a MS sends two types of traffic to a SGSN node: control signals (to set up a connection, handling mobility etc.) and the actual payload traffic. These are handled by different subsystems since these signals have different non-functional requirements. Control signals require reliability and persistent storage of data, while data packets need high throughput but can tolerate some loss of packets. Besides, there are a number of subsystems for other functionality such as handling interfaces to other nodes or charging. The middleware subsystem (MW) handles broking, resource management, transaction handling and other middleware functionality on the top of WPP.

With standardization of GPRS for the UMTS market, Ericsson decided to develop the new SGSN using the same platform and components used for SGSN in the GSM market. This was the result of one year of negotiations and re-engineering. The origin of this decision was common requirements for these two systems. The method to initiate software reuse between these two products were:

a) Identify commonalities between the two systems.
b) Analyze the existing solution for SGSN in the GSM market to identify reusable parts.
c) Develop an architecture that has the potential to be reused and be evolvable for the two systems.

The evolved architecture is shown in Figure 5-3. Old subsystems are inserted in the layers based on the reuse factor, while some of these were split into two subsystems. The MW subsystem is extended to a component framework to support all subsystems with a lot of tasks, e.g. distribution, start and supervision of application logic, node internal communication services, an extended ORB and resource handling. The component framework consists of both run-time components and design/implementation rules to be followed. All components within the component framework are generic; i.e. not aware of 3GPP/ETSI defined concepts and behavior, and thus are reusable in any packet handling application.

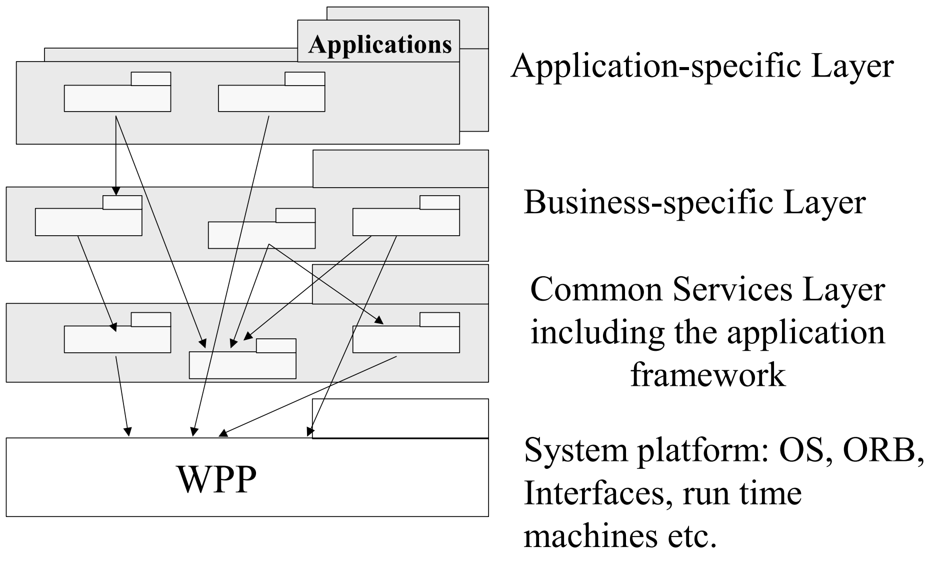

Applications
Application-specific Layer
Business-specific Layer
Common Services Layer including the application framework
WPP
System platform: OS, ORB, Interfaces, run time machines etc.

Figure 5-3 The evolved software architecture of GPRS

On top of the component framework and WPP, the applications should provide all 3GPP/ETSI specified functionality. The functionality on the application level that is shared between applications is grouped into a separate package, called for business-specific functionality.

Applications using this common platform were initially GGSN and SGSN nodes, but GGSN is now moved to another platform. There are now two SGSN nodes for GSM and W-CDMA markets sharing this common platform. These are called *applications* in this thesis, named SGSN-G and SGSN-W.

Applying design rules and design patterns reuses design, while the reused entities ("components") in the software architecture are subsystems. Identifying the reusable entities was done by evaluating candidate subsystems and if necessary splitting these into smaller ones with reuse potential (moving application-specific logic to other subsystems).

5.1.4 Development Environment and Tools

The high-level requirements are written as plain text in the Application Requirement Specification (ARS) and later stored in the Rational RequisitePro tool. UML is used for modeling, using the Rational Rose tool. Programming languages are Erlang, C, Java (mainly for GUIs), Perl and other script languages (for packaging and installation). Communication between modules in different programming languages is done by using CORBA IDL and an extended ORB. IDL files are compiled to generate skeletons and stubs.

The Rational ClearCase tool is used for CM. All files making a delivery are packaged and labeled with a release label. Scripts and makefiles define the contents of a delivery. Various testing tools are used, both simulated environment and real test environment.

To handle changes in requirements or implemented artifacts, Change Requests (CRs) are written as plain text and are handled by a Change Control Board (CCB). Defects detected in system test or later are handled by the trouble reporting process. These processes are further described in [P8] and [P10].

5.1.5 Components and Component Models

Figure 5-4 shows the hierarchical decomposition in the design model. A *subsystem* is modeled as a package in the Rational Rose tool, has formally defined interfaces in IDL and is a collection of function blocks. A (function) *block* has formally defined interfaces in IDL, is a collection of lower level (software) units and is also modeled as a package. A block often implements the functionality represented by one or more analysis classes in the analysis model. Using IDL for interface definition gives language independence. Both subsystems and blocks are mapped to *components* in the implementation view, and are termed high-level and low-level components. While blocks are tightly coupled inside and provide a coherent set of functionality, subsystems are packaging of blocks that belong together in the solution domain (for example middleware, or mobile handling).

A (software) *unit* is a collection of (software) modules and is modeled as a package. Two units within the same block may communicate without going through an interface, but in case these are developed in different programming languages, a formal interface has to be defined even within a block.

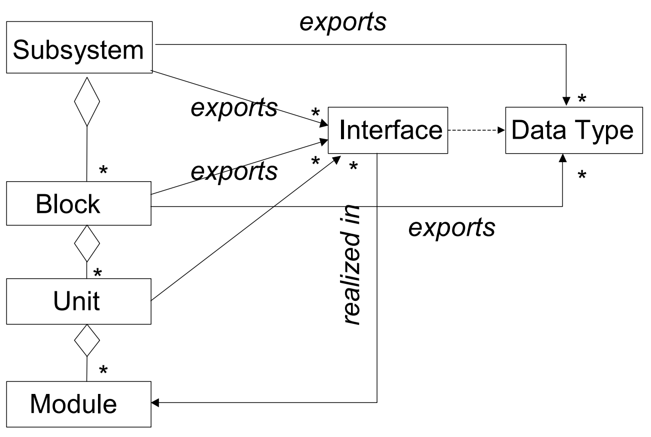


Figure 5-4 Decomposition of logical entities

As described, components are logical entities that are realized as executable entities. The number of subsystems is low and they represent large-grained packages of functionality. Their interfaces are facades to lower level components; i.e. blocks. Components have explicit provided interfaces, while required interfaces are shown as dependencies in the design models. Typically, components have no configuration or test interfaces either.

Components in the three upper layers are developed in-house and are not subject to third-part acquisition. There is one instance of each component in a node and components are stateless. Data for each subscriber is stored in different tables stored in a database, which is part of the Erlang run-time environment. Neither Erlang nor C is object-oriented. Although the initial modeling in the analysis view is done using objects (for example an object is assumed to be instantiated for each MS), code for these objects is later spread over software modules and data is stored in multiple databases. To keep the data for each MS consistent, there are programming rules that define which software module owns which part of data. This is an industrial example of combing object-oriented design with non-object-oriented programming languages. The situation is confusing for new staff, but may be unavoidable since new tools such as the Rational Rose tool are developed for object-oriented design.

There are multiple component frameworks (models) in this case:

a) The CORBA component model, which is used for communication between GUIs and other parts of a node. GUIs are used by operators or maintenance staff.
b) The GPRS component framework defines its own extended ORB and middleware services for applications.
c) The component framework and applications use the application development environment in WPP, i.e. a framework that is plugged into another framework.

This complexity has several reasons:

- Multiple programming languages.
- The component framework offers many services in addition to services offered by WPP.

5.1.6 The Software Process Model

As mentioned, Ericsson uses a tailored or adapted version of RUP, called GSN RUP[3]. A joint Ericsson team in Norway and Sweden has worked continuously with adapting and maintaining RUP, as part of the Method & Tools workflow. Figure 5-5 shows the start view of GSN RUP. Comparing Figure 5-5 with Figure 3-2 (standard RUP) shows the following differences:

- Ericsson Tollgates (TG) replace milestones in RUP. The main purpose of a TG is to decide whether or not to continue into the next stage of a project.
- Business modeling is excluded, since it is done in other parts of the organization.
- A *Conclusion phase* is added, to summarize experience.
- Method & Tools is the same as the Environment workflow.

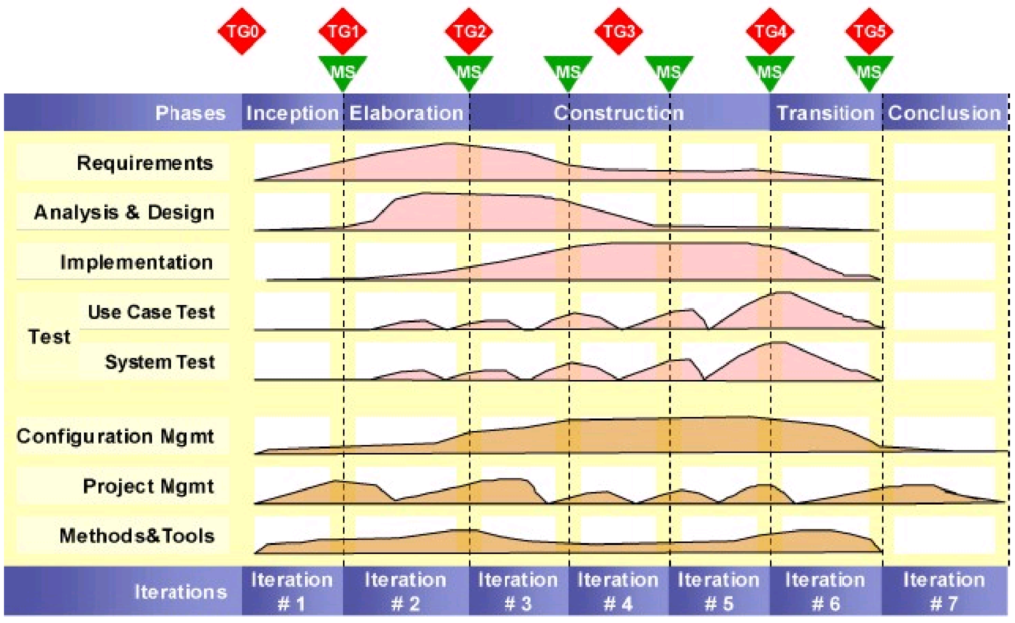


Figure 5-5 The start view of GSN RUP

- Test is divided in two workflows: Use Case Test for testing separate use cases (may also be done in simulated environment) and System Test.
- The Deployment workflow is removed, since it is done in other activities.

Each workflow is also adapted. Some examples are:

[3] There is another adaptation of RUP at Ericsson, called ERUP. The GPRS projects have developed and used their own adaptation (GSN RUP), which is studied in this thesis.

- Because of the importance and complexity of non-functional requirements, the role “non-functional specifier” is added. It should do the activity “detail non-functional requirements”.
- The role “database designer” is removed, since the system uses a database included in the platform.
- RUP roles are mapped to Ericsson positions.
- Design and modeling guidelines, and various templates for coding, documentation, CR handling etc. are linked to the related workflows
- ARS replaces the RUP’s vision and stakeholder request document for the product. Requirement workflow also includes SoC (Statement of Compliance) artifacts, which point out the parts of the standards that are implemented, and the Feature Impact Specification (FIS) documents.

The FIS documents have several roles in different phases:

1. **Before TG0.** Requirements may come from different sources such as ARS, SoC or Change Requests (CRs). Information in these sources is complementary, or sometimes conflicting and should be merged. Cost, impacts and risks for each requirement should be clarified. A use case model does not measure the impact of a requirement on the system. Furthermore, requirements in the ARS are defined as features and it is not clear how to map features to use cases. The FIS looks at the problems listed above and captures the requirements in the ARS, SoC and CRs together to find what impact a requirement has on the system. The responsible for FIS in this stage is the Product and System Management.
2. **TG0-TG1.** More information on requirements fulfillment and an estimate of the impact on each system component is added. The responsible for FIS in this stage is the Pre-study project.
3. **TG1-TG2.** Further breakdown of the implementation and an estimate of the impact on each subsystem component are done. The responsible for FIS in this stage is the Development project.

Ericsson has a tradition of defining requirements as features. Furthermore, product lines or families often define requirements as features. Requirements are divided in two major groups in most literature:

- *Functional requirements* that are concerned with functionality of the system as observed by end users (end users may also cover other systems, operators etc.) and are specified in use cases or features.
- *Quality (or non-functional) requirements*, including requirements that are specific for some functionality (e.g. charging capacity) and all other requirements that are not specified by use cases. Quality is the degree to which software meets customer or user needs or expectations.

A feature may be a functional (e.g. multiple contexts for a MS) or a quality requirement (e.g. interoperability with other nodes, number of users, or reliability defined as continuity of service). Use case models and supplementary specification documents defined by RUP are not sufficient for each situation and are therefore combined with features. Classification of functional vs. quality requirement is not absolute. For example, security may be a quality requirement in one case and a functional requirement in another case (see for instance [Eeles01]).

While there is consensus on using the term *functional requirements* for requirements concerning business goals, other types of requirements are covered by different terms over time and classified differently in literature. They are sometimes called for *non-functional requirements* (an example is RUP), sometimes for *extra-functional requirements* [Crnkovic02], and in some literature for *quality requirements* leading to *quality attribute* of a system [Bosch00]. Sommerville uses the term *emergent properties* [Sommerville01], and finally [Bachman00] calls them for *extra-functional properties* or *quality attributes* or when associated with a service, *quality of service*.

Experience from using RUP as discussed in [P2], [P3], [P5] and [P6] provides the following advantages:

- RUP is presented in web pages with the possibility to link other documents and search for activities, artifacts and roles. These web pages are understandable and easy to use. This is confirmed by internal assessment of GSN RUP and the small survey reported in [P2]. The notation is also rich.
- RUP comes with a set of tools, such as Rational ClearCase for CM and Rational RequisitePro for requirement management. However, the degree of satisfaction with these tools is varying and could be subject of future studies.
- RUP is widely used in industry.
- RUP is extensible by adding plug-ins or RUP's extension mechanisms.

On the other hand, a generic process such as RUP is not suitable for every task:

- Managing requirements for reusable parts is not easy with RUP. RUP is use-case-driven and use cases are defined for observable functionality by a user or an operator. The project tried to define use cases for middleware as proposed in [Jacobson97], but it was not successful: the complexity grows in use case models and most services offered by middleware are not suitable for use cases, e.g. handling concurrency or distributed objects. Instead, textual documents were used for these requirements.
- Many tasks depend on domain-specific knowledge such as identifying components, defining suitable interfaces or identifying objects. Internally developed guidelines are therefore linked to RUP web pages. This means adapting RUP for a domain.

More details on GSN RUP are found in student project reports [Naalsund02] and [Schwarz02].

5.1.7 Data Collection and Metrics

The company has a dedicated team for measurement definition and for collecting and analyzing data. Both direct and indirect metrics are defined. Table 5-1 shows examples of direct measures. All the above metrics have a ratio or absolute scale.

Table 5-1 Examples of direct metrics defined at Ericsson

Name	Description	Purpose
Original Number of High Level Requirements	Total number of requirements, listed in the ARS at TG2	Calculation of Requirements Stability
New or Changed High Level Requirements	Total number of new or changed requirements, listed in the ARS between the TG2 baseline and the delivery date	Calculation of Requirements Stability
Size of Total Product	Total amount of non-commented lines of code in the product, this also includes generated code.	Calculation of Defect-Density
Size of New and Changed Code	Total amount of non-commented new and changed lines of code in the product, including new generated code.	Calculation of Defect-Density
Defects identified in Test	Number of valid trouble reports (duplicates and cancelled trouble reports excluded) written per test phase and after the first six months in operation	Calculation of Defect Detection Percentage, Defect Removal Rates and Defect Densities

Other types of direct metrics that are not defined in the measurement program but are used in various documents are:

- Classification of changes to requirements: new, removed or modified requirement.
- Classification of modifications: modified solution, modified documentation etc.

Metrics of the above types will have the nominal scale. Table 5-2 shows examples of indirect or derived metrics that are calculated by using direct ones and the relevant quality attributes.

This work has collected and analyzed the following quantitative data for 5 releases of SGSN-G (not all data was available for all the 5 releases):

- Data on inspections used in defining a baseline in [P4].
- Available direct and indirect measures as defined in Tables 5-1 and 5-2. These measures are used in interpreting the results, assessing development approaches [P5] [P11] and building a model as explained in Section 6.2.
- TRs stored as plain text. TRs are analyzed in [P8].
- CRs stored as plain text. CRs are analyzed in [P10].
- Size of total products, components and modified code between releases in KLOC. These measures are used in calculating defect-density and modification rate in [P8] and change-proneness in [P10].
- Data on effort, used for studying effort distribution in [P12] and calibrating the UCP estimation method in [P13].

Table 5-2 Examples of indirect metrics defined at Ericsson

Name	Description	Purpose	Quality Attribute(s)
Requirements Stability (Percent)	Percent of high level requirements listed in the ARS not changed between TG2 and delivery	To check the stability of requirements	Stability, need for Extensibility
Defect-Density (No. of Defects/KLOC)	Defects identified/total code and Defects identified/new & modified code	To check the quality of product and work performed	Dependability/ Reliability
Productivity (Person-hours/LOC)	Total hours used in project, divided with total number of new and modified lines of code	To check project productivity	Process compliance
Planning Precision (Percent)	Absolute value of actual minus planned lead time (in number of weeks) divided with planned lead time multiplied with 100	To check project lead time	Scheduling capability

5.2 The INCO Context

This thesis is part of the INCO project [INCO01] and the four project goals are presented in Section 1.2. The focus of this thesis was initially defined to be on software reuse, CBD and Software Process Improvement (SPI) to primarily advance the state of the practice in industry and to learn from experience. This focus gradually changed in two dimensions:

- Due to reorganizations in Ericsson, the organization in Grimstad was put in a transition phase. Thus, SPI initiatives were not feasible. Instead, empirical work was started in the form of quantitative studies and combining these with qualitative data.
- These studies showed that reuse and CBD should be considered together with the incremental approach, e.g. in the study of CRs or effort.

5.3 Developing Large Systems

Data from several releases of one of the products in Figure 5-3 are used in the studies in this thesis. The system is large; i.e. about 450 KLOC in multiple programming languages or over 1 million KLOC in equivalent C code. It took 5 years to build a system of this size. Large systems are complex and complexity leads to many challenges in development and maintenance, in different dimensions. Large systems are developed to be long-lived; i.e. systems should be evolvable and maintainable for several years. Telecom systems may even be in use for decades. An example is the Ericsson AXE switch, being the most widely used switching system in the world, presented first in 1974 [Ericsson04b]. Ericsson writes, “From its inception, the AXE system was designed to accommodate continuous change. Over the years, its array of functions has grown and its hardware has been steadily updated”. AXE serves as a platform for every type of public telephony application, which explains the reason for its long live as a generic system. The GPRS system may also be used for many different services and thus has the potential to be used for several years.

Challenges in engineering large systems are classified in three classes in Table 5-3 and examples are given of how these are handled at Ericsson. As shown in Table 5-3, incremental development, software reuse, product family development and CBD are used as means to handle some of these challenges. Large systems are developed in multiple programming languages, since different parts should be optimized for different quality attributes such as understandability, performance or memory usage. For the last row, the duration of iterations represents a tradeoff between short iterations and the extra resource needed to plan, integrate and validate increments. Each release usually undergoes 5-7 iterations.

In an attempt to find reusable components by reverse engineering of large system, James Neighbors described that the most successful method was to identify subsystems that are tightly coupled inside in data and function and use these as domain-specific reusable components [Neighbors96]. Identifying reusable components in the GPRS system has also been done at the level of subsystems. Neighbors also refers to the industrial experience of using naming conventions, and the fact that complex systems have common problems such as poor documentation, deviation from standard design and complex arrays of versions and features.

Some new trends in industry, especially relevant for large companies are:

- Companies are increasingly using mainstream methods, tools and programming languages instead of proprietary ones. For instance, RUP has replaced internal processes and sometimes is combined with elements of these such as Ericsson TGs. Most other tools for requirement management, test and CM are also commercial, widespread ones.
- Outsourcing of tasks is a new trend in industry. Parts of a product or special phases may be outsourced. However, outsourcing of critical parts is a risk that few companies take. After organizational changes at Ericsson, several experienced personnel were employed by another company based on a contract with Ericsson to take over the maintenance of earlier releases. This outsourcing was possible because of access to personnel with first-hand knowledge of the system.

Table 5-3 Challenges facing large system development

Challenge	Description	Handling challenges, Ericsson
Organizational	Financial investments	- Incremental development: it is important to sell working releases to provide cash flow. - Reuse and product family development: to develop faster, cheaper and better.
	Human resources: - Resources have to be moved gradually from other projects or hired in the expansion phase. - In the transition phase, most resources are moved to other projects, except for the maintenance staff.	- Access to experienced personnel with domain knowledge and experience from the AXE systems. - Ericsson has globally reduced its number of staff drastically and has outsourced many tasks.
Technical	Tools: it is important to validate that tools can handle development in the large.	Lots of effort is used in developing routines for CM (using the Rational ClearCase tool).
	Design: - Decomposition of large systems into units that can be independently developed and maintained. - Composition of large systems. - Selection of programming languages and development environment. - It is important to validate that selected methods do scale up for a large project. - Large systems are long-lived and should be maintainable and evolvable.	- Traditional decomposition into subsystems, blocks etc. (with tight coupling inside, but minimized external coupling) and combined with CBD. - Multiple programming languages. - Strict naming conventions are used at all levels, from models to source code. - Previous experience with developing large systems (new methods are usually combined with industrial experience). - The system is designed to allow hardware and software updates.
Process	- Assigning functionality of right size to iterations and releases. - Selection of a software process model.	- Duration of iterations is 2-3 months. It is almost one year between major releases. - RUP is adapted for the context (see Section 5.1.6).

- Large companies start joint projects for developing new standards, tools and processes; either with companies in the same domain or other domains. An example of co-operation between telecom companies is the Bluetooth Special Interest Group where Ericsson, Nokia, IBM, Agere, Intel, Microsoft, Motorola, Toshiba and thousands of other member companies drive the development of the Bluetooth wireless technology for short-range connection of mobile devices [Bluetooth04]. An example of co-operation with companies in other domains is Ericsson's partnership with Rational, announced in 1999, under which Rational provides Ericsson an integrated set of development tools, processes and services [Ericsson99]. Another well-known co-operation example is developing different standards such as CCM and UML standards in OMG [OMG04].

The main impact of the above trends for software engineering research is a more standardized view of software development, although products vary a lot. Results of empirical assessment of methods and tools get therefore more interesting for others using the same methods and tools.

Some specific characteristics of the GPRS system (also applicable for other large telecom systems) are:

- Personnel turnover has traditionally been small. Telecom industry has not the large turnover as typical IT companies. However, during the recent years, telecom companies have also reduced their staff drastically, including Ericsson.
- Non-functional requirements such as reliability, availability, performance and evolvability or maintainability (see Section 5.1.2) are of great importance. Network nodes should be available almost all the time, have high throughput, and evolved and maintained for several years. The share of different types of non-functional requirements is also reflected in the CRs [P10]. The importance of these requirements led to a focus on quality metrics such as defect-density and change-proneness in RQ2.
- Software is developed for network nodes and there are no direct user interfaces, except for operation and maintenance. Therefore, usability of these interfaces is not as important as for instance for web applications. On the other hand, usability and quality of documentation is important for system operators. This is also reflected in the share of CRs related to documentation [P10].
- Interfaces are protocol-based and governed by international standard organizations. The system should comply with these standards to be used in networks of different telecom operators.
- Components are developed in-house, although in some cases by different Ericsson organizations. Research challenges related to COTS software are therefore excluded in the research questions and design.
- Systems should undergo final test and tuning in a customer site, which may explain relative high number of TRs in system test [P8].

The large product size justifies investments in adapting RUP, developing a component framework, and effort spent on integration and testing. For research in this thesis, the large product size explains why development methods are evaluated and adapted for large systems in the development of an estimation method [P13] and improving inspection techniques [P4]. When discussing the results, the external validity of the results and the possibility of generalization to other domains are discussed, having the above characteristics in mind.

5.4 Research Design in this Thesis

The research has combined qualitative studies of the software process and the related practice, with quantitative studies of archived data and experiments. It has further combined the results to propose improvements in some areas. The rationale for combining studies of different types has been:

- The impact of introducing reuse or incremental development is widespread.
- Studying an industrial case from the inside gives the possibility to collect different types of data. It is useful to take benefit of all available data.
- The results of one study should be confirmed by other studies; i.e. triangulation of data.

The system of this study has the following characteristics:

- It is a large industrial system with characteristics described in Section 5.3. The reused assets are used in two business critical systems.
- The company initiated a reuse program across organizations and countries. The approach required a lot of coordination between development organizations in different countries (both technical and management issues). The approach was an extractive one and involved high risks regarding cost, quality, training, coordination and management support.
- Several releases of an industrial system are studied. This is necessary to understand incremental development and a product family approach, where effects cannot be identified immediately.

During the period of this doctoral work, the telecom industry (or more generally IT companies) met a crisis that resulted in deep cuts in resources and major changes in their profiles. Ericsson has reduced its personnel from over 100 000 to 40 000 in 3 years and centralized its research and development in a few countries. The GPRS development organization in Grimstad was closed down in 2002. Some development and maintenance is outsourced to a company in Grimstad, employing experienced personnel from Ericsson for these tasks. The responsibility for future development was moved to an Ericsson organization in Sweden. The research questions and design were therefore revised between phase one and two of this doctoral work as described below.

Selecting research questions and research strategies has been both top-down and bottom-up:

a) Some research questions and hypotheses were identified from earlier work on software reuse, in the context of INCO and the product family approach. RQ1 aims to describe the decision on software reuse in the context of Ericsson. RQ2 aims to empirically assess some earlier claims on the benefits of reuse. The questionnaire in [P2], Hypotheses in [P8] and some hypotheses in [P10] are based on earlier work. The experiment on inspection methods reported in [P4] is also based on earlier work on the OORTs.
b) Other questions and hypotheses are results of exploratory work on available data and practices in the industry, in a bottom-up style. Some hypotheses in [P10],

identifying metrics in [P11], the estimation method proposed in [P13], observations related to effort distribution in [P12] and the data mining method in [P11] present new ideas and hypotheses that are grounded in the data.

The results of studies in the first group can be more easily merged into the body of existing knowledge. One general concern regarding the results of studies in the second group is the generalizability of the results. This is later discussed in evaluating the results.

There has been three phases in the course of this work, as shown in Figure 1-1 and Table 5-4 (cf. Section 4.5, RC7- Defining research questions):

1. The first phase consists of qualitative studies of the software process and related practices, and a survey to increase the understanding of practice. It also contains an experiment with the goal of improving the practice of inspections. This phase has impact of the top-down approach to the research design.
2. The second phase is identified by quantitative studies of TRs, CRs and effort, with the goal of assessing the impact of development approaches and exploring new knowledge. This phase starts with a top-down confirmatory approach and continues with more bottom-up explorative studies.
3. In the third phase, the results of several studies are combined in a mixed-method approach to reflect on the research method and interpret the results. An estimation method is also developed and metrics for a combination of reuse and incremental development are identified.

5.5 An Overview of Studies

The research questions and relations to the studies, together with type of studies and phase are shown in Table 5-4.

Table 5-4 Type of studies and their relations to Research Questions (RQ), papers and phases

No.	Studies	Type	RQ1	RQ2	RQ3	Paper	Phase
1	Study of reuse practice	Qualitative, descriptive study of textual documents and web pages and own experience.	√			P1 P7	1
2	Study of software process and RUP	Qualitative, descriptive study of textual documents and web pages and own experience.	√		√	P6	
3	Survey of developers' attitude to reuse and software process	Exploratory (small) survey, Quantitative.	√			P2 P9	
4	Study of MDA	Qualitative and exploratory study of MDA, prototyping.	√			P3	
5	Experiment on inspection	Experiment (quantitative) on adapted inspection technique in the context of incremental development.			√	P4	
6	Study of TRs	Quantitative study of data repositories, confirming existing theories in the context of reuse.		√		P8	2
7	Study of CRs	Quantitative and exploratory study of data repositories. New hypotheses.		√		P10	
8	Study of effort distribution	Quantitative study of databases, exploratory. New hypotheses.		√		P12	
9	Developing estimation method	Quantitative study. Adapting existing method for new context.			√	P13	3
10	Identifying metrics	Qualitative, combining the results of studies 6-9.			√	P11	
11	Assessing development approaches	Qualitative, combining the results of studies 2, 5, 6-9, and internally gathered measures.		√		P5	
12	Developing a data mining method	Qualitative, combining the results of studies 6-8.			√	P11	

6 Results

This chapter summarizes the results of the research in three sections on software process, assessing development approaches and proposals for improving the practice. Most results are reported in the papers, but this chapter presents some data that are not yet published and discusses their impact on the results.

6.1 Software Process - RQ1 and RQ3

Six papers are presented in this section: P1, P2, P3, P6, P7 and P9. These papers describe experience with the current software process (related to RQ1) and proposals for adapting GSN RUP for reuse (related to RQ3).

P1. Experiences with Certification of Reusable Components in the GSN Project in Ericsson

This paper describes the reusable artifacts across two telecom systems, where software architecture, including design patterns and guidelines, has a major impact both on functionality and quality. The two systems are developed in different Ericsson organizations. A positive experience with reuse is that organizations have easier access to skilled personnel and shorter training periods.

Certification by third party or a trusted authority can accelerate component acquisition. For components developed in-house, the company itself does the certification. While functional requirements may be mapped to specific components, quality requirements depend on software architecture, several components or the whole system and the software development process. The paper describes how the software architecture and components are certified, especially for quality requirements and reusability. Maintainability should be observed over time, but the software architecture should initially be designed for maintainability. The paper suggests improving the reuse practice in the form of a revised RUP process and a suitable reuse metrics.

Discussion. Bachman et al. write, "The value of certification is proportional with the strength of prediction made about end-system (or strength of compositional reasoning) [Bachman00]. Both components and compositions are subjects of prediction. However, mathematical and formal prediction has not yet been possible (if it ever would be for systems that are not developed by formal methods). Section 2.4 referred to Voas on the difficulty of composing *itilities* [Voas01]. This paper confirms the role of software architecture in implementing quality attributes, while prediction of the system behavior is done by domain expertise, prototyping, simulations and early target testing (especially the operational quality attributes).

P2. Reuse in Theory and Practice: A Survey of Developer Attitudes at Ericsson

The paper describes the state of the software process model, which is an adaptation of RUP. The existing process model is not, however, adapted for reuse. That is workflows are described as if there is a single product development and there is no explicit framework engineering. To provide the information needed by developers, artifacts such as internally developed modeling guidelines and design rules are linked to the workflows in RUP. But these artifacts are also far from mature regarding reuse. The paper suggests that it is important to synchronize the software process model with the development practice.

An internal survey was performed among 10 software developers (9 responses) to explore their attitudes towards the existing process, and to identify and plan aspects that can be improved. The results of the survey showed that design was considered as the most important artifact to reuse, and that participants assumed reused components to be more stable and causing fewer problems than a new one (which is later confirmed by quantitative analysis in [P8]). Although the RUP web pages are frequently used, the main source of information during analysis and design was previous work and consulting in-house experts. The results also showed that the lack of explicit guidelines on reuse might lead to insufficient documentation of reusable artifacts and difficulty in assessing components for reuse. Developers did not consider a reuse repository as critical, as confirmed by other studies [Frakes95]. Poulin discusses three phases of the corporate reuse libraries as [Poulin95]:

1. Very few parts, *empty.*
2. Many parts of low or poor quality, *not to be trusted.*
3. Many parts of little or no use, *irrelevant*.

The paper proposes six major modifications to RUP. Table 6-1 shows these and other minor proposals.

Another central document in GSN RUP is the FIS document (described in Section 5.1.6), which could be adjusted for reuse with small modifications. It is proposed that requirement fulfillment of selected reused components should be discussed.

The questionnaire used in this survey and the improvement suggestions are part of a master’s thesis by two NTNU students [Naalsund02]. Further analysis of the results is done as reported in [P2].

Discussion. Bergström et al. suggest the following steps in adopting RUP [Bergström03]: create awareness of RUP, assess the current situation, motivate with a business case, set adoption goals, identify risks and opportunities, make a high-level adoption plan and a communication plan, and identify software development projects to be supported (pilot projects). This survey is an exploratory step in assessing the current process and to set improvement goals.

To keep a consistent view of the process, relevant RUP web pages should be updated. An example of updating the Analysis and Design workflow is shown in Figure 6-1, where the “Make versus Reuse versus Buy” decision is added as an alternative to designing components in-house. The list of reuse-supporting proposals is not complete and these proposals are not implemented due to the organizational changes in Ericsson.

Table 6-1 Adapting RUP for reuse

TG	Purpose with the phase and proposals on adaptation for reuse
TG0	Purpose: This tollgate is performed prior to the first iteration. It serves the purpose of deciding whether or not to initiate the project.
TG1-Prestudy/ Inception	Purpose: Establish the software scope and boundary of the project. Discover the initial use-cases (primary scenarios of behavior). Establish overall cost and schedule for the entire project and a detailed estimate of the elaboration phase. Estimate risks. **Proposed Reuse Activities:** 1.1. Plan reuse strategy and criteria for the evaluation strategy. *Decision Point 1*: "make vs. buy vs. reuse" decision: are we willing to depend on an outside vendor? Can we renegotiate the requirements (development *with* reuse)? 1.2. Domain analysis (analyze who may reuse the components we make in the future (development *for* reuse)).
TG2-Feasibility/ Elaboration	Purpose: Analyze the problem domain. Define, validate and baseline the architecture. Develop project plan. Eliminate high-risk elements. **Proposed Reuse Activities:** 2.1. Add the activities leading to the second "make vs. buy vs. reuse" decision (development *with* reuse). 2.1.1. Component identification and selection. 2.1.2. Component familiarization. 2.1.3. Feasibility study of COTS. *Decision Point 2*: "make vs. buy vs. reuse" decision 2.1.4. Renegotiation of requirements. 2.2. Update documentation (development *for* and *with* reuse).
TG3-TG4 Execution/ Construction	Purpose: Building the product and evolving the vision, the architecture and the plans until the project is completed. Achieve adequate quality within time limits **Proposed Reuse Activities:** (In each iteration) 3.1. Possibly run "make vs. reuse vs. buy" decision for minor parts (development *with* reuse).
TG5-Execution/ Transition	Purpose: Provide user support. Train user community. Market, distribute/sell product. Achieve user self-support. **Proposed Reuse Activities:** 4.1. Update reuse related documentation (development *for* and *with* reuse). 4.2. Update repository (development *for* reuse).
Conclusion	Purpose: define and store experience from the current software development project. **Proposed Reuse Activities (development *for* and *with* reuse):** 5.1. Conclude documentation. 5.2. Record reuse experience.

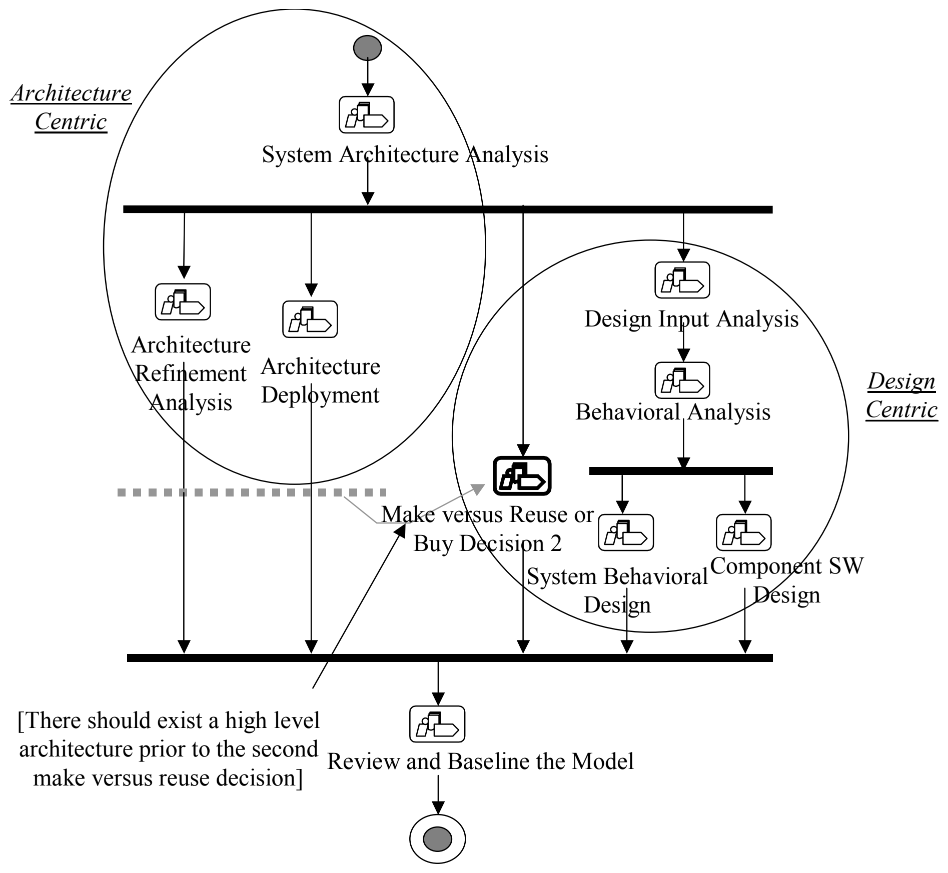

Architecture Centric
System Architecture Analysis
Architecture Refinement Analysis
Architecture Deployment
Design Input Analysis
Behavioral Analysis
Design Centric
Make versus Reuse or Buy Decision 2
System Behavioral Design
Component SW Design
[There should exist a high level architecture prior to the second make versus reuse decision]
Review and Baseline the Model

Figure 6-1 The proposed Analysis and Design workflow with reuse

No other empirical study on RUP regarding reuse has been found and this paper together with [P6] emphasizes the importance of assessing RUP in this aspect. The survey was also important for generating hypotheses that are later assessed in [P8] and [P9].

P3. MDA and Integration of Legacy Systems: An Industrial Case Study

This paper compares model transformations in RUP with transformations in MDA (Model Driven Architecture). Since moving from one model to another is done manually in RUP, there are inconsistencies between models, between models are requirements, and between models and code. Tools on the other hand, do transformations in MDA, but MDA is so far used only for new development. The paper explores how the legacy code could be transformed. Evaluating MDA tools and developing a prototype for reverse engineering of Erlang code to UML models were part of a master's thesis at HiA in spring 2003 [Warsun03]. The results suggest that:

– The concept of platforms is relative and so is a platform-independent model.

- It is hard to integrate legacy systems in the existing MDA tools and these tools are only useful for new development.
- MDA tools vary a lot in how much of the transformation and coding can be done automatically. Few tools support full definition of a system in models (i.e. both structure and behavior) and full code generation.
- Organizations can nevertheless learn from the MDA approach and keep their models synchronized with each other, with the requirements and with the code, even without applying a full MDA approach.

A prototype was developed that reverse engineers code in Erlang and interface descriptions in IDL, and builds structurally complete UML models, thus keeping UML models synchronized with the code. The paper suggests that if a company wants to use an MDA tool, it would be a better solution to wrap legacy software.

Discussion. Bennett et al. define a legacy system as “Software that is vital to our organization, but we do not know what to do with it”, and “Re-engineering is a high cost, high-risk activity with unclear business benefits” [Bennett00]. Sometimes it is the code, but also the data may be important for organizations to integrate with new systems and technologies.

Bennett et al. also write that although it seems obvious that having high-level design/architectural knowledge with traceability to high and low level design helps maintenance, there is little empirical evidence that this actually helps maintenance staff. The practice is that only the source code is maintained, and other representations become inconsistent and cannot no be trusted any longer. This problem may exist far before the maintenance phase. The inconsistency between UML models, and between UML models and code or requirements is not only a maintenance problem as a result of loss of architecture, but also important during development, especially in incremental development where each release builds on a previous release.

P6. Different Aspects of Product Family Adoption

The approach to initiating a product family in Ericsson has been a lightweight one and many artifacts are evolved during product family adoption, although not to the same degree. The paper describes the evolution of software architecture to support reuse and handling of variations, while the software process model is not updated for product family engineering and reuse. This paper discusses what works and what does not work in the software process (also described in Section 5.1.6).

Discussion. Different approaches to product families are discussed in Section 2.5. Johnson has emphasized that reusable components are not (pre)planned; they are discovered (gradually later) [Johnson98]. Ericsson chose the extractive or lightweight approach because of similarities in requirements between an emerging product and an existing one, and to reduce time-to-market for the new product.

P7. An Industrial Case Study of Product Family Development Using a Component Framework

The paper describes the role of an internally developed component framework in promoting reuse and how it is developed it in parallel with applications using it. Unlike component technologies like EJB or COM that are considered for implementation of components, domain-specific component frameworks include reusable designs for a specific domain. This knowledge should be integrated early into the development process of applications.

Discussion. The paper and Section 2.3 discuss that four important factors for the success of reuse are in place; i.e. top management commitment, commonality between products, domain engineering and experienced people. Adapting software process can be done gradually. As discussed in [P6], many aspects of software development should be adapted for product family engineering such as estimation methods, CM routines and metrics.

P9. A Study of Developer Attitude to Component Reuse in Three IT Companies

The paper combines the results of [P2] with similar surveys performed in two other Norwegian IT companies. It also studies the relations between the companies' reuse levels and satisfaction with documentation, efficiency of the requirements renegotiation process and trust to components. The companies' reuse levels are classified as large, medium and small, and all companies use in-house developed components. The results show that requirements re-negotiation may be necessary as for developing with COTS components. Furthermore, component repositories are not considered important. These two conclusions are independent of the reuse level. However, developers' satisfaction with the documentation of reusable components decreased with increasing reuse level and informal communication between developers supplement for this weakness.

Discussion. The study was exploratory and the surveys were small-scale. The results are used for generating hypotheses for future studies.

6.2 Assessing Development Approaches - RQ2

Four papers are discussed in this section: [P5], [P8], [P10] and [P12]. It also presents a model of the impact of development approaches on quality attributes. In addition to the referred literature in the papers, the excellent guidelines of the SPIQ project (Software Process Improvement for better Quality) [SPIQ98], [Mendenhall95], [Cooper01] and [Maxwell02] are used in statistical analyses and presentation of results. Statistical tests are done using Microsoft Excel and Minitab tools.

P5. Using Empirical Studies to Assess Software Development Approaches and Measurement Programs

Incremental development in Ericsson is described, with both features and use cases assigned to increments, and additional artifacts to handle integration of these into a release; i.e. an integration plan and an anatomy plan. It also discusses difficulties in gathering data in incremental development with overlapping increments. The paper combines some internally gathered measures, the results of the empirical studies in Ericsson and qualitative feedbacks to assess approaches to software development and the quality of the measurement program. Examples of metrics that are especially useful for such studies are given, and improvements to the methods and tools for collecting data in the company are suggested.

Discussion. Metrics that are proposed in this paper are further discussed in [P11]. The observations and quantitative results are used to propose a model of the impact of development approaches described at the end of this section.

P8. An Empirical Study of Software Reuse vs. Defect-Density and Stability

Four groups of hypotheses regarding the impact of reuse on defect-density and stability, and the impact of component size on defects and defect-density in the context of reuse are assessed. Historical data on defects (as reported in TRs) and component size are used in the analysis. A quantitative analysis of TRs showed that reused components have significantly lower defect-density than non-reused ones. Reused components have, however, more severe defects than expected, but fewer defects after delivery. No significant relation between the number of defects and component size for all the components as a group or the reused ones was observed. On the other hand, the number of defects increases with component size for non-reused components. Therefore, other factors than size that may explain why certain components are more defect-prone, such as type of functionality, reuse, or type of faults for different programming languages. The results of the same study did not show any relation between defect-density and component size.

Reused components were less modified (more stable) between successive releases than non-reused ones, even if reused components must incorporate evolving requirements from two products. The study also revealed inconsistencies and weaknesses in the existing defect reporting system, by analyzing data that was hardly treated systematically before.

Discussion. Collecting data and some analysis was part of a master's thesis at NTNU in spring 2003 [Schwarz03]. The students inserted data for over 13 000 TRs in a SQL database. TRs were for several releases of SGSN and GGSN products, but data for four releases of a SGSN product was used in the statistical analysis, where data on the size of components were also available. However, this master's thesis did not separate the last two releases, since the third and fourth release were developed within one project and release three was merely a reconfiguration of the nodes. These releases were separated in the later analysis in [P8], and statistical analysis was repeated. Therefore, the numerical results in [Schwarz03] and [P8] differ a bit. Nevertheless, the conclusions are the same.

One important question is to discuss why reused components are less defect-prone but have more severe defects than non-reused ones. Several factors may be important and the significance of these factors should be further studied:

- Reused components are designed more thoroughly and are better tested, since defects in these components can affect two products. This is one of the advantages of design for reuse; i.e. aiming for higher quality.
- Erlang is the dominant programming language for reused components, while C is the dominant one for non-reused ones. Study of the type of defects in [Schwarz03] showed that Erlang units had 20% more faults per KLOC than C-

units and therefore the impact of programming language should be reverse[4]! However, software modules programmed in C showed to have more intra-component defects (defects within a module) than those programmed in Erlang. This can explain why the number of defects increases with component size for non-reused components.

- If a specific type of components dominates one group, this could be a confounding factor. Reused components do not have user interfaces, except for configuration and communication with the operator (these interfaces can be complex as well). On the other hand, reused components handle complex middleware functionality.
- Probably, defects for reused components are given a higher priority to correct. On the other hand, these components have fewer defects after delivery (which is important for reliability [Fenton00b]).

P10. An Empirical Study of Software Change: Origin, Acceptance Rate, and Functionality vs. Quality Attributes

In this paper, results of quantitative analysis of CRs in four releases of one system are reported. The results showed that earlier releases of the system are no longer evolved. Perfective changes to functionality and quality attributes are most common. Functionality is enhanced and improved in each release, while quality attributes are mostly improved and have fewer changes in the form of new requirements. The project organization initiates most CRs itself, rather than customers or changing environments. The releases showed an increasing tendency to accept CRs, which normally affects project plans. Changes related to functionality and quality attributes seem to have similar acceptance rates. While reused components could be expected to be more change-prone, no statistical significant difference between the change-proneness of reused and non-reused components was observed.

Discussion. In addition to discussing the results reported in this paper, some results are presented here that are published in [P10].

The IEEE Standard 1219 on software maintenance defines software maintenance as “The modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment” [IEEE1219]. This definition is not suitable for incremental development, where change is foreseen and delivery in increments is pre-planned (although the actual changes may only be partly pre-planned). Sommerville divides maintenance in three categories [Sommerville01]: fault repairing, adapting to new operative environment and adding or modifying the system’s functionality. He mentions that different people give these types of maintenance different names:

[4] We have also assessed the hypotheses using size in Equivalent LOC, with the same results. For calculating EKLOC, Erlang is multiplied by 3.2, Java with 2.4, and IDL with 2.35. Other studies have used other equivalent factors (for example 1.4 for Erlang to C). This study defined two hypotheses that Erlang and C modules include in average the same amount of functionality (equal means for size) and are equally defect-prone. Assessing these two hypotheses revealed a new equivalent factor for Erlang to C, being 2.3. This needs further verification.

- *Corrective maintenance* is universally used to refer to maintenance for fault repair.
- A*daptive maintenance* sometimes means adapting to new environment and sometimes means adapting to new requirements.
- *Perfective maintenance* is both used for perfecting the software by implementing new requirements, and for improving the system's structure and performance.

Fenton et al. add preventive maintenance to this list as well [Fenton97]:

- *Preventive maintenance* is combing the code to find faults before they become failures.

While maintenance is often used in connection with corrective maintenance, the term *evolution* is becoming more common to use for changing software for new requirements or adapting to new environments, and is better suited for evolutionary or incremental development. This paper uses maintenance to hold to known concepts, but corrective maintenance is not in the scope of this paper and was earlier studied in [P8]. Figure 6-2 shows the origin and type of changes in each release of the system.

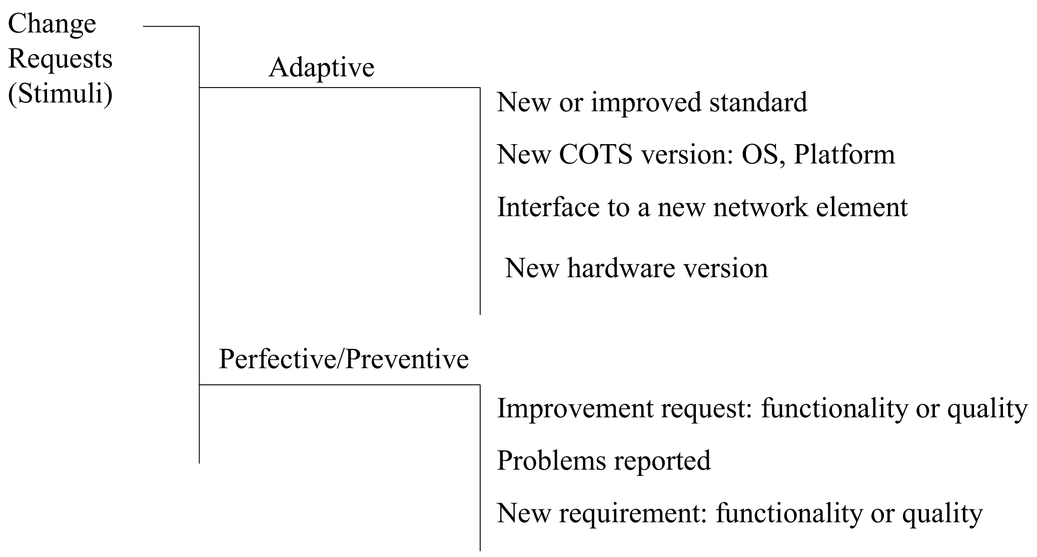

Change Requests (Stimuli)
Adaptive
New or improved standard
New COTS version: OS, Platform
Interface to a new network element
New hardware version
Perfective/Preventive
Improvement request: functionality or quality
Problems reported
New requirement: functionality or quality

Figure 6-2 Different types of CRs

Methods for assessment of maintainability are:

- Use of metrics such as number and impact of changes. Change requests per component can be an indication of the volatility of component design.
- Bosch recommends *change scenarios* which discuss changes that are most likely to happen and their impact on the architecture [Bosch00]. Qualitative assessment by ATAM uses scenarios and finding stimulus, responses and mechanisms to guarantee maintainability [Barbacci00].
- For COTS components, other techniques may be used such as fault injection or monitoring [Vigder99].

This study has used the first method by quantitative analysis of CRs. In the current practice of Ericsson, new requirements are either handled by:

a) The ARS for each release of a system.
b) The stream of CRs and that may add, delete or modify a requirement or an implementation.

Larsson et al. suggest that the number of requirements of a common component grows faster, but the paper does not give hard data for this claim [Larsson00]. Some other studies claim that most changes originate from external factors. Bennett et al. write that a request for change often originates from the users of the system [Bennett00]. From this point of view, since non-reused components have more application-specific functionality, they could be more change-prone. The results showed that most CRs stem from the project organization to improve functionality or quality attributes and the share of CRs related to quality attributes is higher. In other words, it could be expected that reused components are more change-prone. Quantitative results reported in [P8] and [P10] indicate the opposite conclusion: reused components are less modified between releases and the difference in #CRs/Component size is not significant[5]. The impact of CRs in LOC is not known. Therefore it is not possible to answer how much modification of code is due to CRs or other new requirements. The granularity of components is large in this study, since CRs give the impact at the subsystem level, which has impact on the statistical conclusion validity.

It is important to ask whether similar results could be verified with COTS components as well. Companies may think that COTS components change more often than internally developed ones, while changes in COTS component may be more visible and therefore be better remembered. The origin and type of changes in COTS components is not empirically studied in the literature.

Bennett et al. have proposed a model of incremental evolution of systems [Bennett00]. This paper compares this model with industrial data. One of the reviewers of the paper asked whether earlier releases were ever used, since they are not evolved any more. The earlier releases have in fact been installed and used. However, these are no longer evolved (only maintained for a period), since requirements are forwarded to the next release, as suggested by Bennett et al.

A study of the number of issued CRs over time and the date of requirement baseline in different releases showed that an unexpectedly high number of CRs were issued during a short time right after the requirement baseline. The question is whether the organization takes the costly decision to baseline requirements too early, while the product is still undergoing dramatic evolution. It also showed that in periods, the organization has to deal with several releases, while after a while all effort will be directed towards the new release and the old one enters the classic maintenance phase.

The impact of CRs on subsystems was also studied. Requirement changes may result in local changes in a component, several components or even the architecture. Only 104 of 169 CRs had data on the affected subsystems (i.e. high-level components). Table 6-2 shows the results. The majority of CRs affect only one subsystem. However, the granularity of subsystems is large and the impact within a subsystem is not known.

[5] Whether reused or non-reused components have less #CRs/Component size depended on which points are included in the analysis. The data has two outliers that affect the means.

Table 6-2 No. of subsystems affected per CR, of 104 CRs

No. of affected subsystems	One	Two	Three	Four	More than four
No. of CRs	57	31	5	4	7

There is a lack of empirical studies on software maintenance and evolution. Data that is used in literature on maintenance categories, distributions, source of changes etc. are either from studies performed many years ago, or built on surveys results. Bennett et al. mention some challenges meeting empirical studies on software maintenance to be [Bennett00]:

- Very small programs do not have maintenance problems and research must scale up to industrial applications for them to be useful.
- More empirical information about the nature of software maintenance, in terms of its effect on the software itself, on processes, on organizations and on people is needed. What actually happens from release to release? For example Cusumano and Selby reported that a feature set might change 30% during each iteration, as a result of the team learning process during iteration [Cusumano97].
- Recent technologies such as agents, components and GUIs need to be explored from a maintenance perspective.
- The conventional analysis of Lientz et al. on distribution of maintenance categories is no longer useful for modern software development (or at least should be verified for these approaches), since technologies have changed (see the paper for more details or [Lientz78]). It does not help reasoning about component-based systems, distributed systems etc.

The study has contributed to the state-of-the-art of evolution by suggesting new classifications of changes (functionality vs. quality attributes and different categories in each) and verifying an incremental model of software evolution.

P12. A Study of Effort Breakdown Profile in Incremental Large-Scale Software Development

Effort breakdown profiles are important to study and such profiles should be updated for major changes in development approaches or tools. Data from two latest releases shows that half the effort is spent before system test on specification, analysis, design, coding and unit testing. The other half is spent on system test (20-25%), project management (10-11%), adapting and maintaining processes for software development (2-5%) and CM (12-13%).

Discussion. Systematic use of CM has a crucial role in CBD and incremental development. Functionality is delivered in chunks that must be integrated and maintained. Increasing effort needed for CM and integration is predicted in literature for CBD. Probably, the effort for CM and testing increases with incremental development of large systems, which is a hypothesis that should be further verified by other studies. Estimation methods that assume most of the effort is spent on analysis and design may therefore need revision. This is also the first study that shows the cost of adapting and maintaining RUP in a large industrial project.

Combining Results

The goal with data exploration is to increase the understanding of a phenomenon, to generate hypotheses or theory, or to verify some known theories. A model based on the results of the quantitative studies and qualitative observations (this thesis and others) is developed here. It shows the impact of development approaches on practices and in turn on dependent quality metrics. Other studies on the impact of development approaches on quality attributes are reported in [Nuefelder00], [Zowghi02] and [MacCormack03].

Table 6-3 shows a summary of data already given in the papers in order to facilitate this discussion. Although the data are not enough to perform statistical analysis, they are still useful in developing a model that should be further verified. Note that release 1 has a low number of CRs and TRs, since CR and TR handling processes have matured over time. For instance, some changes of release 1 were handled informally.

Table 6-3 Data from internal measures and the studies in this thesis

Quality Metrics	Release 1	Release 2	Release 3	Release 4
Requirement Stability	92%	75%	91%	69%
Number of CRs	10	37	4	118
Acceptance rate of CRs	40%	51%	75%	62%
Number of TRs	6	602	61	1953
Planning precision	91%	95%	91%	78%

Figure 6-3 and Table 6-4 summarize the observations. Development approaches are independent variables that lead to practices as described in Table 6-4.

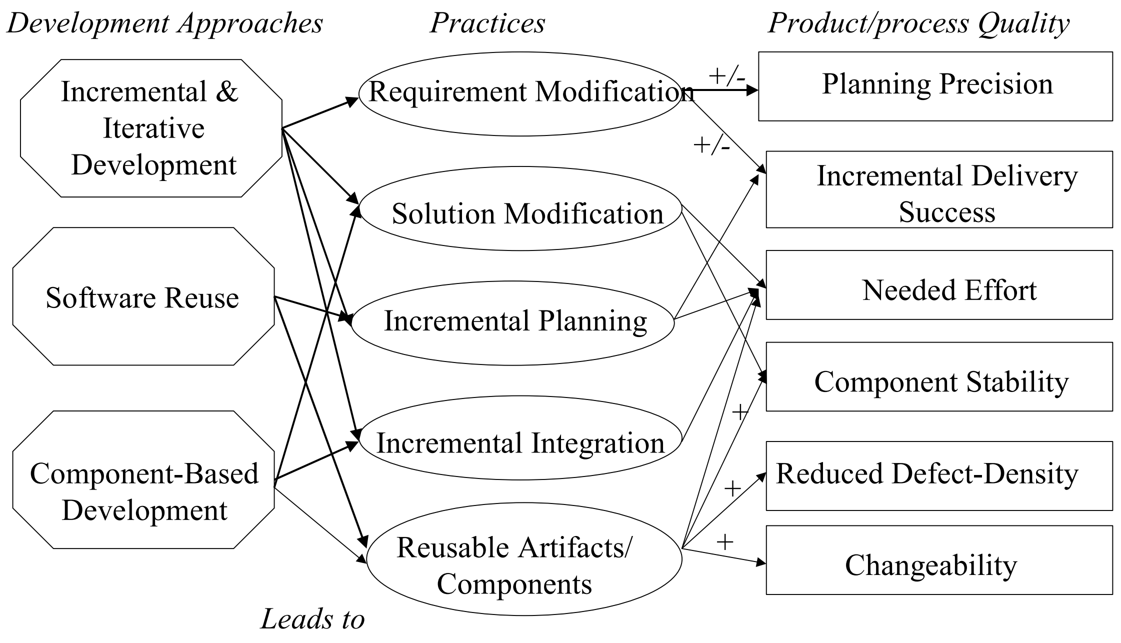


Figure 6-3 The impact of development approaches and practices on quality metrics

The software process in the company is an adaptation of RUP. When incremental or iterative development is mentioned, this specific process is meant. “Software reuse and product family development” and “CBD” are shown separately, since a reusable artifact can be any type of artifact, including software processes, a component or a component framework.

Table 6-5 describes the impact on product and process quality metrics. The last column also shows whether qualitative or quantitative data can verify the impact.

Table 6-4 The relations of development approaches to practices

Development approach	**Development practice**	**Description**
Incremental and iterative development	Requirement modification	Project scope is discovered and established gradually and the project is open to change. Incremental development may therefore lead to increased requirement modification.
	Solution modification	Implemented solutions are modified; either to improve and enhance them or to realize new requirements.
	Incremental planning	Requirements are assigned to increments. It is important to define iterations of suitable duration and right functionality, and to solve dependencies between requirements.
	Incremental integration	Solutions must be integrated in each iteration and release, according to an integration plan, and previous releases may need updates.
Software reuse and product family development	Incremental planning	Development *for* reuse: some reusable artifacts should be developed first, e.g. the component framework. Development *with* reuse: reuse must be planned, especially for COTS components or here the WPP. A release may depend on reusable artifacts from another project.
	Reusable artifacts	Reusable artifacts (including components) should be developed and certified.
CBD	Solution modification	Components are modified in several releases and iterations, unless components are defined in a way that new requirements are assigned to new components or new interfaces (the disadvantage is perhaps poor structure due to too fine granularity).
	Incremental integration	New versions of components should be integrated into each release.

Table 6-5 The impact of practices on the product and process quality metrics

Development practice	Quality metrics	Description
Requirement modification	Planning precision	Modifications in requirements (measured in requirement stability) affect planning precision. This impact can be positive (some requirements may be removed to deliver on time) or negative (new requirements need more effort). Quantitative data in Table 6-3 shows reduced planning precision with reduced requirement stability. Two reasons are identified: 1) the acceptance rate of CRs has increased; 2) only 5% of CRs ask to remove a requirement [P10].
	Increment-al delivery success	Only planning precision reflects this at the moment, but success of incremental delivery includes delivering on time, delivery of increments of right size, and with right and verified functionality. Requirement modification changes the original delivery plan. The effect can be positive if the original plans were too optimistic, or negative if requirement modifications reduce the product quality.
Solution modification	Needed effort	Artifacts should be re-opened and understood before modification. These artifacts should also be quality-assured by inspections, reviews etc. Increased effort is therefore suggested. The observed low inspection coverage can be due to incremental modification of solutions [P5].
	Component stability	When components are iteratively modified, stability between releases is reduced.
Incremental planning	Increment-al delivery success	Qualitative feedbacks indicate that it is difficult to map requirements into increments of right size and many non-functional requirements could not be tested early, leading to “big bang” testing [P5] [P7]. An “integration plan” was therefore developed.
Incremental integration	Needed effort	Incremental integration will need more effort for CM and regression testing [P12].
Reusable components	Needed effort	Extra effort for developing *for* reuse will pay off in total reduced effort and cost. There is no data to assess ROI for reuse. Less defects after delivery reduces maintenance cost [P8].
	Component stability	Reused components are more stable [P8].
	Reduced defect-density	Reused components are less defect-prone [P8].
	Change-ability	Most changes impact one or two subsystems, but the granularity of subsystems is large in the study [P10].

While software reuse has had a positive impact on changeability and component quality (in terms of reduced defect-density), it has made incremental delivery success more difficult. Incremental development has had a negative impact on project metrics reflected in decreasing requirement stability, decreasing planning precision, and increased integration and testing effort. Benediktsson et al. suggest reduction in effort with incremental development and high number of iterations when the diseconomy of scale is large, but their model is a theoretical one that needs empirical assessment [Benediktsson03].

The positive impact of incremental development in reducing risks is not measured, although some requirements that were originally planned were later removed. Increased effort is not surprising as it would be cheaper to develop a system in a waterfall model, if all the requirements were known in the beginning. Other disadvantages may be reduced by for example combining design item responsibility and increment responsibility, or integration-driven delivery as Ericsson has chosen. One reason for the negative impact may be in being unprepared for the challenges, such as too early requirement baseline.

6.3 Improving the Practice - RQ3

Three papers are presented in this section: [P4], [P11] and [P13]. These papers, together with proposals for adapting RUP for reuse, are related to **RQ3**.

P4. Object-Oriented Reading Techniques for Inspection of UML Models - An Industrial Experiment

This paper describes an experiment to evaluate the cost-efficiency of tailored Object-Oriented Reading Techniques (OORTs) in a large-scale software project. The OORTs were developed at the University of Maryland. The techniques have earlier been tested on small projects where UML models are developed from scratch. This is the first controlled experiment in industry on their applicability and with incremental development. The results showed that the OORTs fit well into an incremental development process and managed to detect defects not found by the existing reading techniques. The study demonstrated the need for further development and empirical assessment of these techniques and for better integration with industrial work practice. As part of the study, data from several earlier inspections in Ericsson were collected and analyzed to have a baseline for comparing.

Discussion. Two teams of a total of four students have been involved in tailoring the techniques, collecting historical data and performing the experiment in their master's theses at NTNU and HiA [Arif02][Bunde02]. The study demonstrated that the inspection techniques should be adapted for large system development and the context. Here, UCSs describe steps in use cases, while UML models only show actors and relations between use cases. Of the seven original OORTs, OORT-4 (Class Diagram vs. Class Description Document) changed focus to Class Diagram for internal consistency and OORT-5 (Class Description vs. Requirement Description) was removed, since it was not applicable in Ericsson. The study revealed inconsistencies between models, as also described in [P3].

P11. Exploring Industrial Data Repositories: Where Software Development Approaches Meet

The paper presents a method for mining industrial data repositories in empirical research, using studies described in [P8], [P10] and [P12]. The challenges of integration are classified in two categories:

- The *physical challenge* refers to the integration of data. It may be handled by inserting all data in a common database, defining metadata or defining tools that collect and analyze different sources of data, such as in the French railway company SNCF [Beaurepaire04].
- The *conceptual challenge* refers to integrating the results of separate studies with one another and integrating the results into theories. In empirical studies at Ericsson, it is observed when development approaches are combined, while metrics and measurement programs are not. To develop advanced theories on the relations between development approaches and their impact on quality attributes, measurement programs should be updated to collect some basic data for a combination of development approaches. Metrics for a combination of incremental development, reuse and CBD are identified.

Discussion. For component-based systems developed in object-oriented languages, metrics defined in various object-oriented literature are applicable, e.g. [Fenton97] [Briand02] [Alshayeb03]. With modeling in UML, metrics defined for UML models are also useful, e.g. [Lanza02] [Kim02]. Paulin outlines some metrics for component-based systems (and any type of system) as [Heineman01-Chapter 23]:

- Schedule: actual vs. planned.
- Productivity: total development hours for the project/total number of LOC.
- Quality: total number of defects and severity.
- Product stability: number of open and implemented change requests that affect the requirement baseline.
- Reuse%: Reused LOC/Total LOC.
- Cost per LOC.

For components, Paulin adds:

- LOC per component. For COTS components or generally when the source code is not available, LOC should be replaced with other metrics such as physical size in Kbytes.
- Labor: effort expended per component.
- Classification of the component: new code, changed code, built for reuse or reused code.
- Change requests per component as indication of the volatility of component design.
- Defects per component as a measure of the reliability of the component.
- Cost per component.

Sedigh-Ali et al. also propose use cases per component, but this may be difficult because of scattering and tangling effects [Sedigh-Ali01a].

The studies in Ericsson revealed inconsistencies in the data collection system (e.g. in granularity of data) and lack of some basic metrics that could be useful in assessing development approaches. For example, effort spent on each component, requirement, or modified solution is not recorded. Therefore, it is not possible to answer (quantitatively) whether reuse is cost-beneficial, whether requirements were correctly assigned to iterations regarding needed effort (schedule overruns may be because of poor estimation or unrealistic planning of an iteration), or what is the impact of changes on lower level components. Metrics for component-based systems that are developed incrementally are proposed. Data should be collected automatically as far as possible and be stored in a common database with query possibilities.

P13. Use Case Points for Effort Estimation - Adaptation for Incremental Large-Scale Development and Reuse Using Historical Data

The Use Case Point (UCP) estimation method is earlier used for estimating effort in small systems, with a waterfall model of development. The paper describes calibrating the method for Ericsson using historical data, with incremental changes in UCSs and with reuse of software from a previous release (using the COCOMO 2.0 reuse formula). Data on effort spent in one release are used to calibrate the method and the method is verified using data from the successive release.

Discussion. Effort Estimation is a challenge every software project faces. Ericsson has used an inside-out estimation method performed by experts. Studies show that expert estimations tend to be too optimistic and large projects are usually under-estimated. The UCP estimation method may be used in addition to expert estimates to improve the accuracy of estimates. There is no standard for writing use cases and UCSs in this study were much more complex than previous studies using the UCP estimation method. These complex, incrementally developed UCSs were broken into smaller ones. There are 13 technical factors (e.g. distributed system, reusable code and security) and eight environmental factors (e.g. object-oriented experience and stable requirements) in the original method. The technical factors have little impact on the estimation results and some earlier studies have proposed to drop these. On the other hand, the environmental factors can have large impact on the results. The projects were assumed as average, setting the total weight of these factors to 1 to drop assigning values to these factors that are highly subjective. Furthermore, effort to implement a use case point (PH/UCP) varies in different studies and in this study, the maximum value used in previous studies (36 PH/UCP) only counted for the effort needed before system test for this system. This was explained by comparing effort breakdown profiles of these studies. To account for reuse of software, a COCOMO 2.0 formula for reuse is applied by calculating an Adaptation Factor (AF) equal to 0.55 (effort needed to reuse software comparing to developing it from scratch).

Results of the study showed that the UCP estimation method could be calibrated for a given context and produce relative accurate estimates. There are two factors that need further study and possible adjustments: the PH/UCP and the AF.

6.4 Summary

The results and their relation to research questions were presented in Sections 6.1-6.3. The studies cover several aspects of software development due to the emerging research design, the type of available data, and the fact that there is a combination of development approaches in the real context that should be studied as a whole. The attempt has been made to use all the available data, but not to "overuse" them, and be aware of the limitations of the results as discussed later in Chapter 7. Introducing product family engineering and incremental development have benefits that are either verified here or in other studies. There are also challenges in adaptation of software processes that should be answered. For large-scale development, it is important to verify that a method (such as the UCP estimation method) scales up.

7 *Evaluation and Discussion*

This chapter answers the three research questions RQ1-RQ3 based on the results. Further, the relations of contributions to the research questions, context, papers and INCO goals are discussed. There is also a discussion of validity threats and the experience from working in the field and how ethical issues are handled.

7.1 Research Questions Revisited

Answers to the three research questions are:

RQ1. *Why a reuse program is initiated and how is it implemented?* The question is answered as:

a) A product family is initiated because of the similarity between requirements of the emerging system (SGSN-W) and an existing system (SGSN-G), and because of the possibility to reuse an internally developed platform and components. Having a common base for the two products makes it possible to adapt the product for different markets, with either GSM or W-CDMA (for UMTS networks) or both. Shorter time-to-market and reduced cost are suggested, but are not assessed in this thesis.
b) A lightweight or extractive approach to product family adoption was chosen. Software architecture is evolved, a component framework is developed, and a common software process and environment is defined. Management support, common goals and common infrastructure, and experienced personnel have been critical for the success of reuse. Other studies have also emphasized the importance of management support and common wisdom in the success of reuse programs [Morisio02] [Griss95].
c) The software process model (GSN RUP) is not adapted for design for and with reuse: workflows are described as if there is a single product development, there is no explicit framework engineering activity, and reuse is not included in aspects such as the estimation method and metrics. Guidelines for modeling and design are linked to the related activities in GSN RUP, but these should also be improved regarding developing for and with reuse. Experienced staff and domain knowledge compensate for these shortcomings to some degree.

RQ2. *What is the impact of software reuse, CBD and incremental development on the quality?* Here, the impact of development approaches on product quality metrics and on project attributes such as schedule or effort are sought. The answers are:

a) Reuse benefits have been observed in form of lower defect-density and higher stability of reused components between releases by analyzing TRs and the size of modified code. A study of CRs also showed no significant difference in change-proneness between reuse and non-reused components. The studies described in [P8] and [P10] are the first empirical studies on the quality impact of reuse in a large-scale industrial system and are summarized in C1. The confounding factors that affect validity of the results are discussed, but the conclusion is that reuse has a positive impact on quality, since reused components will be better designed, better tested and changed with more care.
b) The analysis of CRs suggests that functionality is both enhanced and improved, while quality attributes are mostly improved in each release of the system. Most CRs were related to quality attributes in terms of modified requirements. In other words, incremental development leads to incremental perfection of these attributes. A model of evolution as proposed by Bennett et al. was observed [Bennett00]; i.e. earlier releases of the system are only maintained for a period and would no longer be evolved [P10]. The project organization initiates most change requests itself, rather than customers or changing environments, contrary to what was expected and proposed by others (see [P10]). These observations are summarized in C2.
c) Incremental development of large systems needs probably more integration and testing effort, based on quantitative analysis of effort spent in two releases and qualitative observations as explained in Table 6-5. Other studies have proposed considering diseconomy of scale for large systems [Symons91] [COCOMO 2.0 in Boehm95]. Benediktsson et al. suggest compensating the diseconomy of scale with a sufficient increase in the number of iterations [Benediktsson03]. Their analysis did not consider other factors than scale. The effort needed for integration and regression testing should be added as a factor when discussing incremental development. The AF (Adaptation Factor) that is used to estimate effort for reuse of software developed from a previous release contra developing for the first time (the formula is borrowed from COCOMO 2.0) includes the effort needed for modifications and integration, and was set to 0.55 [P13]. Even higher AF factors may be proposed. These observations are part of C3.
d) Qualitative observations and quantitative data are combined to propose a model of the impact of development approaches on some quality metrics. Some impacts are verified by data, while others are not quantitatively assessed and need metrics as explained in [P11]. A relation between increased requirement volatility, increasing acceptance rate of CRs and reduced planning precision was observed. Others have also proposed a relation between requirement volatility and both schedule and cost performance, as described in [P10]. The factor of increasing acceptance rate of changes is propose by this thesis, since adopting incremental development makes the project more open to change, even for requirements specified for a single iteration.

RQ3. *How to combine the qualitative and quantitative results to improve the practice in some aspects?* Five contributions come from this work:

a) An effort estimation method is developed using complex UCSs, with incremental changes in these and reuse of software from previous releases. This estimation method, which is an adaptation of the UCP method, is a top-down method that can be used in parallel with expert estimation to improve the estimation results or used stand-alone. UCSs and effort spent in two releases are used to calibrate and evaluate the method. A high PH/UCP was used compared to earlier studies, which may be explained by the complexity of the project and the fact that the environmental factors were set as an average project. The technical factors were also set as an average project, since such factors are highly subjective and difficult to compare with other studies. The study was the first one on a large project and included a reuse factor for incremental development, and is summarized in C3.
b) Metrics for a combination of development approaches are identified that can help the conceptual integration of results [P5] [P11]. Although metrics are extensively discussed in the literature and various metrics for CBD are proposed, the combination with incremental development is not discussed before. These metrics could be useful in assessing development approaches and their impact, and are summarized in C4.
c) The experience from mining data repositories is analyzed to develop a research method for future studies, as summarized in C5. This method combines literature study with bottom-up pre-study of data to generate theories or hypotheses. Steps in a data mining process as proposed in [Cooper01] will cover the execution phase. Three cases of using such a method are reported in [P11]. Data used in these studies needed preparation and insertion in a database with query possibilities (here SQL), since data was mainly stored as plain text. Two types of challenges in collecting data and interpreting the results are observed: the physical integration challenge and the conceptual one. The physical challenge has been subject of earlier studies. Inserting all data in a common database, defining metadata or using tools that collect and analyze different sources of data may handle it. On the other hand, the conceptual challenge is less discussed in other studies and this study has proposed defining metrics for a combination of development approaches as discussed above.
d) There is agreement in other studies that having a software process for reuse has a positive impact on reuse (see Section 2.4 and [Rine98]). This work proposes to adapt GSN RUP for reuse by adding some activities in the existing process model and modifying some others. These proposals are summarized in C6a.
e) Results of an experiment comparing inspection techniques may be used to improve techniques for inspection of UML models [P4]. The new adapted OORTs were more successful in detecting inconsistencies between models, while the existing inspection technique in Ericsson detected more faults in the design of use case realizations. More defects were detected in the individual reading phase than in the meeting by using the OORTs; i.e. having more structured techniques such as the OORTs will improve the individual reading phase. On the other hand, these techniques should be adapted more for the

context and be combined with internal guidelines to increase their effectiveness in detecting defects of all types. The results are summarized in C6b.

The above contributions were described in Section 1.6. Table 7-1 shows the relations of contributions to the research questions and the papers. Two papers are not directly connected to the contributions: [P3] and [P9]. [P3] presents the development process in the context of MDA and investigates re-engineering of the system as a legacy system in this context. It is a contribution to the state-of-the-art of MDA and no other studies on legacy systems in this context were found. Results of [P9] were used in the generation of hypotheses for a future survey on the state-of-the-practice of CBD.

Table 7-1 The relations of Contributions (C) to Research Questions (RQ) and papers (P)

C	**Research questions**			**Papers**												
	RQ1	**RQ2**	**RQ3**	**P1**	**P2**	**P3**	**P4**	**P5**	**P6**	**P7**	**P8**	**P9**	**P10**	**P11**	**P12**	**P13**
C1		√									√		√			
C2		√											√			
C3			√												√	√
C4			√					√						√		
C5			√											√		
C6a	√		√	√	√	(√)			√	√		(√)				
C6b			√				√									

7.2 Contributions, Development Approaches and the Context

As discussed in Section 1.6, development approaches should be considered in combination with one another. Section 5.3 discussed the impact of context in two dimensions: scale (large-scale development) and the telecom domain. This section discusses the relations of contributions to development approaches and the context. In some cases, the results are influenced by the characteristics of the GPRS system as a telecom system, which affects generalizability to other domains.

Table 7-2 The relations of Contributions (C) to development approaches and the context

C	Reuse	CBD	Incremental development	Large system development	GPRS (Telecom)
C1. Reuse benefits			Data from several releases is analyzed.	First industrial large-scale study.	More reuse in telecom systems (?).
C2. Software evolution		The granularity of subsystems is large.	Increasing acceptance rate. Most CRs are initiated internally to improve quality. Earlier releases are no longer evolved.	Large systems are long-lived and are improved incrementally.	Different quality require-ments are important in different types of systems.
C3. Effort estimation method	Reuse of software from a previous release.		Adapting for incremental changes in UCSs. Large CM and testing effort.	Complex UCSs.	Large testing effort.
C4. Metrics	Adapted for reuse.	Adapted for CBD.	Adapted for incremental development.	Different granularity of components.	Traditional Ericsson decomposi tion.
C5. Data mining method				Conceptual and physical integration challenges.	Data is stored in several repositories.
C6a. Software process model	Adapting RUP for reuse.	A reusable internally developed component framework acts as a platform.	Inconsistency between UML models, and between UML models and requirements or code is a problem using RUP.	Describing an industrial case of product family development and adaptation of RUP.	GSN RUP adaptation.
C6b. Inspection tech-niques			Adapting for incremental changes in UCSs.	Complex UML models and UCSs.	

A few notes on the contents of Table 7-2:

- C2 and C3 are exploratory studies that generate hypotheses for future assessment.
- For C5, storing data in a common database or defining relations between metrics are solutions that are proposed in other studies. This thesis contributes to identify the conceptual challenge in the integration of the results due to metrics that are defined for single approaches, and not a combination of them.
- Although reuse adaptation proposals in C6b are for the Ericsson adaptation of RUP (GSN RUP), they may be reusable in other contexts. These proposals follow the notations of RUP and do not contain any product-specific details.

7.3 Relations to INCO Goals

The relations between the results and the INCO goals as defined in Sections 1.2 and 5.2 are now considered:

G1. Advancing the state-of-the-art of software engineering. It is thought that the work reported in this thesis advances the state-of-the-art of software engineering as defined by its contributions. Better understanding of approaches to product family engineering, software reuse and incremental development is achieved, as reflected in the contributions C1, C2 and C4.

G2. Advancing the state-of-the-practice in software-intensive industry and for own students. Some feedback is given to Ericsson, but improvement activities stopped due to organizational changes. However, C3 to C6 are reusable in other contexts. Several students have participated in the studies.

G4. Disseminating and exchanging the knowledge gained. Most results are published and presented at international and national conferences or workshops. During this thesis work, five groups of master's students of a total 11 students from NTNU and HiA have performed their project works and master's theses at Ericsson, which is an example of university-industry co-operation. I have held presentations in courses at NTNU and am a co-lecturer at HiA, where I teach software processes, CBD and empirical studies. I have used empirical studies in this thesis as examples in these courses. Furthermore, INCO plans to participate in an international seminar in Oslo on SPI on 7-8 September 2004, where the results of this thesis will be presented.

7.4 Evaluation of Validity Threats

Four groups of validity threats in empirical research are considered in Section 4.3 and validity threats of individual studies are discussed in the papers. In Section 4.1, two possible remedies to improve the validity of studies are mentioned:

1. Replication over time and in multiple contexts. This work has assessed some earlier theories in new contexts:
   - The relation between defect-density or the number of defects and component size was earlier studied, but not in the context of reuse [P8].

  - The origin and type of changes or maintenance activities were earlier studied, but not in the context of incremental development of a large-scale system [P10].
2. Combination of data and research strategies. The choice of research strategy (a mixed-method design) increases the validity of results. In the interpretation of the results, quantitative data are combined with each other and with qualitative observations. Examples are:
   - Study of the impact of reuse both on defect-density [P8], modification degree [P8] and change-proneness [P10].
   - Combining results of different studies in Section 6.2- Combining Results.

Combining data and research strategies needs a good grasp of the context, access to multiple sources of information, and a time frame that allows collecting and interpreting different types of data. However, it is powerful in the sense that it combines all the evidence. Some common threats to the validity of results are further discussed here.

Quantitative studies. The collected data is considered to be reliable. It is gathered from the company's data repositories, in controlled experiments, or from the company's internal measures. Some threats to validity of quantitative studies and how these are handled are as follows:

- *Conclusion validity.* Some analysis could only be done on the subsystem level, which gives too few data points for statistical analysis in [P8] and [P10]. A second threat is due to the missing physical integration of data repositories (cf. [P11]). For example, TRs report defects identified in system test and in later phases, and not during inspections and unit testing. However, reliability is often considered to be related to defects detected in later phases, especially after delivery (which are costly to repair and have impact on the users' perception of quality). In the study of CRs, these were stored in several web pages and in different formats [P10]. This threat is handled by inserting all available data in a common SQL database.
- *Internal validity.* Missing data is the greatest threat to the internal validity of the studies on TRs and CRs [P8][P10]. This is due to the processes of reporting troubles and changes, since these reports miss data about the faulty components or affected components if this data were not known at the time of initiating a TR or CR. Missing data of this type do not introduce systematic bias to the results. The reasons for missing data were sought and the distributions were analyzed when possible (e.g. [P8]). Missing data is not substituted and the statistical tests are robust when the data size is large enough. The data is complete in the studies on effort and the estimation method [P12] [P13], the inspection experiment [P4] and the survey on developers' attitude to reuse [P2] [P9].
- *Construct validity.* The construct validity of the questionnaire used in the survey on developers' attitude to reuse is not addressed, since this was a pre-study and had a small scale [P2]. It should be verified whether the quality metrics used in quantitative studies (such as defect-density, stability and change-proneness) are software quality indicators. These metrics are mostly taken from the literature. The identified metrics for a combination of development approaches presented in [P11] should be verified for construct validity, which may be the subject of future studies.

- *External validity.* In Section 5.3**,** challenges in developing large systems were discussed and some characteristics of the GPRS system that may impact external validity were presented. The external validity of the contributions is discussed later in this section.

Qualitative studies. Again, the collected data is reliable, using internal reports, feedbacks and own experience, but this data is subjective and can be subject to other interpretations as well. Prior knowledge of the system and the organization, and valuable feedback from colleagues improve the validity. Rival explanations are addressed when interpreting the results and the conclusions are combined with quantitative results as far as possible.

Case study research. Section 4.2 discussed rationales for performing a case study, including the case being a critical, representative or revelatory one. The rationales for justifying this research are:

- Affiliation in the company provided the chance to access and analyze data that is rarely accessible to empirical studies (revelatory case). In some cases, the studies are the first ones on an industrial large-scale system [P4] [P8] [P10] [P12] [P13].
- The system in the study can represent a critical case for verifying theories. Assessing the impact of reuse in a company with an extractive approach to reuse (not pre-planned from the beginning), across multiple organizations and with high risks of large-scale development can strengthen the theory on reuse benefits in other cases (related to C1). The same is true for C3 and C6a: adapting the UCP estimation for complex use cases and a large system and the experiment on OORTs in incremental development with complex UML models verifies that the methods are adaptable to the context and do scale up.
- In some cases, the study is “an example” of industrial practice, such as the software process model and the practice, being in different maturity levels regarding reuse [P2] [P7].

External validity of contributions. This work has performed confirmatory, descriptive and exploratory studies, with different degrees of generalizability:

- C1. Results in C1 confirm existing theories and are therefore easier to be reused in other contexts [P8].
- C2. Section 4.2 discusses that a case study may show to be a falsifying case, in which case the results are more interesting for the research community. Some pre-assumptions were revised in this thesis, especially in [P10]. For example, most changes stem from the project organization and not from external actors as assumed in other studies; i.e. a falsifying case. Hypotheses regarding the share of changes in software evolution are grounded in the available data and may at least be generalized to similar systems in the company or in the same domain. The system in study has high focus on some quality requirements. The share of these requirements may be different in other domains. However, probably other systems such as web-based ones have other quality requirements with the same importance. Generalization to other domains needs further study.
- C3. The adaptations in the effort estimation method (breaking down complex use cases, assuming average project and using an adaptation factor for software reuse) are reusable in other large systems with incremental development [P13].

The distribution of effort over development phases is grounded in the data and may be valid for large systems, but not for smaller ones that do not face the same challenges for CM, integration and testing [P12].

- C4. The identified metrics is reusable in other systems with the combination of reuse and incremental development [P11]. As described in Section 5.3, development methods are increasingly becoming common in software projects across companies.
- C5. The data mining method is reusable in the analysis of other systems [P11].
- C6a. RUP must be adapted for the context and so are the adaptation proposals for reuse. However, the proposals are generic and may be reused in other adaptations of RUP.
- C6b. Results of the experiment on inspection techniques may be of interest in future improvement of the OORTs [P4].

7.5 Working in the Field

Two aspects are discussed here: ethical issues and being exposed to organizational changes during a thesis work.

Being an employee of the company during this research has had several advantages. I had first-hand knowledge about the routines for collecting data, it was “easy” to access data (although in practice most of the data in quantitative studies was collected and analyzed by mining several data repositories), and knowing the colleagues helped in different stages of data collection, performing the survey and the experiment. Nevertheless, in any study in the field, there are ethical issues that should be considered.

This work has followed a common principle: the company and the participants are informed on the goal of each study and permission was gained to collect the data. Another concern has been to avoid interrupting on-going work. Sometimes, a study was delayed several times for the right moment to perform or even was cancelled. An example is the experiment on inspection techniques in [P4] that was delayed to fit the inspection plan. There are also specific issues for each study:

- Several students have been involved in collecting and analyzing data. They have all signed confidentiality statements according to the company’s rules.
- In publishing some of the results, the data is aggregated and presented by means or medians to avoid too detailed information. Data that may be considered as confidential are not published.
- Key personnel were asked to comment the results or read the draft of a paper.

As discussed by Singer et al., empirical research in software engineering needs some rules regarding ethical issues [Singer02]. For example, should we report problematic processes in a company? This work discusses problems in the measurement program and processes for collecting data, e.g. reporting defects or effort. These problems are not specific to this company and the literature reveals that most companies face similar challenges. Lots of data were collected that were not properly analyzed; either no metrics are defined or metrics are not connected to quality goals, or there is a lack of resources to perform analyses. The overall feedback from conferences and workshops has been positive, admiring the company’s willingness to allow empirical studies of on-going projects.

During this work, Ericsson decided to centralize all development of the product in study in a few centers and gradually closed down the GPRS development project in Grimstad. As described before, this affected the course of this work. Nevertheless, the research was re-designed and still performed in an industrial context, but with different focus. This experience confirms that working in the field needs flexibility and incremental, emerging research design.

8 Conclusions and Directions for Future Work

This thesis has presented the results of several empirical studies performed at Ericsson, which is one of the world's leading suppliers of mobile (and IT) systems. These studies were sometimes the first performed in a real context of a large and long-lived system. The studies combine literature study, collecting and analyzing quantitative data from data repositories and qualitative data from different sources, experiments, statistical hypotheses testing and case studies. A mixed-method research design was applied to allow taking benefit of all available data, combining the results and answering questions that are not possible to answer otherwise. The top-down confirmatory approach is combined with the bottom-up explorative and descriptive approaches.

This work mainly analyzed data that the company itself had not analyzed at all or not to the extent presented in this thesis. Prior knowledge of the product in the study and the organization, and combining different types of data with one another and with previous work improve the validity of the results.

Empirical research is performed to verify theories, develop new theories or extend existing ones, and improve the practice. The thesis contributes to these aspects by:

1. **Describing different aspects of software development**; i.e. ***the power of example***:
   - 1.a) *The practice of software development in a large-scale product family has been described.* The product family consists of two products that is initiated using a lightweight and extractive approach. A component framework is developed that embraces many quality attributes and acts as a platform for application developers. Management support, common goals and common infrastructure, domain knowledge and experienced people are success factors to reuse. Components are developed in-house or by other Ericsson organizations. Although internally developed guidelines and design rules are added to the software process model (an adaptation of RUP), software reuse and product family approach is still not explicit in the internal process model: workflows are explained as if there is a single product development and there is no framework engineering activity.
   - 1.b) *This work identified aspects with improvement potential for the company and proposed some improvements*: a) reuse activities were proposed that could be added to the software process model, b) results of an experiment

comparing the company's inspections techniques with the adapted OORTs showed that the new techniques detected inconsistency defects that were not detected by the existing technique, c) an effort estimation method for top-down estimation using UCSs is developed that can be used in addition to expert estimates to improve the accuracy of early estimates, and d) the quality of metrics, the measurement program, and the processes for reporting defects, changes and effort were evaluated. It is suggested to collect data automatically as far as possible and in a common database. It is also suggested to analyze this data and use the results in evaluating product and project goals, and development approaches.

2. **Verifying existing theories and assessing existing methods in new contexts**; i.e. ***the power of replication***:
   2.a) *Reuse benefits were quantitatively verified for a large system and with components developed in-house.* Reused components had actually lower defect-density and were less modified between releases than non-reused ones. No difference was observed in the change-proneness of reused and non-reused components. Other studies claimed that reused components change more, since these should meet requirements of several products. Although some confounding factors were identified, the conclusion is that reused components are designed and verified better, and are changed with more care.
   2.b) *This work evaluated the UCP estimation method and adapted in the context of incremental development of a large system.* The method was earlier tested only on small projects and with use cases developed from scratch. The results of this study verified that the method scales up with certain modifications, and works well without the technical and environmental factors. For incremental development, steps were added to count modifications in UCSs. Furthermore, the reuse of software from a previous release is accounted for by adding an adaptation factor borrowed from COCOMO 2.0.
   2.c) *This work evaluated the OORTs and adapted it in the context of incremental development of a large system.* Results of the first controlled industrial experiment on the techniques showed that the OORTs and the existing inspection technique detected different types of defects but had almost the same cost-efficiency.
   2.d) *This work evaluated RUP in the context of product family development and proposed adaptations for reuse.* Although RUP is the most widespread software development process, no other studies on the reuse aspect have been found. Adaptation for reuse may be done by adding activities to existing workflows, such as "Make vs. Reuse vs. Buy decision" and "Record reuse experience". Separating framework engineering and application engineering may be the subject of future studies.
3. **Generating new theories, hypotheses or methods** by analyzing data from new perspectives (as in grounded theory) or combining the results of several studies; i.e. ***the power of generalization***:
   3.a) *The origin of Change Requests and the distribution over functionality vs. quality attributes was studied.* The results showed that functionality is both

enhanced and improved between releases, while quality attributes are mostly improved. Most change requests were related to quality attributes and were issued by the project itself. Although the share of maintenance activities (corrective, perfective, adaptive or preventive) was earlier studied, this was the first empirical study on functionality vs. quality attributes and in the context of incremental development. The study covers activities related to software evolution and corrective activities are not included in this study.

3.b) *The distribution of effort over development phases for incremental development of a large system was studied.* The results showed that only half the effort is spent on development before system testing. It also empirically showed the share of CM (12-13%), system test (20-25%), and adapting and maintaining the software process model (2-5%). Incremental development needs incremental integration of software and regression testing, as reflected in the effort needed for CM and system testing.

3.c) *The results of quantitative and qualitative studies have been combined in a model of the impact of development approaches on quality metrics.* Development approaches should be studied in combination with one another, since approaches have multiple and crosscutting impacts on development practices and quality metrics. For example, software reuse improves product quality, but creates challenges in incremental planning and incremental delivery. Incremental and iterative development leads to requirements modification, which may reduce planning precision by adding more requirements or improve it by removing requirements. Software reuse and CBD are proposed to reduce effort, but some extra effort is needed for incremental integration, incremental changes in artifacts that are already developed and verification.

3.d) *This work has identified metrics for a combination of development approaches*. Metrics for reuse and CBD are defined in various parts of the literature. However, these metrics should be combined with one another, with metrics for incremental development and with industrial practice. It is importance to define a proper granularity of components in metrics, collect data at the end of each release, label components as reused or new, and collect effort per component and requirement.

3.e) *A data mining method for exploring industrial data repositories has been developed* based on the experience from quantitative analyses. The method combines theory search with bottom-up pre-study of data for hypotheses generation in the preparation phase. It uses steps of a data mining process in the execution phase as proposed by Cooper et al. with modifications [Cooper01]. The contributions described in C1, C2 and C3 are pragmatic examples of the value of such repositories in empirical research.

This work covers multiple aspects of software development. Reuse, CBD and incremental development have many advantages, but also require a systematic approach in introducing each and in combining these. Possible directions for future work are:

- **Use cases for effort estimation.** The effort estimation method (C3) should be tested in other projects. There is no standard way of writing use cases.

"Usefulness for estimation" can be defined as a criterion to study practices from this view.

- **Study of RUP adaptations.** A comprehensive literature study on RUP adaptations in other projects is necessary, identifying aspects for classifying and evaluating these and comparing the results.
- **Empirical studies on software evolution.** Some data on requirements as defined in ARS for several releases is still not analyzed. Analyzing these data on requirements evolution between releases will complete the picture of the origin of changes as described in C2.
- **Study of effort distribution.** More empirical studies on effort distribution in different contexts and for different development approaches are necessary to identify the parameters that have most impact on such distributions.
- **The impact of development approaches on product and project metrics.** The model presented in Section 6.2 can be extended for future assessment in university or industrial environments. It was discussed that quick changes in technologies do not allow proper evaluations of them. However, this is not the only reason for poor empirical assessment. Other reasons are lack of guidelines (describing what is important to assess) and lack of benchmarking data to compare with.
- **Relevant metrics for incremental development of large systems.** Validating and extending the identified metrics described in C4 with focus on incremental development of component-based systems can be the subject of future work. These metrics are important to define a framework for future research on software evolution and building more complex models on the relations between development approaches and quality attributes.

References to Part I

[Allen98] Allen, P., Frost, S.: *Component-Based Development for Enterprise Systems, Applying the SELECT Perspective*. Cambridge University Press/SIGS, Cambridge, 1998.

[Alshayeb03] Alshayeb, M., Li, W.: An Empirical Evaluation of Object-Oriented Metrics in Two Different Iterative Software Processes. *IEEE Trans. Software Engineering*, 29(11), pp. 1043-1049, November 2003.

[Arif02] Arif, T., Hegde, L.C.: Inspection of Object-Oriented Construction. NTNU master's thesis spring 2002, 165 p. www.idi.ntnu.no/grupper/su/su-diploma-2002/Arif-OORT_Thesis-external.pdf.

[Arisholm04] Arisholm, E., Sjøberg, D.: Evaluating the Effect of a Delegated versus Centralized Control Style on the Maintainability of Object-Oriented Software. To appear in *IEEE Trans. Software Engineering*, 30(7), July 2004.

[Arlow02] Arlow, J., Neustadt, I.: *UML and The Unified Process. Practical Object-Oriented Analysis and Design*. Addison-Wesley, 2002.

[Atkinson02] Atkinson, C., Bayer, J., Bunse, C., Kamsties, E., Laitenberger, O., Laqua, R., Muthig, D., Paech, B., Wüst, J., Zettel, J.: *Component-based Product Line Engineering with UML*. Addison-Wesley, 2002.

[Atkinson03] Atkinson, C., Kuhne, T.: Aspect-Oriented Development with Stratified Frameworks. *IEEE Software*, 20(1), pp. 81-89. January/February 2003.

[Bachmann00] Bachmann, F., Bass, L., Buhman, C., Comella-Dorda, S., Long, F., Robert, J., Seacord, R., Wallnau, K.: Volume II: Technical Concepts of Component-Based Software Engineering. *SEI Technical Report* number CMU/SEI-2000-TR-008. http://www.sei.cmu.edu/

[Barbacci00] Barbacci, M.R., Ellison, R.J., Weinstock, C.B., Wood, W.G.: Quality Attribute Workshop, Participants Handbook. *SEI Special Report*, CMU/SEI-2000-SR-001, 2000. http://www.sei.cmu.edu/publications/

[Basili75] Basili, V., Turner, J.: Iterative Enhancement: A Practical Technique for Software Development. *IEEE Trans. Software Engineering*, 1(12), pp. 390-396, December 1975.

[Basili84] Basili, V.R., Weiss, D.: A Methodology for Collecting Valid Software Engineering Data. *IEEE Trans. Software Engineering*, 10(11), pp. 758-773, November 1984.

[Basili01] Basili, V.R., Boehm, B.: COTS-Based Systems Top 10 List. *IEEE Computer*, 34(5), pp. 91-93, May 2001.

[Baskerville03] Baskerville, R., Ramesh, B., Levine, L., Pries-Heje, J., Slaughter, S.: Is Internet-Speed Software Development Different? *IEEE Software*, 20(6), pp. 70-77, November/December 2003.

[Bass00] Bass, L., Buhman, C., Comella-Dorda, S., Long, F., Robert, J., Seacord, R., Wallnau, K: Volume I: Market assessment of Component-based Software

Engineering. *SEI Technical Report* number CMU/SEI-2001-TN-007. http://www.sei.cmu.edu/

[Beaurepaire04] Beaurepaire, O., Lecardeux, B., Havart, C.: Industrialization of Software-Quality-Led Project Management Process at the SNCF (French Railways). *Proc. the 8th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering* (QAOOSE'04), Olso, Norway, June 15, 2004, pp. 47-59.

[Benediktsson03] Benediktsson, O., Dalcher, D.: Developing a new Understanding of Effort Estimation in Incremental Software Development Projects. *Proc. Intl. Conf. Software & Systems Engineering and their Applications* (ICSSEA'03), December 2-4, 2003, Paris, France. Volume 3, Session 13, ISSN 1637-5033, 10 p.

[Bennett00] Bennett, K., Rajlich, V.: Software Maintenance and Evolution: A Roadmap. Proc. The Conference on the Future of Software Engineering, June 04-11, 2000, Limerick, Ireland, pp. 73-87. Anthony Finkelstein (Ed.), *ACM Press* 2000, Order number 592000-1, ISBN 1-58113-253-0.

[Bergström03] Bergström, S., Råberg, L.: *Adopting the Rational Unified Process, Success with the RUP*. Addison-Wesley, 2003.

[Bluetooth04] The Official Bluetooth website, cited June 29, 2004. http://www.Bluetooth.com

[Boehm85] Boehm, B.W.: A Spiral Model of Software Development and Enhancement. Proc. Int'l Workshop Software Process and Software Environments. *ACM Press*, 1985, also in *ACM Software Engineering Notes*, August 1986, pp. 22-42.

[Boehm95] Boehm, B.W., Clark, B., Horowitz, E., Westland, C., Madachy, R., Selby, R.: Cost Models for Future Software Life Cycle Processes: COCOMO 2.0. *USC center for software engineering*, 1995. http://sunset.usc.edu/publications/TECHRPTS/1995/index.html

[Bosch00] Bosch, J.: *Design and Use of Software Architectures: Adpoting and Evolving a Product-line Approach*. Addison-Wesley, 2000.

[Bosch02] Bosch, J.: Maturity and Evolution in Software Product Lines: Approaches, Artifacts and Organization. Proc. of the 2nd Software Product Line Conference-SPLC2, *LNCS 2379 Springer* 2002, ISBN 3-540-43985-4, pp. 257-271. http://www.cs.rug.nl/~bosch/

[Briand01] Briand, L., Wüst, J.: Integrating Scenario-based and Measurement-based Software Product Assessment. *Journal of Systems and Software*, 59(1), pp. 3-22. *SEI Report* No. 42.00/E, *ISERN Report* No. ISERN-00-04. http://www.sce.carleton.ca/faculty/briand/isern-00-04.pdf, http://www.sce.carleton.ca/Squall/pubs_journal.html#2001

[Briand02] Briand, L., Wüst, J.: Empirical Studies of Quality Models in Object-Oriented Systems. Advances in Computers, *Academic Press*, Vol. 56, pp. 97-166, updated 18 February 2002. http://www.harcourt-international.com/serials/computers/

[Brownsword00] Brownsword, L., Oberndorf, T., Sledge, C.: Developing New Processes for COTS-Based Systems. *IEEE Software*, 17(4), pp. 48-55, July/August 2000.

[Bruin02] de Bruin, H., van Vliet, H.: The Future of Component-Based Development is Generation, not Retrieval. *Proc. CBSE Workshop in the 9th Annual IEEE International Conference and Workshop on the Engineering of Computer-Based Systems- ECBS02*. http://www.idt.mdh.se/~icc/cbse-ecbs2002/

[Bunde02] Bunde, G.A., Pedersen, A.: Defect Reduction by Improving Inspection of UML Diagrams in the GPRS Project. HiA master's thesis spring 2002, 118 p. http://siving.hia.no/ikt02/ikt6400/g08/

[Carney00] Carney, D., Long, F.: What Do you Mean by COTS? Finally, a Useful Answer. *IEEE Software*, 17(2), pp. 83-86, March/April 2000.

[CBSEnet04] The CBSEnet project, July 5, 2004: http://www.cbsenet.org

[Cheesman00] Cheesman, J., Daniels, J.: *UML Components: A Simple Process for Specifying Component-Based Software*. Addison-Wesley, 2000.

[Clements01] Clements, P., Northrop, L.: *Software Product Lines- Practices and Patterns*. Pearson Education (Addison-Wesley), 2001.

[Clements02a] Clements, P.: Being Proactive Pays Off. *IEEE Software*, 19(4), pp. 28-30, July/August 2002.

[Clements02b] Clements, P., Northrop, L.M.: Salion, Inc.: A Software Product Line Case Study. *SEI Technical Report* number CMU/SEI-2002-TR-038, November 2002.

[Cohen02] Cohen, S.: Product Line State of the Practice Report. *SEI Technical Note* number CMU/SEI-2002-TN-01, 2002. http://www.sei.cmu.edu/publications/documents/02.reports/02tn017.html

[Cooper01] Cooper, D.R., Schindler, P.S.: *Business Research Methods*. McGraw-Hill International edition. 7th Edition, 2001.

[Creswell94] Creswell, J.W.: *Research Design, Qualitative and Quantitative Approaches*. Sage Publications, 1994.

[Creswell03] Creswell, J.W.: *Research Design, Qualitative, Quantitative, and Mixed Methods Approaches*. Sage Publications, 2002.

[Crnkovic02] Crnkovic, I., Larsen, M.: *Building Reliable Component-Based Software Systems*. Artech House Publishers, 2002.

[Cusumano97] Cusumano, M.A., Selby, R.W.: *Microsoft Secrets: How the World's Most Powerful Software Company Creates Technology, Shapes Markets, and Manages People*. HarperCollins, 1997.

[D'Souza98] D'Souza, D.F., Wills, A.C.: *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Addison-Wesley, 1998.

[Eeles01] Eeles, P.: Capturing Architectural Requirements. *The Rational Edge*, November 2001. http://www.therationaledge.com/nov_01/t_architecturalRequirements_pe.html

[Ekeroth00] Ekeroth, L., Hedström, P.M.: GPRS Support Nodes. *Ericsson Review*, 2000:3, pp. 156-169.

[Endres03] Endres, A., Rombach, D.: *A Handbook of Software and Systems Engineering, Empirical Observations, Laws, and Theories*. Person Education Limited, 2003.

[Ericsson99] Ericsson Press Release, cited 29 June 2004. http://www.ericsson.com/press/archive/1999q1/19990125-0025.html

[Ericsson04a] cited April 10, 2004. http://www.ericsson.com/products/main/GSM_EDGE_WCDMA_hpaoi.shtml,

[Ericsson04b] AXE, cited June 26, 2004. http://www.ericsson.com/technology/tech_articles/AXE.shtml

[Escalante03] Escalante, M., Gutierrez, P.: CBSE: State of the Practice. *Proc. of the Sixteenth International Conference on Software & Systems Engineering and their Applications (ICSSEA'2003)*, 2-4 December 2003, Paris, Volume 2, Session 9: Reuse & Components, ISSN: 1637-5033, 16 p.

[Fenton97] Fenton, N., Pfleeger, S.L.: *Software metrics: A Rigorous and Practical Approach*. International Thomson Computer Press, 2nd edition, 1997.

[Fenton00a] Fenton, N.E., Neil, M.: Software Metrics: Roadmap. The Conference on the Future of Software Engineering, 04-11 June 2000, Limerick, Ireland, pp. 357-370. Anthony Finkelstein (Ed.), *ACM Press* 2000, Order number 592000-1, ISBN 1-58113-253-0.

[Fenton00b] Fenton, N.E., Ohlsson, N.: Quantitative Analysis of Faults and Failures in a Complex Software System. *IEEE Trans. Software Engineering*, 26(8), pp. 797-814, 2000.

[Flyvbjerg04] Flyvbjerg, B.: Five Misunderstandings about Case-Study Research. In [Seale04], pp. 420-434.

[Frakes95] Frakes, W.B., Fox, C.J.: Sixteen Questions about Software Reuse. *Communications of the ACM*, 38(6), pp. 75-87, June 1995.

[Ghosh02] Ghosh, S.: Improving Current Component-Based Development Techniques for Successful component-Based Software Development. *Proc. International Conference on Software Reuse* (ICSR7), Workshop on Component-based Software Development Processes, 2002. http://www.idt.mdh.se/CBprocesses/

[Gilb76] Gilb, T.: *Software Metrics*. Chartwell-Bratt, 1976.

[Gilb88] Gilb, T.: *The Principles of Software Engineering Management*. Addison-Wesley, 1988.

[Graham97] Graham, L., Henderson-Sellers, B., Younessi, H.: *The OPEN Process Specification*. Addison-Wesley, 1997.

[Griss93] Griss, M.L.: Software Reuse: From Library to Factory. *IBM Systems Journal*, November-December 1993, 32(4), pp. 548-566.

[Griss95] Griss, M.L., Wosser, M.: Making Reuse Work in Hewlett-Packard. *IEEE Software*, 12(1), pp. 105-107, January 1995.

[Grundy00] Grundy, J.C.: An Implementation Architecture for Aspect-Oriented Component Engineering. Proc. 5th International Conference on Parallel and Distributed Processing Techniques and Applications: Special Session on Aspect-oriented Programming, *CSREA Press*. http://www.cs.auckland.ac.nz/~john-g/aspects.html

[GSM04] The GSM Association, cited July 5, 2004: http://www.gsmworld.com/technology/gprs/intro.shtml#8

[Heineman01] Heineman, G.T., Councill, W.T.: *Component-Based Software Engineering, Putting Pieces Together*. Addison-Wesley, 2001.

[Heires01] Heires, J.T.: What I Did Last Summer: A Software Development Benchmarking Case Study. *IEEE Software*, 18(5), pp. 33-39, September/October 2001.

[Hissam98] Hissam, S., Carney, D: Isolating Faults in Complex COTS-Based Systems. *SEI Monographs on the Use of Commercial Software in Government Systems*, 1998. http://www.sei.cmu.edu/cbs/papers/monographs/isolating-faults/isolating.faults.htm

[Holmen01] Holmen, U.E., Strand, P.: Selection of Software Development Process- A Study of Changes in the Software Development Process of Five Norwegian Companies. Project work report at NTNU, autumn 2001, 52 p. http://www.idi.ntnu.no/grupper/su/sif8094-reports/2001/p10.pdf

[IDI-SU04] Software Engineering Group at the Department of Computer and Information Science (IDI), NTNU, publications: http://www.idi.ntnu.no/grupper/su/index.php3?file=publ/INT-PUBL.php3

[IEEE1219] IEEE Standard 1219: Standard for Software Maintenance. *IEEE Computer Society Press*, 1993.

[INCO01] The INCO project: http://www.ifi.uio.no/~isu/INCO/

[Jacobson97] Jacobson, I., Griss, M., Jonsson, P.: *Software Reuse: Architecture, Process and Organization for Business Success*. Addison-Wesley, 1997.

[Jacobson03] Jacobson, I.: Use Cases and Aspects - Working Seamlessly Together. *Journal of Object Technology*, 2(4), pp. 7-28, July/August 2003. http://www.jot.fm

[Jalote04] Jalote, P., Palit, A., Kurien, P., Peethamber, V.T.: Timeboxing: A Process Model for Iterative Software Development. *The Journal of Systems and Software*, 2004:70, pp. 117-127.

[Jazayeri00] Jazayeri, M., Ran, A., van der Linden, F.: *Software Architecture for Product Families*. Addison-Wesley, 2000.

[Johnson98] Johnson, R.E., Foote, B.: Designing Reusable Classes. *Journal of Object-Oriented Programming*, 1(3), pp. 26-49, 1998.

[Juristo01] Juristo, N., Moreno, A.: *Basics of Software Engineering Experimentation*. Boston Kluwer Academic, 2001.

[Jørgensen04] Jørgensen, M., Sjøberg, D.: Generalization and Theory Building in Software Engineering Research. Accepted at the *8th International Conference on Empirical Assessment in Software Engineering (EASE2004)*, May 24-25, 2004, Edinburgh, Scotland.

[Kang90] Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. *SEI Technical Report* CMU/SEI-90-TR-21, SEI, Carnegie Mellon University, Pittsburgh, 1990.

[Karlsson95] Karlsson, E.A. (Ed.): *Software Reuse, a Holistic Approach*. John Wiley & Sons, 1995.

[Karlsson02] Karlsson, E.A.: Incremental Development- Terminology and Guidelines. In *Handbook of Software Engineering and Knowledge Engineering, Volume 1*. World Scientific, 2002, pp. 381-401

[Kent99] Kent, B.: *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.

[Kim02] Kim, H., Boldyreff, C.: Dveloping Software Metrics Applicable to UML Models. *Proc. 6th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering* (QAOOSE'02), June 11, 2002, University of Málaga, Spain. http://alarcos.inf-cr.uclm.es/qaoose2002/

[Kitchenham95] Kitchenham, B.A., Pickard, L., Pfleeger, S.L.: Case studies for Method and Tool Evaluation. *IEEE Software* 12(4), pp. 52-62, July 1995.

[Kitchenham01] Kitchenham, B.A., Hughes, R.T., Linkman, S.G.: Modeling Software Measurement Data. *IEEE Trans. Software Engineering*, 27(9), pp. 788-804, September 2001.

[Kitchenham02] Kitchenham, B.A., Pfleeger, S.L., Hoaglin, D.C., Rosenberg, J.: Preliminary Guidelines for Empirical Research in Software Engineering. *IEEE Trans. Software Engineering,* 28(8), pp. 721-734, August 2002.

[Kruchten00] Kruchten, P.: *The Rational Unified Process. An Introduction*. Addison-Wesley, 2000.

[Kruchten01] Kruchten, P.: The Nature of Software: What's so Special about Software Engineering? *The Rational Edge*, October 2001. http://www.therationaledge.com/

[Krueger02] Krueger, C.: Eliminating the Adoption Barrier. *IEEE Software*, 19(4), pp. 29-31, July/August 2002.

[Lanza02] Lanza, M., Ducasse, S.: Beyond Language Independent Object-Oriented Metrics: Model Independent Metrics. *Proc. 6th ECOOP workshop on Quantitative Approaches in Object-Oriented Software Engineering* (QAOOSE'02), 2002. http://alarcos.inf-cr.uclm.es/qaoose2002/QAOOSE2002AccPapers.htm

[Larman03] Larman, C., Basili, V.R.: Iterative and Incremental Development: A Brief History. *IEEE Computer*, 36(6), pp. 47-56, June 2003.

[Larsson00] Larsson, M., Crnkovic, I.: Development Experiences of Component-Based System. *Proc. 7th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems*, Edinburgh, Scotland, April 2000, pp. 246-254.

[Lientz78] Lientz, B.P., Swanson, E.B., Tompkins, G.E.: Characteristics of Application Software Maintenance. *Communications of the ACM*, 21(6), pp. 466-471, June 1978.

[Linden02] van der Linden, F.: Software Product Families in Europe: The Esaps and Café Projects. *IEEE Software*, 19(4), pp. 41-49, July/August 2002.

[MacCormack03] MacCormack, A., Kemerer, C.F., Cusumano, M., Crandall, B.: Trade-offs between Productivity and Quality in Selecting Software Development Practices. *IEEE Software*, 20(5), pp. 78-85, September/October 2003.

[Maxwell02] Maxwell, K.D.: *Applied Statistics for Software Managers*. Prentice Hall PTR, 2002.

[MacCormack03] MacCormack, A., Kemerer, C.F., Cusumano, M., Crandall, B.: Trade-offs between Productivity and Quality in Selecting Software Development Practices. *IEEE Software,* 20(5), pp. 78-85, September/October 2003.

[McGarry01] McGarry, J.: When it Comes to Measuring Software, Every Project is Unique. *IEEE Software*, 18(5), pp. 19-20, September/October 2001.

[McGregor02] McGregor, J.D., Northrop, L.M., Jarred, S., Pohl, K.: Initiating Software Product Lines. *IEEE Software*, 19(4), pp. 24-27, July/August 2002.

[McIlroy69] McIlroy, D.: Mass-produced Software Components. *Proc. Software Engineering Concepts and Techniques*, 1968 NATO Conference on Software Engineering, Buxton, J.M., Naur, P., Randell, B. (eds.), January 1969, pp. 138-155, available through Petroceli/Charter, New York, 1969.

[Mendenhall95] Mendenhall, W., Sincich, T.: *Statistics for Engineering and the Sciences*. Prentice Hall International Editions, 1995.

[Mili02] Mili, H., Mili, A., Yacoub, S., Addy, E.: *Reuse-based Software Engineering. Techniques, Organizations, and Controls*. John-Wiley & Sons, 2002.

[Mills76] Mills, H.: Software Development. *IEEE Trans. Software Engineering*, December 1976, pp. 265-273.

[Morisio00] Morisio, M., Seaman, S., Parra, A., Basili, V., Kraft, S., Condon, S.: Investigating and Improving a COTS-Based Software Development Process, Proc. 22nd International Conference on Software Engineering ICSE'2000, Limerick, Ireland, 2000, *IEEE Computer Society Press*, pp. 31-40.

[Morisio02] Morisio, M., Ezran, M., Tully, C.: Success and Failures in Software Reuse. *IEEE Trans. Software Eng*ineering, 28(4), pp. 340-357, April 2002.

[Morisio03] Morisio, M., Torchiano, M.: Definition and Classification of COTS: A Proposal. *Proc. The International Conference on COTS-Based Software Systems* ICCBSS'03, *LNCS 2255*, pp. 165-175, 2003.

[Naalsund01] Naalsund, E., Walseth, O.A.: Component-Based Development, Models for COTS/ Software Reuse. NTNU project work report, autumn 2001, 69 p. http://www.idi.ntnu.no/grupper/su/sif8094-reports/2001/p4.pdf.

[Naalsund02] Naalsund, E., Walseth, O.A.: Decision making in component-based development. NTNU master's thesis, spring 2002, 92 p. http://www.idi.ntnu.no/grupper/su/su-diploma-2002/naalsund_-_CBD_(GSN_Public_Version).pdf

[Neighbors96] Neighbors, J.M.: Finding Reusable Software Components in Large Systems. *Proc. the 3rd Working Conference on Reverse Engineering* (WCRE'96), November 8-10, 1996, Monterey CA, USA, pp. 2-10.

[Nuefelder00] Neufelder, A.M.: How to Measure the Impact of Specific Development Practices on Fielded Defect Density. *Proc. 11th International Symposium on Software Reliability Engineering* (ISSRE'00), 2000, pp. 148-160.

[Noppen02] Noppen, J., Tekinerdogan, B., Aksit, M., Glandrup, M., Nicola, V.: Optimizing Software Development Policies for Evolutionary System Requirements. Proc. ECOOP2002 Workshop Reader, *Springer Verlag LNCS 2548*, 2002. http://www.joint.org/use/2002/sub/

[Northrop02] Northrop, L.M.: SEI's Software Product Line Tenets. *IEEE Software*, 19(4), pp. 32-40, July/August 2002.

[OMG04] The Object Management Group: http://www.omg.org

[Parnas72] Parnas, D.L.: On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12), pp. 1053-1058, December 1972.

[Parnas76] Parnas, D.L.: On the Design and Development of Program Families. *IEEE Trans. Software Eng*ineering, 2(1), pp.1-9, 1976.

[Paul96] Paul, R.A.: Metrics-Guided Reuse. *International Journal on Artificial Intelligence Tools*, 5(1 and 2), pp. 155-166, 1996.

[Pawlak04] Pawlak, R., Younessi, H.: On Getting Use Cases and Aspects to Work Together. *Journal of Object Technology*, 3(1), pp. 15-26, January/February 2004. http://www.jot.fm/issues/issue_2004_01/column2

[Pfleeger93] Pfleeger, S.L.: Lessons Learned in Building a Corporate Metrics Program. *IEEE Software*, 10(3), pp. 67-74, May/June 1993.

[Poulin95] Poulin, J.S.: Populating Software Repositories: Incentives and Domain-Specific Software. *Journal of System and Software*, 1995:30, pp. 187-199.

[Ramesh04] Ramesh, V., Glass, R.L., Vessey, I.: Research in Computer Science: An Empirical Study. *Journal of Systems and Software*, 70(2004), pp. 165-176.

[Rine98] Rine, D.C., Sonnemann, R.M: Investments in Reusable Software: A Study of Software Reuse Investment Success Factors. *The Journal of Systems and Software*, 1998:41, pp. 17-32.

[Saeki02] Saeki, M.: Attribute Methods: Embedding Quantification Techniques to Development Methods. *6th ECOOP workshop on Quantitative Approaches in Object-Oriented Software Engineering* (QAOOSE'02), 2002. http://alarcos.inf-cr.uclm.es/qaoose2002/QAOOSE2002AccPapers.htm

[Schwarz02] Schwarz, H., Killi, O.M., Skånhaug, S.R.: A Study of Industrial Component-Based Development, Ericsson. NTNU project work report, autumn 2002, 105 p. http://www.idi.ntnu.no/grupper/su/sif8094-reports/2002/p2.pdf

[Schwarz03] Schwarz, H., Killi, O.M.: An Empirical Study of the Quality Attributes of the GSN System at Ericsson. NTNU master's thesis, spring 2003, 109 p. http://www.idi.ntnu.no/grupper/su/su-diploma-2003/killi_schwarz-empirical_study_ericsson_external-v1.pdf

[Seale04] Seale, C., Gobo, G., Gubrium, J.F., Silverman, D.: *Qualitative Research Practice*. Sage Publications, 2004.

[Seaman99] Seaman, C.B.: Qualitative Methods in Empirical Studies of Software Engineering. *IEEE Trans. Software Engineering*, 25(4), pp. 557-572, 1999.

[Sedigh-Ali01a] Sedigh-Ali S., Ghafoor, A., Paul, R. A.: Metrics-Guided Quality Management for Component-Based Systems. Proc. 25th Annual International Computer Software and Applications Conference (COMPSAC'01), Chicago, October 8-12, 2001, *IEEE CS Press*, pp. 303-310.

[Sedigh-Ali01b] Sedigh-Ali, S., Ghafoor, A.: Software Engineering Metrics for COTS-Based Systems. *IEEE Computer*, 34(5), pp. 44-50, May 2001.

[SEI04a] The Software Engineering Institute at the Carnegie Mellon University: http://www.sei.cmu.edu/plp/plp_publications.html

[SEI04b] Product Line Approach to Software Development at SEI, cited July 5, 2004: http://www.sei.cmu.edu/plp/plp_init.html

[Simula04] The Simula Research Laboratory, July 5, 2004, http://www.simula.no/

[Singer02] Singer, J., Vinson, N.G.: Ethical Issues in Empirical Studies of Software Engineering. *IEEE Trans. Software Engineering*, 28(12), pp. 1171-1180, December 2002.

[Sommerville01] Sommerville, I.: *Software Engineering*. 6th edition, Addison-Wesley, 2001.

[SPIQ98] The SPIQ project: http://www.idi.ntnu.no/~spiq/

[Standish04] The Standish Group: http://www.standishgroup.com/

[Symons91] Symons, P.R.: *Software Sizing and Estimating MK II FPA (Function Point Analysis)*, John Wiley & Sons, 1991.

[Szyperski97] Szyperski, C.: *Component Software- Beyond Object-Oriented Programming*. Addison-Wesley, 1997.

[Szyperski02] Szyperski, C., (with Gruntz, D., Murer, S.): *Component Software, Beyond Object-Oriented Programming*. Addison Wesley, 2nd edition, 2002.

[Tarr99] Tarr, P., Ossher, H., Harrison, W., Sutton, S.: N Degrees of Separation: Multi-Dimensional Separation of Concerns. Proc. The Int'l Conference on Software Engineering (ICSE'99), *IEEE CS Press,* pp.107-119, 1999.

[Torchiano04] Torchiano, M., Morisio, M.: Overlooked Facts on COTS-Based Development. *IEEE Software*, 21(2), pp. 88-93, March/April 2004.

[UML2.0] UML 2.0 Specifications: http://www.uml.org/

[Voas98] Voas, J.M.: Certifying Off-the-Shelf Software Components. *IEEE Computer*, 31(6), pp. 53-59, June 1998.

[Vigder98a] Vigder, M.R., Gentleman, W.M., Dean, J.: COTS Software Integration: State of the art. *NRC Report* No. 39198, 1998, 22 p. http://wwwsel.iit.nrc.ca/abstracts/NRC39198.abs

[Vigder98b] Vigder, M.: Inspecting COTS Based Software Systems, Verifying an Architecture to Support Management of Long-Lived Systems, *NRC Report* No. 41604, 1998. http://wwwsel.iit.nrc.ca/projects/cots/COTSpg.html.

[Vigder99] Vigder, M.: Building Maintainable Component-Based Systems. *Proc. 1999 International Workshop on Component-Based Software Engineering*, pp. 17-18 May 1999. http://www.sei.cmu.edu/cbs/icse99/papers/38/38.pdf

[Voas01] Voas, J.: Composing Software Component "itilities". *IEEE Software*, 18(4), pp. 16-17, July/August 2001.

[Wallnau98] Wallnau, K.C., Carney, D., Pollak, B.: How COTS Software Affects the Design of COTS-Intensive Systems. *Spotlight*, 1(1), June 1998. http://interactive.sei.cmu.edu/Features/1998/June/cots_software/Cots_Software.htm

[Warsun03] Warsun Najib, Selo: MDA and Integration of Legacy Systems. HiA master's thesis, spring 2003, 85 p. http://fag.grm.hia.no/ikt6400/hovedoppgave/lister/tidl_pro/prosjekter.aspx?db=2003

[Wohlin00] Wohlin, C., Runeseon, P., M. Höst, Ohlsson, M.C., Regnell, B., Wesslén, A.: *Experimentation in Software Engineering*. Kluwer Academic Publications, 2000.

[Yin03] Yin, R.K.: *Case Study Research, Design and Methods*. Sage Publications, 2003.

[Zave98] Zave, P., Jackson, M.: A Component-Based Approach to Telecommunication Software. *IEEE Software*, 15(5), pp. 70-78, September/October 1998.

[Zowghi02] Zowghi, D., Nurmuliani, N.: A Study of the Impact of Requirements Volatility on Software Project Performance. *Proc. 9th International Asia-Pacific Software Engineering Conference* (APSEC'02), 2002, pp. 3-11.

9 *Papers*

This chapter contains the papers in the order given in Section 1.5.

P1. Experiences with Certification of Reusable Components in the GSN Project in Ericsson, Norway

Parastoo Mohagheghi
Ericsson AS, Grimstad, Norway
Tel + 47 37.293069, Fax +47 37.043098
etopam@eto.ericsson.se

Reidar Conradi
Dept. Computer and Information Science, NTNU, NO-7491 Trondheim, Norway
Tel +47 73.593444, Fax +47 73.594466
conradi@idi.ntnu.no

Abstract

Software reuse, or component-based development is regarded as one of the most potent software technologies in order to reduce lead times, increase functionality, and reduce costs. The Norwegian INCO R&D project (INcremental and COmponent-based development) aims at developing and evaluating better methods in this area [9]. It involves the University of Oslo and NTNU in Trondheim, with Ericsson as one of the cooperating industrial companies.

In this paper we discuss the experiences with the process to identify, develop and verify the reusable components at Ericsson in Grimstad, Norway. We present and assess the existing methods for internal reuse across two development projects.

Keywords
Software reuse, Components, Layered system architecture, Software quality, Quality requirements.

1. Introduction

Companies in the telecommunication industry face tremendous commercial and technical challenges characterized by very short time to market, high demands on new features, and pressure on development costs to obtain highest market penetration. For instance, Ericsson has worldwide adopted the following priorities: *faster, better, cheaper* – in that order. Software reuse, or component-based development, seems to be the most potent development strategy to meet these challenges [2][8]. However, reuse is no panacea either [4].

When software components are developed and reused *internally*, adequate quality control can be achieved, but the lead-time will increase. Newer development models, such as incremental development, are promoting reuse of ready-made, *external* components in order to slash lead times. However, external COTS (Components-Off-The-Shelf) introduce new concerns of certification and risk assessment [1]. Both internal and external reuse involves intricate (re)negotiation and prioritization of requirements, delicate compromises between top-down and bottom-up architectural design, and planning with not-yet-released components (e.g. middleware).

The present work is a pre-study of reuse in the GSN (GPRS Support Node, where GPRS stands for General Packet Radio Service) project [6], and where Ericsson in Grimstad, Norway is one of the main participants. We present and assess the existing methods for component identification and certification at Ericsson in Grimstad for reuse across several projects.

In the following, Section 2 presents the local setting. Section 3 introduces the reusable components while Section 4, 5 and 6 discuss the quality schemes for reusable components and certification. Section 7 summarizes experiences and aspects for further study.

2. The Local Setting at Ericsson AS

Ericsson is one of the world's leading suppliers of third generation mobile systems. The aim of software development at Ericsson in Grimstad is to build robust, highly available and distributed systems for real-time applications, such as GPRS and UMTS networks. Both COTS and internal development are considered in the development process. The GSN project at Ericsson has successfully developed a set of components that are reused for applications serving UMTS networks. To support such reuse, the GSN project has defined a common software architecture based on layering of functionality and an overall reuse process for developing the software.

Figure 1 shows the *four GSN architectural layers*: the top-most application-specific layer, the two common layers of business-specific and middleware reusable components, and the bottom system layer. Each layer contains both internally developed components and COTS.

Application systems use components in the common part. Applications address functional requirements, configuration of the total system and share components in the business-specific layer. The middleware layer addresses middleware functionality, non-functional requirements and what is called *system functionality* (to bring the system in an operational state and keep it stable). It also implements a framework for application development.

Application systems sharing this reusable architecture are nodes in the GPRS or UMTS network, both developed by Ericsson AS, the former in Norway and the latter in Sweden. However, the process of identifying reusable components up to the point that they are verified and integrated in a final product, still has shortcomings. Focus in this article is on certification of reusable components in the middleware and business specific layers in Figure 1, what we have called for "common parts" in short.

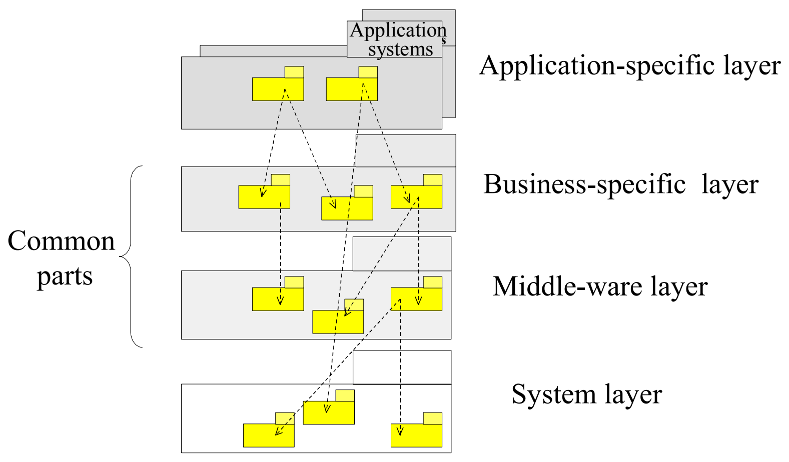


Figure 1 GSN application architecture with four layers

The most important reusable artifact is the software architecture. By (software) architecture we mean a description/specification of the high-level system structure, its components, their relations, and the principles (strategies) and guidelines that govern the design and evolution of the system. The system architecture description is therefore an artifact, being the result of the system design activity.

Middleware is also an artifact that is reused across applications. It addresses requirements from several applications regarding non-functional requirements and traditional middleware functionality. Several business-specific components are also reusable.

3. The Reusable Artifacts

Because of shared functional requirements, use cases and design artifacts (e.g. patterns) may be reused as well. The development process consists of an adaptation of RUP [7], a quality scheme, and configuration management (CM) routines. This process (model) is also a reusable artifact.

We can summarize the *reusable artifacts* as:

- A layered architecture, its generic components and general guidelines.
- Reusable components are either in the business-specific or middleware layers (both internally developed, and called common parts in Fig. 1), or in the basic system layer. Components in the business-specific or middleware layers are mostly written in the proprietary Erlang language, a real-time version of Lisp, and contain almost half part of the total amount of code written in Erlang. The system layer is a platform targeted towards wireless packet data networks containing hardware, operative systems and software for added features.
- Architectural (i.e. design) patterns and more specific guidelines.
- Partly shared requirements and use cases across applications.
- Common process, based on an adaptation of RUP and including a quality scheme and CM routines -see below.

- A development environment based on UML.
- Tools as test tools, debugging tools, simulators, and quality assurance schemes.

The adaptation of the *RUP process* is a joint effort between the GPRS and UMTS organizations in Ericsson. It covers tailoring of subprocesses (for requirement specification, analysis and design, implementation, test, deployment and CM), guidelines for incremental planning, what artifacts should be exchanged and produced, and which tools that should be used and how.

To give a measure of the software complexity, we can mention that the GPRS project has almost 150 KLOC (1000 lines of code excluding comments) written in Erlang, 100 KLOC written in C and 4 KLOC written in Java. No figures are available for the number of reusable components but the applications share more than 60% of the code.

4. The Quality Scheme for the Architecture

The architecture was originally developed to answer the requirements for a specific application (GPRS). Having reuse in mind (between different teams in different organizations), the approach has later been to develop and evolve architectural patterns and guidelines that are reusable also to UMTS applications.

With requirements we mean both functional requirements and non-functional requirements. The latter are called quality requirements in [3], and are either development requirements (e.g. maintainability and reusability) or operational requirements (e.g. performance and fault-tolerance). While it is possible to map functional requirements to specific components, quality requirements depend on architecture, development process, software quality and so on. The architecture should meet all these requirements.

The process of identifying the building blocks of the architecture has partly been a *top-down* approach with focus on functionality, as well as performance, fault-tolerance, and scalability. A later recognition of shared requirements in the extended domain (here UMTS) has lead to a *bottom-up,* reverse engineering of the developed architecture to identify reusable parts across applications. This implies a joint development effort across teams and organizations. However, we do not yet have a full-fledged product-line architecture.

Some important questions to verify reuse of the architecture are:

- How well can the architecture and components for a specific product meet the requirements for other products? The answer may lie in the degree of shared requirements. The project has succeeded to reuse the architecture, generic components and patterns in such a wide degree that it justifies investments considering development with reuse.
- How well are the components documented? How much information is available on interfaces and internal implementations? As mentioned initially, this is easier to co-ordinate when components are developed inside Ericsson and the source code is available. Nevertheless one of the most critical issues in reuse is the quality of the documentation, which should be improved. The Rational UML tool is used in the development environment and all interfaces, data types and packages are documented in the model. In addition guidelines, APIs (Application Programming Interfaces) and other documentation are available.

- How well the developed architecture meets the operational requirements in the domain? This has been based on knowledge of the domain and the individual components, overall prototyping, traffic model estimations, intensive testing, and architectural improvements.
- How well the developed architecture meets the development requirements? It is not easy to answer as measuring the maintainability or flexibility of an architecture needs observations over a time. But we mean that the developed architecture has the potential to address these aspects. This is discussed more in the coming chapter.

As mentioned, design patterns and guidelines are also considered part of the architecture. A design pattern is a solution to a common problem. Hence when similarities between problems are recognized, a verified solution is a candidate for generalization to a pattern. This solution must however have characteristics of a reusable solution regarding flexibility, design quality, performance etc. A large number of patterns are identified and documented for modeling, design, implementation, documentation or test. Based on the type of pattern, different teams of experts should approve the pattern.

5. Certification of the Architecture Regarding Quality Requirements

The architecture is designed to address both functional and quality (non-functional) requirements. While the functional requirements are defined as use cases, quality requirements are documented as the *Supplementary Specifications* for the system. One of the main challenges in the projects is the task of breaking down the quality requirements to requirements towards architecture, components in different layers or different execution environments. For instance a node should be available for more than 99.995% of the time. How can we break down this requirement to possible unavailability of the infrastructure, the platform, the middleware or the applications? This is an issue that needs more discussion and is not much answered by RUP either.

All components should be optimized be certified by performing inspections and unit testing. When the components are integrated, integration testing and finally target testing are done. The project however recognized that the architecture and the functionality encapsulated in the middleware layer (including the framework) address most of the quality requirements. The first step is then to capture the requirements towards architecture and the middleware layer:

- In some cases, a quality requirement may be converted to a use case. If such a conversion is possible, the use case may be tested and verified as functional use cases. For example the framework should be able to restart a single thread of execution in case it crashes.
- Other requirements are described in a Supplementary Specification for the middleware. This contains the result of breaking down the quality requirements towards the node when it was possible to do so, requirement on documentation, testability etc.

Discussion on how to best capture quality requirements is still going on.

Quality requirements as performance and availability are certified by development of scenarios for traffic model and measuring the behavior, simulation, and target testing.

The results should be analyzed for architectural improvements. Inspections, a database of trouble reports and check lists are used for other requirements as maintainability and documentation.

The architecture defines requirements to applications to adopt a design pattern or design rule to fulfill quality requirements as well.

The final question is how to predict the behavior of the system for quality requirements? Domain expertise, prototyping, simulations and early target testing are used to answer this. Especially it is important to develop incrementally and test as soon as possible to do adjustments, also for the architecture.

6. The Quality Scheme for Developing New Components

The process for software reuse is still not fully organized and formalized. When the decision for reuse is taken, the development process (RUP) should be modified to enhance the potential for reuse. The current process is summarized in the following steps:

a) The first question when facing a new component is how generic this component will be. The component may be placed in the application-specific layer, the business-specific layer (reusable for applications in the same domain), or the middleware layer (the most generic part).
b) If the component is recognized to be a reusable one:
    - Identify the degree of reusability.
    - Identify the cost of development to make the component reusable (compared to the alternative of developing a solution specified and optimized for a specific product).
    - Identify the cost of optimization, specialization and integration, if the component is developed to be more generic.
c) Develop a plan for verifying the component. This depends on the kind of component and covers inspections, prototyping, unit testing and system testing, before making it available as a reusable part by running extra test cases. A complete verification plan may cover all these steps.

When the reuse is across products and organizations in Ericsson, a joint team of experts (called the Software Technical Board, SW TB) takes the decision regarding shared artifacts. The SW TB should address identification to verification of the reusable component and together with the involved organizations decide which organization *owns* this artifact (should handle the development and maintenance). Teams in different product areas support the SW TB.

7. Experiences and Suggestions for Further Improvements

As mentioned, an adaptation of RUP has been chosen to be the development process. The development is incremental where the product owners and the software technical board jointly set priorities. Reuse is recognized to be one of the most important technologies to achieve reduced lead-time, increased quality, and reduced cost of development. Another positive experience with reusable process, architecture and tools is that organizations have easier access to skilled persons and shorter training periods in case of replacements.

Some aspects for further consideration regarding reuse are:

1. Improving the process for identifying the common components. This is mainly based on expertise of domain experts rather than defined characteristics for these components.
2. Coupling the development of common parts to the development plan of products using them.
3. Finding, adopting, improving or developing tools that makes the reuse process easier. An example is use of the multi-site ClearCase tool for configuration management of files.
4. Improving and formalizing the RUP-based, incremental development process and teaching the organization to use the process. This is not always easy when development teams in different products should take requirements from other products into consideration during planning. Conflicts between short-time interests and long-term benefits from developing reusable parts must be solved, see e.g. [5].
5. Developing techniques to search the developed products for reusable parts and improving the reuse repository.
6. Define a suitable **reuse metrics**, collect data according to this, and use the data to improve the overall reuse process.

The topic of certifying the architecture and the system regarding quality requirements should be more investigated and formalized. Some aspects are:

1. Improve the process of breaking down the quality requirements.
2. Improve the development process (an adaptation of RUP) on how to capture these requirements in the model or specifications.
3. Improve planning for certification of quality requirements. While functional requirements are tested early, test of quality requirements has a tendency to be delayed to later phases of development, when it is costly to change the architecture.

8. Conclusions

Implementing software reuse combined with incremental development is considered to be the technology that allows Ericsson to develop faster, better and cheaper products. However, future improvement of the technology, process, and tools is necessary to achieve even better results. The INCO project aims to help Ericsson in measuring, analyzing, understanding, and improving their reuse process, and thereby the software products.

9. References

1. Barry W. Boehm and Chris Abts: "COTS Integration: Plug and Pray?", *IEEE Computer*, January 1999, p. 135-138.

2. Barry W. Boehm et al.: "*Software Cost Estimation with Cocomo II* (with CD-ROM)", August 2000, ISBN 0-130-26692-2, Prentice Hall, 502 p. See also slides from the FEAST2000 workshop in 10-12 July, 2000, Imperial College, London, htpp://www-dse.doc.ic.ac.uk/~mml/f2000/pdf/Boehm_keynote.pdf.

3. Jan Bosch: "*Design & Use of Software Architectures: Adopting and evolving a product line approach*", Addison-Wesley, May 2000, ISBN 0-201-67494-7.
4. Dave Card and Ed Comer: "Why Do So Many Reuse Programs Fail", *IEEE Software*, Sept. 1994, p. 114-115.
5. John Favaro: "A Comparison of Approaches to Reuse Investment Analysis", Proc. Fourth International Conference on Software Reuse, 1996, *IEEE CS Press*, p. 136-145.
6. GPRS project at Ericsson: http://www.ericsson.com/3g/how/gprs.html
7. Ivar Jacobson, Grady Booch, and James Rumbaugh: "*The Unified Software Development Process*", Addison-Wesley Object Technology Series, 1999, 512 p., ISBN 0-201-57169-2 (on the Rational Unified Process, RUP).
8. Guttorm Sindre, Reidar Conradi, and Even-André Karlsson: "The REBOOT Approach to Software Reuse", *Journal of Systems and Software* (Special Issue on Software Reuse), Vol. 30, No. 3, (Sept. 1995), p. 201-212, http://www.idi.ntnu.no/grupper/su/publ/pdf/jss.df.
9. Dag Sjøberg / Reidar Conradi: "INCO proposal for NFR's IKT-2010 program", 15 June 2000, Oslo/Trondheim, 52 p., http://www.idi.ntnu.no/grupper/su/inco.html.

P2. Reuse in Theory and Practice: A Survey of Developer Attitudes at Ericsson

Parastoo Mohagheghi, Reidar Conradi, Erlend Naalsund, Ole Anders Walseth
Ericsson Norway-Grimstad, Postuttak, NO-4898 Grimstad, Norway
Department of Computer and Information Science, NTNU, NO-7491 Trondheim, Norway
parastoo.mohagheghi@eto.ericsson.se, conradi@idi.ntnu.no

Abstract

The goal of software process models is to help developers to decide what to do and when to do it. However, it is often a gap between the process model and the actual process. Ericsson has successfully developed two large-scale telecommunication systems based on reusing the same architecture, framework, and many other core assets. However, the software process model is not updated for reuse. We performed a survey in the organization to evaluate developer attitudes regarding reuse and the software process model, and to study the effect of the gap between the process model and the practice of reuse. The results showed that the developers are aware of the importance of reuse and are motivated for it. It also showed that lack of explicit guidelines on reuse has impact on the reuse practice, such as insufficient documentation and testing of reusable components. Although a reuse repository was not considered important, the participants answered that introducing explicit activities related to reuse would improve the process model.

Keywords
Reuse, product line engineering, software process improvement, survey.

1. Introduction

Many organizations are using a product line approach for software development by exploiting commonalities between software systems and thus reusing a set of core assets. The approach to start a product line or system family can be either *heavyweight* or *lightweight,* depending on the context. The main difference between these two approaches is the degree to which some reusable assets are identified before the first product [15, 16].

Developing families of systems include activities for identifying commonalities and differences, developing reusable core assets such as a common software architecture and framework, developing applications based on the reusable assets, and planning and managing product lines. Software processes for reuse-based or product line engineering [1, 4, 5, 11, 13, 14] provide concepts and guidelines to plan for reuse, and to create and evolve systems that are based on large-scale reuse. The assumption is that organizations that design for families of systems, rather than a single system, should do this consciously and reflect their practice in their software process model.

Ericsson has developed two products to deliver GPRS (General Packet Radio Service) to the GSM and UMTS networks using a lightweight approach. These products share a common software process, an adaptation of the Rational Unified Process or RUP [21], software architecture, and core assets. Although the adaptation of RUP has been done in parallel with initiating the system family, it has not been adapted for this aspect of development and thus lacks explicit guidelines for reuse and system family engineering. I.e. there is a gap between the process model (the adapted RUP process) and the actual process (the practice of software development). We wanted to study the developer attitudes regarding reuse, and to decide whether to initiate a software process improvement activity to improve the process model.

We performed a survey in the organization with questions on reuse and the process model. Results of the survey are used to evaluate four null hypotheses, and to explore the improvement areas. Our results confirm that developers are aware of the importance of reuse, perceive reused components as more stable and reliable, and are motivated for changes in the process model to promote reuse. It also shows the importance of the existing knowledge and expertise in the software development process. We finally introduce a set of improvement suggestions to the process model.

The study was done as part of a MSc diploma thesis at the Norwegian University of Science and Technology (NTNU) and in the scope of the INCO project. INCO (INcremental and COmponent-based engineering) is a cooperative project between NTNU and the University of Oslo (the latter as coordinator), funded by the Norwegian Research Council.

The remainder of the paper is structured as follows: Section 2 describes some state of the art. Section 3 describes the Ericsson context and Section 4 is on the research problem. Section 5 describes the questionnaire used in the survey, the defined null hypotheses, and the main results. The null hypotheses are evaluated in Section 6. Section 7 discusses the validity threats, further results, and improvement suggestions to the process model. The paper is concluded in Section 8.

2. System families and reuse

Parnas wrote the first paper on development of systems with common properties in 1976. He wrote:” We consider a set of programs to constitute a *family*, whenever it is worthwhile to study programs from the set by *first* studying the common properties of the set and *then* determining the special properties of the individual family members” [20]. He called these systems program families, while the most recent terms are *system families*, *application families* or *product lines*. The Software Engineering Institute’s (SEI) Product Line Practices initiative has used the definition of a *software product line* as “a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission, and that are developed from a common set of core assets in a prescribed way” [5]. Hence system families are built around *reuse*: reuse of requirements, reuse of software architecture and design, and reuse of implementation. Especially important is reuse of *software architecture*, being defined as: “Structure or structures of the system, which compromise software components, the externally visible properties of those components, and the relationships among them” [3].

2.1. Role of the component frameworks in promoting reuse and developing system families

*Object-oriented framework*s have been proposed as a reusable software architecture that embodies an abstract design and which is extended mainly using specialization [4, 14]. With increasing use of component-based approaches, component models and component frameworks are introduced. Sometimes these two terms are used interchangeably, while Bachman and some others separate these two: "A *component model* defines the standards and conventions imposed on developers of components. A *component framework* is implementation of services that support or enforce a component model" [2, 10]. A well-known component model (and partially framework) is the Object Management Group's (OMG's) CORBA (Common Object Request Broker Architecture). A component framework serves several purposes:

- Like operating systems, frameworks are active and act directly on components to manage its lifecycle or resources [2].
- They capture design decisions and define standards for component developers, where the goal is to satisfy certain performance specifications (or quality attributes).
- They define a software architecture for a particular domain [1] and hence can be part of the reference architecture.
- They capture commonalities in the application domain, and define mechanisms to handle variability.

Customized frameworks are developed for a specific domain, and serve the same role as standard component frameworks.

2.2. How to initiate a system family?

We distinguish between two main approaches for introducing a system family: *heavyweight* and *lightweigh*t. In the heavyweight approach, commonalities are identified *first* by domain engineering and product variations are foreseen. In the lightweight approach, a first product is developed and the organization then uses mining efforts to extract commonalities [16]. The choice of approach also affects cost and the organization structure. With a heavyweight approach, the initial cost of a product line is significantly higher than for a single product. But after a few products, the product line is assumed to have lower cumulative costs. A heavyweight approach also needs a two-tiered organization for development of reusable assets and development of products. With a lightweight approach, the organization can delay the organizational changes to after the first product.

Krueger claims that the lightweight approach can reduce the adoption barrier to large-scale reuse, as it is a low-risk strategy with lower upfront cost [15]. Often an organization does not have time or resources to initiate a product line from the start, or wants to explore the market first, or initiate a family from products currently in production. Johnson and Foote write in [12] that useful abstractions are usually designed from the bottom up; i.e. they are discovered not invented. Hence the chosen approach and the degree to which some assets are delivered before the first product varies, and there is no single approach for all circumstances.

2.3. Software processes for engineering system families

Several software development processes support product line engineering and reuse. Examples are Jacobson, Griss and Jonsson's approach [11], the REBOOT method (REuse Based on Object-Oriented Techniques) with it's emphasize on development *for* and *with* reuse [14], Feature-Oriented Domain Analysis (FODA) [13], and the more recent KobrA approach [1]. SEI defines three essential product line activities [19]:

1. Core asset development or *domain engineering* for developing the architecture and the reusable assets (development *for* reuse)
2. *Application engineering* to build the individual products (development *with* reuse)
3. *Management* at the technical and organizational level.

When developing several systems based on some reusable assets, the focus is on identifying commonalities and planning for variations. Therefore software processes will include activities to handle these two aspects in all phases of software development; from requirement engineering to deployment and configuration management. With increasing use of component-based approaches, activities for component development, utilizing COTS (Commercial-Off-The-Shelf) components and developing systems based on components are also included in software processes.

3. The Ericsson context

Telecommunication and data communication are converging disciplines, and packet-switched services open for a new era of applications. The General Packet Radio Service (GPRS) system provides a solution for end-to-end Internet Protocol (IP) communication between a mobile entity and an Internet Service Provider (ISP). The GPRS Support Nodes (GSNs) constitute the parts of the Ericsson cellular system core network that switch packet data. The two main nodes are the Serving GPRS Support Node (SGSN) and the Gateway GPRS Support Node (GGSN) [8].

3.1. The system family for GSM and UMTS

The GSNs were first developed to provide packet data capability to the GSM (Global System for Mobile communication) cellular network. A later recognition of shared requirements with the forthcoming UMTS system (Universal Mobile Telecommunication System) lead to reverse engineering of the developed architecture to identify reusable parts across applications and to evolve the architecture to an architecture that can support both products. This was a joint development effort across teams and organizations for several months, with negotiations and renegotiations. The enhanced, hierarchical reuse-based GSN architecture is shown in Figure 1. Both systems are using the same platform (WPP), which is a high-performance packet switching platform developed by Ericsson. They also share components in the business specific layer and the middleware layer (called *Common parts* in Figure 1). The business-specific components offer services for the packet switching networks. The middleware provides a customized component framework for building robust, real-time applications for processing transactions in a distributed multiprocessor environment that use CORBA and its Interface Definition Language (IDL) [17]. The organization has also been adapted to this view: an organization unit is assigned to develop common parts, while other units develop the applications. The reusable assets are evolved in parallel with the products, taking into account requirements from both products.

Figure 1 is one view of the system architecture, where the hierarchical structure is based on what is common and what is application specific. Other views of the architecture reveal that all components in the application and business-specific layers use the framework in the middleware layer, and all components in the three upper layers use the services offered by WPP.

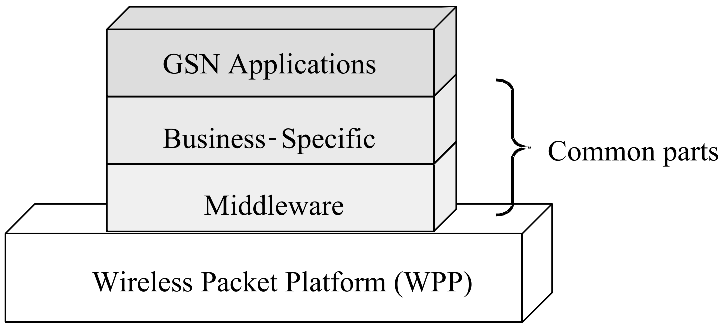


Figure 1 The GSN architecture

The reused components in the common parts stand for 60% of the code in an application, where an application in this context is a product based on WPP and consisting of the three upper layers. Size of each application (not including WPP) is over 600 NKLOC (Non-Commented Lines Of Code measured in equivalent C code). Software components are mostly developed internally, but COTS components are also used. Software modules are written in C, Java and Erlang (a programming language for programming concurrent, real-time, distributed fault-tolerant systems). Several Ericsson organizations in Sweden, Norway and Germany have cooperated in developing the GSNs, but recently the development is moved to Sweden.

GSN’s approach to develop a system family has been a lightweight approach: The first product was initially developed and released, and the commonalities between the developed system, and the requirements for the new product lead to the decision on reuse. The organization used mining efforts to extract the reusable assets and enhanced the architecture as a baseline for developing new products. The approach gave much shorter time-to-market for the second product, while the first one could still meet its hard schedules for delivery.

3.2. State of the GSN software process

The software process has been developed in parallel with the products. The first products were using a simple, internally developed software process, describing the main phases of the lifecycle and the related artifacts. After the first release, the organization decided to adapt the RUP. The adaptation is done by adding, removing or modifying phases, activities, roles and artifacts in the standard RUP process. The adapted process is defined and maintained for the GSN projects by an internal unit in the organization, with people from two organizations in Norway and Sweden. The products are developed incrementally, and new features are added to each version of the products.

RUP is an architecture-centric process, which is an advantage when dealing with families of systems using the same reference architecture. But RUP in its original form is not a process for system families. As explained in Section 2.3, software processes for building system families include reuse-related activities. Although the adaptation of RUP has been done in parallel with initiating the system family, it has not been adapted for this aspect of development. The main workflows (requirement, analysis and design, implementation and testing) are described as if there is a single product development, while configuration management activities handle several versions and several products. There is no framework engineering in the adapted RUP, and developing framework components is an indistinguishable part of application engineering. To provide the information needed for software developers, artifacts such as internally developed modeling guidelines and design rules are linked to the workflows in RUP, and play a complementary role to the process model. At this stage, the process looks like an ad-hoc approach to reuse and system family development, where pieces are added to the software process without realizing the affect of this patching.

4. The research problem

Bridging the gap between the process model and the actual process can be subject of a software process improvement activity. But why to start an improvement activity aimed at the process model, when the organization already has successfully designed and evolved a system family with extensive reuse, using a "reuse-free" software process model? Many studies show that software is not developed according to the process model anyway. For example in [6], Parnas and Clements show a graph of a software designer's activities over time, where activities (requirement, design etc.) are performed at seemingly random times. So why define an ideal process model when no one follows it in practice? The authors answer that the organization should attempt to produce the ideal process for different reasons. Below are some of their reasons and some reasons added by us:

- "Designers need guidance". A well-documented process model describes what to do first and how to proceed.
- "We will come closer to a rational design if we try to follow the process (model) rather than proceed on an ad-hoc basis". If the process is adapted for reuse and system family engineering, it will promote reuse and design for change; i.e. to foresee future variability and evolution.
- "If we have agreed on an ideal process, it becomes much easier to measure the progress".
- The process model shows the outsiders how the products are developed, and therefore should reflect the practice.

Software process assessment is central in any improvement activity, where the goal is to understand the current process and to identify and plan areas that can be improved. The research questions we posed in our research were:

RQ1: Does the lack of explicit reuse-related activities in the process model affect the reuse practice?
RQ2: How the developers experience the current process model?
RQ3: Are the developers motivated for change?

To answer the above questions, we developed a set of hypotheses. Verification of the hypotheses was done based on the results of a survey in the organization.

5. Survey: Hypotheses and questions

The following four null hypotheses were defined:

H01: Reuse in software development gives no significant advantages.
H02: It is easy for a given design/code component to choose between reuse "as-is", reuse "with modification", or developing from scratch.
H03: The current process model works well.
H04: Criteria for compliance with existing architecture are clearly defined.

Participants in the survey were 10 developers of the same development team, and included 8 designers and 2 testers. We got 9 filled-in questionnaires back. The team was selected because their work was ready for inspection (which was object for another experiment on inspection of UML models), and they could assign time to participate in the survey (designed to take less than one hour for each). This is non-probability sampling, based on convenience [23]. The range of their experience in Ericsson was varying: 1 person with only 9 months of experience, 7 persons with experience from 2-5 years, and one person with 13 years of experience. The sample size is 5%, and the participants had different roles in the team and different years of experience in the organization. The conclusion is that the sample is representative for the organization in Grimstad. The participants were unaware of our hypotheses, and they have answered the questionnaires separately.

Table 1 shows an overview of the questions, their relation to the null hypotheses (some questions are not related to any hypothesis), results for most of the questions, and references to the figures containing other results. The abbreviations in Table 1 are CBD for Component-Based Development, OO for Object-Oriented, CM for Configuration Management, RM for Requirement Management, A&D for Analysis & Design, NFR for Non-Functional Requirements, and GSN RUP for the adapted RUP process. Answers are either Yes, No, or Sometimes/To some degree (shown as *Other*). Q6 and Q18 are shown separately for two reasons: They had other alternatives than Yes/No, and 3 participants had (wrongly) selected more than one answer. More details on some of the results are given below.

Q1a-e: As shown in Figure 2, the participants consider shorter development time as the most important advantage of reuse, followed by lower development costs and a more standardized architecture.

Q3a-e: Requirements for the system are specified first in text and stored in a database. The functional requirements are later specified in use cases, while the non-functional requirements (NFR) are specified in Supplementary Specifications. As shown in Figure 4, *Design* was considered the artifact being most important to be reused (8 participants rated it as very high to high). Test data/documentation is of secondary importance.

Q22: The question was: "GSN RUP does not include reuse activities such as activities for comparing candidate components, evaluating existing components and deciding whether to reuse or not. Will introducing such activities have positive effect on the development process/have no effect or have negative effect?". Here 8 participants answered that it will have positive effect, and one meant that it wouldn't have any impact.

6. Evaluation of hypotheses

Evaluation of H01: H01 states that reuse gives no significant advantage. 8 questions were related to H01: Q1a-e, Q2a, Q9 and Q10. As shown in Figure 1, the participants answered that reuse give advantages such as shorter time-to-market and lower development costs. In Q2a, 8 participants answered that reuse and component-based technologies are of very high or high importance. In Q9, 6 participants mean that a reused component is more stable and causes fewer problems than a new one. The only result in favor of the null hypotheses is the result of Q10, where the participants mean that integration of reused components might cause problems. *Hence H01 is rejected.*

Evaluation of H02: H02 states that it is easy to decide between reusing a component as it is, reusing with modifications or developing a new component from scratch. 2 questions were directly related to H02: Q5 and Q6. 5 participants meant that the existing process for finding, assessing and reuse of components does not work well and 6 answered that they consult experts when taking this decision, in addition to using the process and guidelines. Several questions give indications that taking such decision is not easy and the reason may be insufficient documentation of the framework and reusable assets (Q7a-b, Q17), or unclear criteria regarding compliance with architecture (Q23a-b*). Hence H02 is rejected.*

Evaluation of H03: H03 states that the current process model works well. We discussed the reuse aspect in H02. 4 questions are related to the adapted RUP process: Q18-Q21. Most participants always or often refer to GSN RUP during requirement management, or analysis and design. However Q18 shows that the main source of information during analysis and design is previous work, and not the process model. All 9 participants said that the GSN RUP web pages are understandable. Our interpretation of the results is that although GSN RUP is frequently used, experts and experience plays an important role. *All in all, we can't reject H03.*

Evaluation of H04: H04 states that criteria for architectural compliance are clearly defined. 3 questions were related to this. In Q23a, 7 participants meant that the criteria are defined to some degree but are rather fuzzy, and in Q23b, 8 participants answered that this is often or sometimes a problem. In Q24, 5 participants said that criteria for design regarding non-functional requirements are not clearly defined. *Hence H04 is rejected.*

Table 1 Survey questions, relation to null hypotheses, and results

	Null Hypotheses				Answers			
Questions	**H01**	**H02**	**H03**	**H04**	**Yes**	**Other**	**No**	**Blank**
General on reuse								
Q1a-e: Benefits of reuse: Lower development costs, shorter development time, higher product quality, standard architecture and lower maintenance costs.	x				See Figure 2.			
Q2a-f: Importance of approaches/activities: Reuse/CBD, OO development, testing, inspections, formal methods and CM.	x (2a)				See Figure 3.			
Q3a-e: What is important to be reused: Requirements, use cases, design, code, test data/documentation.					See Figure 4.			
Reuse in the project								
Q4: Reuse is as high as possible.					4	1	3	1
Q5: Is the process of finding, assessing and reusing existing code/design components functioning?		x			4		5	
Q6: How do you decide whether to reuse a code/design component "as-is", reuse "with modification", or make a new component from scratch?		x			See below.			
Q7a: Are the existing code/design components sufficiently documented?		x				3	5	1
Q7b: If 'Sometimes' or 'No': Is this a problem?		x			7		1	1
Q8: Would the construction of a reuse repository be worthwhile?					3		4	2
Reused components								
Q9: A reused component is usually more stable/reliable.	x				6	2	1	
Q10: Integration when reusing components works usually well.					1		7	1
Q11: Is any extra effort put into testing/documenting potentially reusable components?					4		5	
Q12: Do you test a component for non-functional properties before integration with other components?					2	4	2	1

Table 1 (Cont.) Survey questions, relation to null hypotheses, and results

	Null Hypotheses				**Answers**			
Questions	**H 0 1**	**H 0 2**	**H 0 3**	**H 0 4**	**Yes**	**Othe r**	**No**	**Blan k**
Requirements								
Q13: Is the requirements renegotiation process working efficiently?					4		4	1
Q14: In a typical project, requirements are usually flexible.					3	4	1	1
Q15: Are requirements often changed / renegotiated during a project?					6	2		1
Component Framework								
Q16a: Do you know components of the component framework well?					4		5	
Q16b: Do you know interfaces of the component framework well?					4		5	
Q16c: Do you know design rules of the component framework well?					6		3	
Q17: Is the component framework sufficiently documented?		x			2		6	1
GSN RUP								
Q18: What is your main source of guideline information during A&D?			x		See below.			
Q19: Do you always/often refer to GSN RUP workflows during RM?			x		6		1	2
Q20: Do you always/often refer to GSN RUP workflows during A&D?			x		8		1	
Q21: Is the information in the GSN RUP web pages understandable?			x		9			
Q22: Will introducing reuse activities in GSN RUP have positive effect?					8		1	
Architecture compliance								
Q23a: Are criteria for compliance with architecture clearly defined?				x	1	7	1	
Q23b: If not 'Yes', does these shortcomings often lead to problems?				x	2	6		1
Q24: Are criteria for design regarding NFR well defined?				x	3		5	1

Q6: How do you decide whether to reuse a code /design component "as-is", reuse "with modification", or make a new component ?	Guidelines	Experts	GSN RUP	Not defined
	3	6	4	2

Q18: What is your main source of guideline information during A&D?	Other developers	Previous work	GSN RUP
	3	7	4

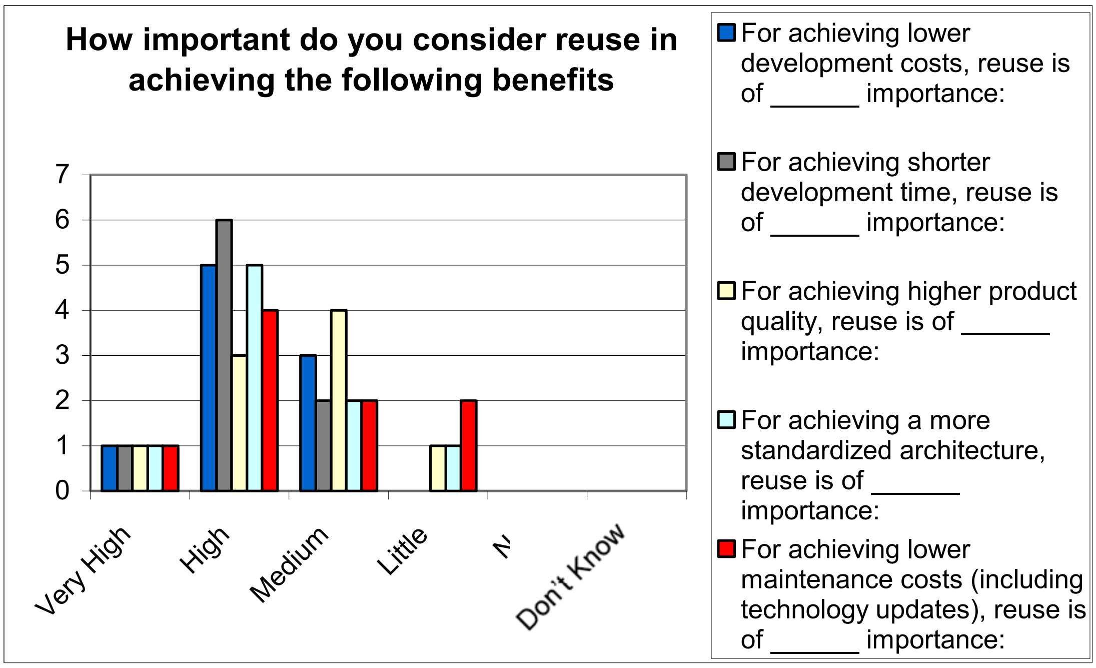


Figure 2 Results of Q1a-e. Columns are in the same sequence as in the description field.

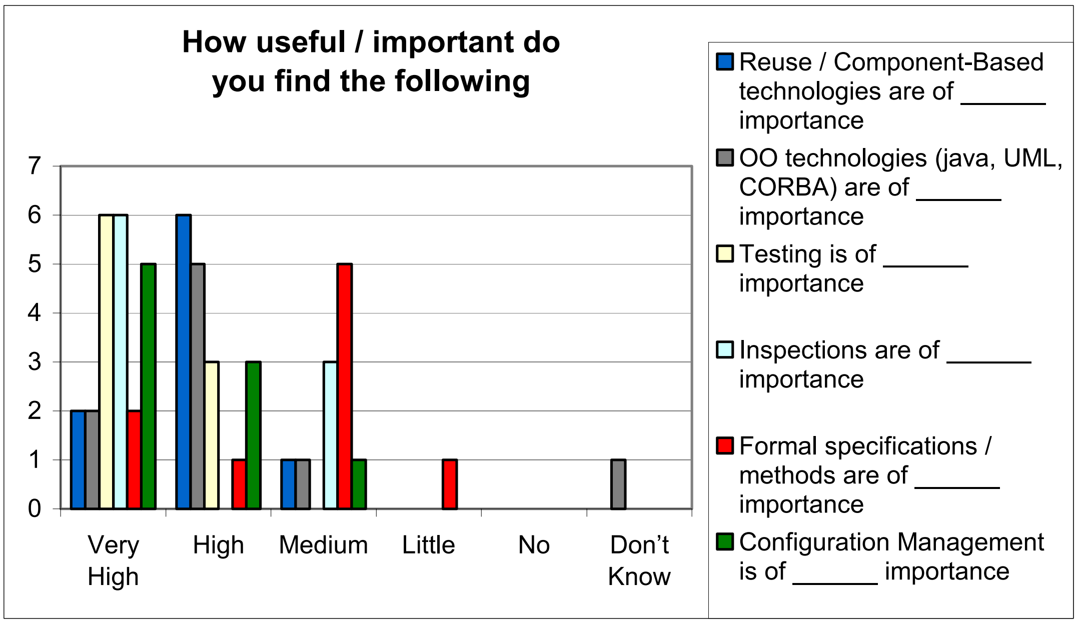


Figure 3 Results of Q2a-f. Columns are in the same sequence as in the description field.

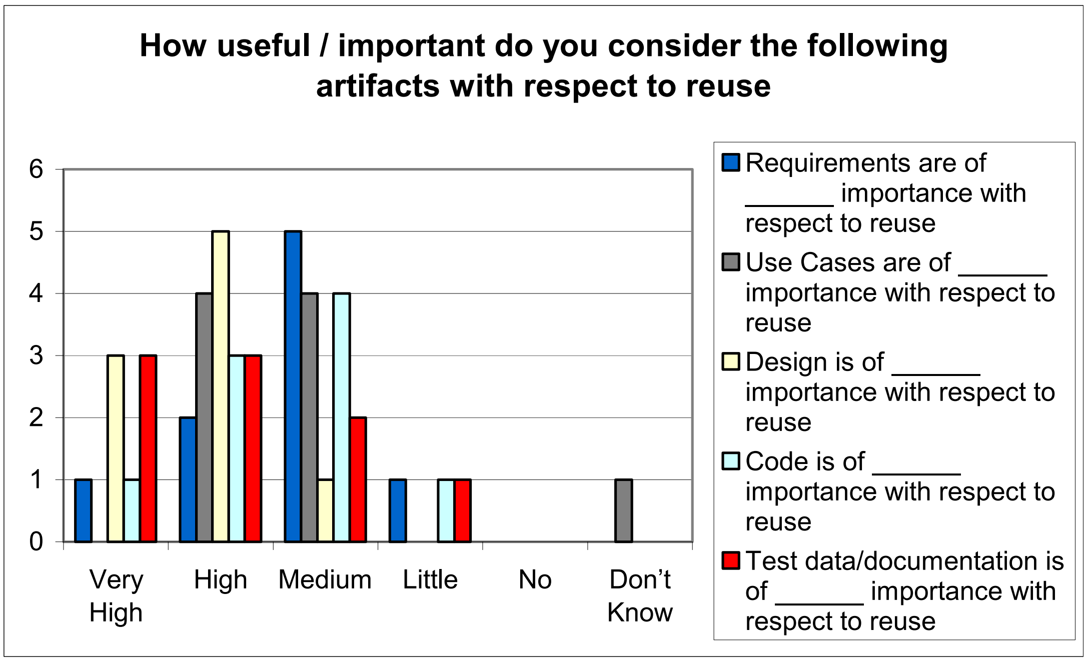


Figure 4 Results of Q3a-e. Columns are in the same sequence as in the description field.

7. Discussion

We discuss the validity threats of our study, discuss other results from the survey, and introduce our improvement suggestions.

7.1.Validity discussion

Threats to experimental validity are classified and elaborated in [23]. Threats to validity of this survey are:

Internal validity: the participants' previous knowledge and experience on some approaches to software development can have impact on their answers for Q1-Q3. For example formal methods are not used in the project and may therefore be rated as less important.

External validity: It is difficult to generalize the results of the survey to other organizations as the participants were from the same organization. However we find examples of similar surveys performed in several organizations (such as in [9]) and studies on reuse and SPI (such as theses defined in [7], which we compare our results with to evaluate external validity.

Construct validity: No threats are identified.

Conclusion validity: We have not performed statistical analysis on the results when we evaluated the hypotheses. The questionnaire had few participants.

7.2. Further interpretation of the results

We asked the participants on the importance of testing, inspections and configuration management. The interesting result is that all of them are rated as very highly important in Figure 3. Testing and configuration management are areas supported by computerized tools. Thesis 3 in [7] suggests that only these areas with stable processes are well suited for computerized tools, while the creativity factor is more important in other areas such as modeling.

The GSNs do not have any reuse repository and the participants rely on the collaborative work, internal experts and the existing architecture to take reuse-related decisions or find reusable assets. The result of Q8 is not in favor of reuse repositories either. The impact of CASE tools and reuse repositories on promoting reuse is also studied by in [9], and the conclusion was that neither of them has been effective in promoting reuse.

When it comes to requirements, 3 participants said that the requirements are usually flexible and 4 answered that they sometimes are (Q14), and 6 participants meant that requirements often change (Q15). Data from the projects show indeed that the requirement stability has been decreasing, and that 20-30% of the requirements change during lifetime of a project.

We had two questions regarding non-functional requirements. In Q12 we asked whether developers test components for non-functional (often called quality) properties before integration. 2 participants answered yes, 4 participants answered sometimes and 2 answered no. In Q24, 5 participants said that criteria for design regarding non-functional requirements are not well defined (which may be the reason for not testing for these requirements), while only 3 said that they are well defined. The adapted RUP process has activities for specification of such requirements, but our results show need for improving specification and verification of non-functional requirements as well.

In **RQ1** we asked whether the lack of explicit reuse-related activities in the process model affect the reuse practice. We notice symptoms that can support such conclusion:

- Reused components are not sufficiently documented.
- Assessing components for reuse is not easy.
- Criteria for architectural compliance are not clearly defined.
- Components are not sufficiently tested for non-functional requirements.

RQ2 is related to H02-H04 and is already discussed.

In **RQ3**, we asked whether developers are motivated for change. 8 participants answered that introducing reuse-related activities would improve the process model, and thus they are motivated for change. This is in line with Conradi and Fuggetta's thesis in [7] that developers are motivated for change and many SPI initiatives should therefore be started bottom-up.

The survey in [9] concludes that most developers prefer to reuse than to build from scratch. We got the same conclusion in Q9 where the participants meant that a reused component is more stable and reliable than a new one.

Our results in Q6 and Q18 show the high importance of expertise and experience, and having examples from previous work (shall we call it for three ex-es?) in software development. These factors compensate for the shortcomings in the process model.

7.3. Improvement suggestions

Ericsson has already performed several process audits and larger surveys on the GSN RUP process. The goal with a process audit is to assess the process conformance; i.e. to assess consistency between the process model and the execution. We had questions that are relevant for a process audit (Q18-22) but our study was mostly focused on attitudes regarding reuse and reuse in practice. As the process model does not have guidelines on reuse-related activities and system family engineering, the scope of our study is beyond process conformance. We think that the process model should get consistent with the actual process. We have presented suggestions on reuse activities that can be added to the adapted RUP in [18] and [22]. Some of these are listed shortly below. Our further work on this issue is stopped at the moment due to the organizational changes in Ericsson in Norway.

Based on the survey results and similar studies, we concluded that a process improvement activity should not focus on building a reuse repository or change of tools, but provide better guidelines for reuse and system family development. Our baseline is the existing process model with 4 phases defined in RUP (Inception, Elaboration, Construction and Transition) and a fifth phase added by Ericsson (Conclusion), with several workflows in each of them (requirement management, analysis and design, etc). We suggest these modifications to the process model:

1. Adding the activity *Additional requirement fulfillment analysis* to the requirement workflow. The goal is to find whether a reused component has additional functionally that is value-adding or should be disabled.
2. Adding these activities to the Inception Phase: a) *Plan reuse strategy* with a decision point on *Make vs. Reuse vs. Buy*. b) *Domain analysis*.
3. Adding the activities *Feasibility study of COTS* and *Renegotiation of requirements* to the Elaboration Phase. It should also have a second decision point on *Make vs. Reuse vs. Buy.*
4. Adding the activity *Updating of documentation* to the Elaboration, Construction and Transition Phases, especially for reusable components.
5. Adding the activity *Record reuse experience* to the Conclusion phase.
6. Distinguishing *framework engineering* and *application engineering* in line with processes such as KobrA [1].

Some of the suggestions are easier to introduce than others. For example introducing framework engineering or domain analysis will have impact on many workflows, while suggestions 1, 4 and 5 have less impact. Priority of the improvement suggestions should be decided as well.

SPI initiatives should be coherent with business goals and strategies. Improving the process model into a process for large-scale reuse and system family development is definitely coherent with Ericsson's business goals.

8. Conclusions

The GSN applications have a high degree of reuse and share a common architecture and process model. The lightweight approach to reuse has been successful in achieving shorter time-to-market and lower development costs. However the process model does not reflect software development in practice. We posed several questions in the beginning of this study: Does lack of explicit reuse-related activities have impact on the reuse practice? What are developers attitudes regarding reuse? Can we defend initiating software process improvement activities to bridge the gap between theory and practice?

We concluded that developers are aware of the importance of reuse, perceive reused components as more stable and reliable, and are motivated for changes in the process model to promote reuse. We also mentioned that insufficient documentation of reusable assets or difficulties in assessment of components for reuse can be related to the lack of explicit guidelines in the process model. As the software is developed incrementally and the project has been running for 5 years, the existing knowledge and the internally developed guidelines compensate for shortcomings in the process model. In Section 4 we discussed why it is necessary to improve the process model, and in Section 7.3 we introduced some improvement suggestions that may be integrated into the adapted RUP process.

We think that a gap between the process model and the actual process is fairly common. Process conformance studies focus on consistency between these two. However, we usually assume that the process model is more mature than the actual process, which is not the case here. We think that this study provided us valuable insight into the practice of reuse and we believe that improving the software process model will promote reuse and improve the reuse practice. Our improvement suggestions to the adapted process may be reused in other adaptation works as well.

9. Acknowledgements

We thank Ericsson in Grimstad for the opportunity to perform the survey.

10. References

[1] C. Atkinson, J. Bayer, C. Bunse, E. Kamsties, O. Laitenberger, R. Laqua, D. Muthig, B. Paech, J. Wust, J. Zettel, "*Component-based Product Line Engineering with UML*", Addison-Wesley, 2002.

[2] F. Bachman, L. Bass, C. Buhman, S. Comella-Dorda, F. Long, J. Robert, R. Seacord, K. Wallnau, "Volume II: Technical concepts of Component-based Software Engineering", *SEI technical report number CMU/SEI-2000-TR-008*. http://www.sei.cmu.edu/

[3] L. Bass, P. Clements, R. Kazman, "*Software Architecture in Practice*", Addison-Wesley, 1998.

[4] J. Bosch, "*Design and Use of Software Architecture: Adpoting and Evolving a Product-Line Approach*", Addison-Wesley, 2000.

[5] P. Clements, L.M. Northrop, "*Software Product Lines: Practices and Patterns*", Addison-Wesley, 2001.

[6] P.C. Clements, D.L. Parnas "A Rational Design Process, How and Why to Fake it", *IEEE Trans. Software Eng*., SE-12(2):251-257, Feb. 1986.

[7] R. Conradi, A. Fuggetta, "Improving Software Process Improvement", *IEEE Software*, 19(4):92-99, July-Aug. 2002.

[8] L. Ekeroth, P.M. Hedstrom, "GPRS Support Nodes", *Ericsson Review*, No. 3, 2000, 156-169.

[9] W.B. Frakes, C.J. Fox, "Sixteen Questions about Software Reuse", *Comm. ACM*, 38(6):75-87, 1995.

[10] G.T. Heineman, W.T. Councill, "*Computer-Based Software Engineering, Putting the Pieces Together*", Addison-Wesley, 2001.

[11] I. Jacobson, M. Griss, P. Jonsson, "*Software Reuse: Architecture, Process and Organization for Business Success*", ACM Press, 1997.

[12] R.E. Johnson, B. Foote, "Designing Reusable Classes", *Journal of Object-Oriented Programming*, 1(3):26-49, July-Aug. 1988.

[13] K. Kang, S. Cohen, J. Hess, W. Novak, A. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study (*CMU/SEI-90-TR-21*, ADA 235785). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1990.

[14] E.-A. Karlsson (Ed.), "*Software Reuse, a Holistic Approach*", John Wiley & Sons, 1995.

[15] C. Krueger, "Eliminating the Adoption Barrier", *IEEE Software*, 19(4):29-31, July-Aug. 2002.

[16] J.D. McGregor, L.M. Northrop, S. Jarred, K. Pohl, "Initiating Software Product Lines", *IEEE Software*, 19(4):24-27, July-Aug. 2002.

[17] P. Mohagheghi, R. Conradi, "Experiences with Certification of Reusable Components in the GSN Project in Ericsson, Norway", Proc. 4th ICSE Workshop on Component-Based Software Engineering: Component certification and System Prediction, ICSE'2001, Toronto, Canada, May 14-15, 2001, 27-31.

[18] E. Naalsund, O.A. Walseth, "Decision Making in Component-Based Development", NTNU diploma thesis, 14 June 2002, 92 p., http://www.idi.ntnu.no/grupper/su/su-diploma-2002/naalsund_-_CBD_(GSN_Public_Version).pdf

[19] L.M. Northrop, "SEI's Software Product Line Tenets", *IEEE Software*, 19(4):32-40, July-Aug. 2002.

[20] D.L. Parnas, "On the Design and Development of Program Families", *IEEE Trans. Software Eng.*, SE-2(1):1-9, March 1976.

[21] Rational Unified Process, Rational Home Page, http://www.rational.com

[22] H. Schwarz, O.M. Killi, S.R. Skånhaug, "Study of Industrial Component-Based Development", NTNU pre-diploma thesis, 22 Nov. 2002, 105 p. http://www.idi.ntnu.no/grupper/su/sif8094-reports/2002/p2.pdf

[23] C. Wohlin, P. Runeson, M. Höst, M.C. Ohlsson, B. Regnell, A. Wesslén, "*Experimentation in Software Engineering, an Introduction*", Kluwer Academic Publishers,2000.

P3. MDA and Integration of Legacy Systems: An Industrial Case Study

Parastoo Mohagheghi[1], Jan Pettersen Nytun[2], Selo[2], Warsun Najib[2]
[1]Ericson Norway-Grimstad, Postuttak, N-4898, Grimstad, Norway
[1]Department of Computer and Information Science, NTNU, N-7491 Trondheim, Norway
[1]Simula Research Laboratory, P.O.BOX 134, N-1325 Lysaker, Norway
[2]Agder University College, N-4876 Grimstad, Norway
parastoo.mohagheghi@eto.ericsson.se, jan.p.nytun@hia.no

Abstract

The Object Management Group's (OMG) Model Driven Architecture (MDA) addresses the complete life cycle of designing, implementing, integrating, and managing applications. There is a need to integrate existing legacy systems with new systems and technologies in the context of MDA. This paper presents a case study at Ericsson in Grimstad on the relationship between the existing models and MDA concepts, and the possibility of model transformations to develop models that are platform and technology independent. A tool is also developed that uses the code developed in Erlang, and CORBA IDL files to produce a structurally complete design model in UML.

1. Introduction

The success of MDA highly depends on integration of legacy systems in the MDA context, where a legacy system is any system that is already developed and is operational. Legacy systems have been developed by using a variety of software development processes, platforms and programming languages. Ericsson has developed two large-scale telecommunication systems based on reusing the same platforms and development environment. We started a research process (as part of the INCO project [3]) to understand the development process in the context of MDA, and to study the possibility to transform from a PSM to a PSM at a higher level of abstraction, or to a PIM. Part of the study is done during a MSc thesis written in the Agder University College in spring 2003 [8]. We studied what a platform is in our context, which software artifacts are platform independent or dependent, and developed a tool for model transformation, which may be part of an environment for round-trip engineering.

The remainder of the paper is structured as follows: Section 2 describes some state of the art. Section 3 presents the Ericsson context, and Section 4 describes platforms in this context and transformations. Section 5 describes a tool for transformation, and the paper is concluded in Section 6.

2. Model-Driven Architecture

The Model-Driven Architecture (MDA) starts with the well-known and long established idea of separating the specification of the operation of a system from the details of the way that the system uses the capabilities of its platform [5]. The requirements for the system are modeled in a *Computation Independent Model* (CIM) describing the situation in which the system will be used. It is also common to have an information model (similar to the ODP information viewpoint [4]) that is computation independent. The other two core model concepts in MDA are the *Platform Independent Model* (PIM) and the *Platform Specific Model* (PSM). A PIM describes the system but does not show details of how its platform is being used. A PIM may be transformed into one or more PSMs. In an MDA specification of a system, CIM requirements should be traceable to the PIM and PSM constructs that implement them, and vice versa [5]. Models are defined in the Unified Modeling Language (UML) as the OMG's standard modeling language. UML meta-models and models may be exchanged between tools by using another OMG standard, the XML Metadata Interchange (XMI).

Model transformation is the process of converting one model to another model of the same system [5]. An MDA *mapping* provides specifications for transformation of a PIM into a PSM for a particular platform. Mapping may be between a PIM to another PIM (model refinement for example to build a bridge between analysis and design), PIM to PSM (when the platform is selected), PSM to PSM (model refinement during realization and deployment), or PSM to PIM (reverse engineering and extracting core abstractions).

Like most qualities, platform independence is a matter of degree [5]. When a model abstracts some technical details on realization of functionality, it is a PIM. However it may be committed to a platform and hence be a PSM.

3. The Ericsson Context

GPRS (General Packet Radio Services) provides a solution for end-to-end Internet Protocol (IP) communication between a mobile entity and an Internet Service Provider (ISP). Ericsson has developed two products to deliver GPRS to the GSM (Global System for Mobile communication) and W-CDMA (Wideband Code Division Multiple Access) networks [1].

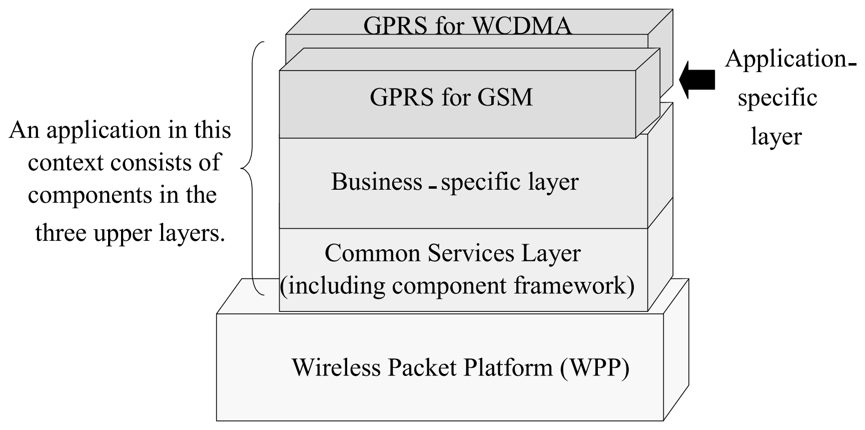

GPRS for WCDMA
GPRS for GSM
Application-specific layer
An application in this context consists of components in the three upper layers.
Business - specific layer
Common Services Layer (including component framework)
Wireless Packet Platform (WPP)

Figure 1 The GPRS Nodes software architecture

Figure 1 is one view of the software architecture, where the hierarchical structure is based on what is common and what is application specific. Other views of the architecture reveal that all components in the application-specific and business-specific layers use a component framework in the common services layer, and all components in the three upper layers use the services offered by WPP [6]. Size of each application is over 600 NKLOC (Non-commented Kilo Lines Of Code measured in equivalent C code). Software components are mostly developed internally, but COTS components are also used. Software modules are written in C, Erlang (a functional language for programming concurrent, real-time, and distributed systems [2]), and Java (only for user interfaces). The software development process is an adaptation of the Rational Unified Process (RUP) [7]. UML modeling is done by using the Rational Rose tool.

4. Platforms and Transformations

Figure 2 shows the software process from requirements to the executables, several models representing the system, and the relationships between these models and the MDA concepts.

The use case model, domain object model, use case specifications and supplementary specifications (textual documents) are developed in the Requirement workflow. Requirements of the system are then *transformed* to classes and behavior (as described in sequence diagrams) in the Analysis workflow. Design is a *refinement* of analysis, adding new classes, interfaces and subsystems, and assigning them to components. Elements in the design model are subsystems, blocks (each subsystem consists of a number of blocks), units (each block consists of a number of units) and software modules (each unit is realized in one or several modules). IDL files are either generated from the component model, or written by hand. From these IDL files, skeletons and stubs are generated, and finally realization is done manually.

Some subsystems in the design model make a component framework for real-time distributed systems that uses CORBA and its Interface Definition Language (IDL), and Erlang/OTP for its realization (OTP stands for Open Telecommunication Platform, which offers services for programmers in Erlang [2]). In the design phase, it may be seen as a technology-neutral virtual machine as described by MDA (a virtual machine is defined as a set of parts and services, which are defined independently of any specific platform and which are realized in platform-specific ways on different platforms. A virtual machine is a platform, and such a model is specific to that platform [5]).

RUP calls moving from one model to another one for translation, transformation or refinement. Hence software development in the adapted RUP process may also be seen as a series of transformations. However a transformation in RUP is different from a transformation in MDA, since a transformation in MDA starts from a complete model and have a record of transformation. UML models and other artifacts developed in the requirement workflow describe the system in the *problem domain* (as required by the GPRS specifications), and not in the *solution domain*. These are part of a PIM that is not computationally complete. Models in the analysis workflow describe the system in the solution domain and are also part of a PIM. It is first in the design workflow that we could have a computationally complete PIM (that contains all the information necessary for generating code), but it is dependent on the component framework with its realization in CORBA and OTP. On the other hand, each PSM at a higher level of abstraction is a PIM relative to the PSM at the lower level (less technology dependent).

The curved gray arrow in Figure 2 shows a tool called Translator, which is described in Section 5.

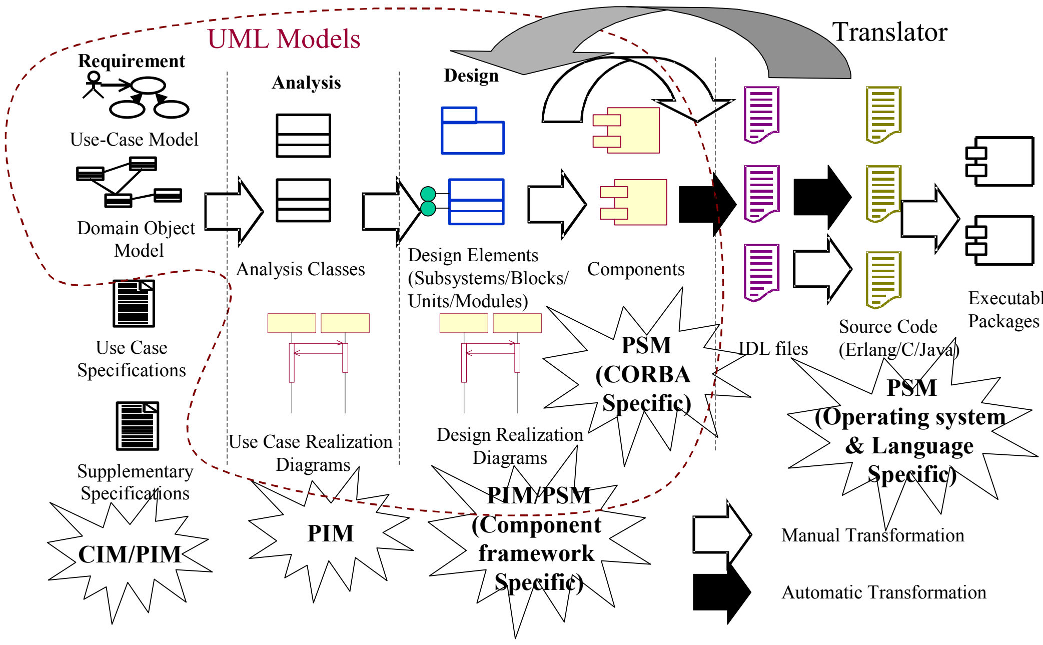

UML Models
Translator
Requirement
Use-Case Model
Domain Object Model
Use Case Specifications
Supplementary Specifications
Analysis
Analysis Classes
Use Case Realization Diagrams
Design
Design Elements (Subsystems/Blocks/ Units/Modules)
Design Realization Diagrams
Components
IDL files
Source Code (Erlang/C/Java)
Executable Packages
CIM/PIM
PIM
PIM/PSM (Component framework Specific)
PSM (CORBA Specific)
PSM (Operating system & Language Specific)
Manual Transformation
Automatic Transformation

Figure 2 From requirements to executables

We notice that most transformations are done manually and therefore:

- There is a risk for inconsistencies between textual requirements and the UML models, between different UML models, and between UML models and the code. Inspections and testing are performed to discover such inconsistencies, which are costly.
- Developers may update the code, IDL files, or the design model without updating other models.

Not all models are developed completely. The analysis model (consisting of analysis classes and sequence diagrams describing the behavior) is only developed for a fraction of use cases. The reason is simply the cost. Another example is the design model where not all the units are completely modeled. If the platform changes, there is not a complete PIM for generation of a PSM in another platform.

5. The Translator

We studied the possibility of reverse engineering the code in order to develop a complete PIM or PSM. We restricted our study to the Erlang environment in the first phase. Our method is based on:

- Filtering out parts of the code that is platform specific, where a platform in this context is the Erlang/OTP platform and CORBA. Among these aspects were operations for starting and restarting the applications and processes, consistency check, transaction handling (a set of signaling messages interchanged between

software modules aiming at completion of a common task), and communication mechanisms during message passing.
- Combing the code with IDL files: Erlang is a dynamically typed language, and the programmer does not declare data types. Therefore we had to use the IDL files to extract data types.
- Using XMI for model exchange.

We studied several commercial tools but ended with making our own tool, the *Erlang to XMI Translator*. The reason was that none of the tools supported reverse engineering from Erlang code or from the sequence diagrams in the design model (although these diagrams are neither complete nor always synchronized with changes in the code).

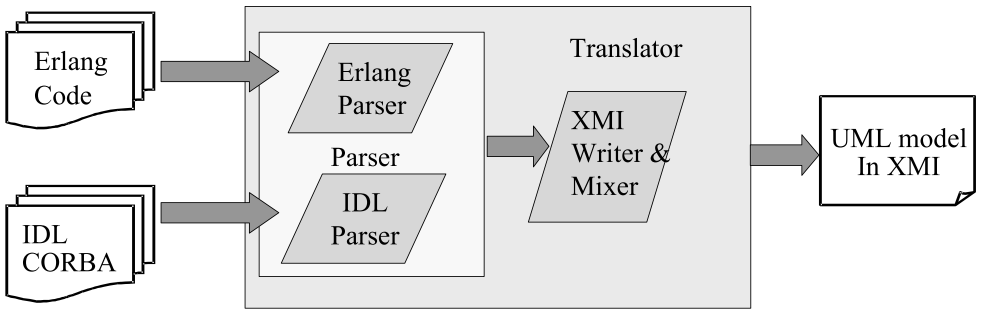


Figure 3 The Erlang to XMI Translator

The resulting UML model is in XMI, which may be opened by other tools such as Rational Rose (the Rose plug-in for XMI must be installed). As we recognized the need to be able to separately parse single subsystems (parsing the total system takes too long time and a subsystem may be updated at any time), we have developed an XMI mixer that combines separate XMI files (from the translator or other tools that export UML models in XMI) and generates a complete model. The tool is developed in Java. The resulting model has the following characteristics:

- It is still dependent on the internally developed component framework and uses its services. However, it is independent of CORBA, the Erlang language and OTP.
- It is a *structurally complete model*, and shows the complete structure of the design model. However it does not have information on the behavior. We have not extracted the behavior of the system that is described in the code. To do so, we would need an action semantics language.
- It is using XMI version 1.0 and UML version 1.4.

Some characteristics of Erlang make the transformation more complex than for other programming languages. In Erlang, data types are not specified, and therefore we used the IDL files for identifying data types. Another problem was that Erlang allows defining methods with the same name, but different number of parameters in a single software module. Although internal coding guidelines recommends using different method names, sometimes programmers have kept these methods to keep the code backward compatible. In these cases we chose the method with higher number of parameters, and recognize that the code should be manually updated.

As mentioned in Section 4, the component framework may be seen as a virtual machine, realized in CORBA and Erlang/OTP. It also includes design rules for application developers that describe how to use its services, and templates for programmers that include operations for using these services in Erlang (and C as well). We mapped each Erlang file to a UML class, and the exported methods in an Erlang file were mapped to public operations in the UML class. However we removed methods that depend on the OTP platform. This removal makes the model platform independent, but the virtual machine looses some of the services that were not described in a technology-neutral way; e.g. services for starting the system and transaction handling.

We recognized the following advantages of raising the level of abstraction by transforming a PSM to another PSM:

- The model is synchronized with the code. Any changes in the code can be automatically mirrored in the model by using the developed tool.
- The UML model may be used to develop the system on other platforms than CORBA or other languages than Erlang. It may also be integrated with other models or be used for future development of applications.
- The model is exchangeable to by using XMI.
- The new UML model may be used during inspections or for developing test cases.

6. Discussion and Conclusions

Ericsson uses Erlang for its good performance and characteristics suitable for concurrent, distributed applications. But Erlang is not in the list of languages supported by commercial MDA tools. However our study confirmed the possibility and low cost of developing a tool that helps to keep the UML models synchronized with the code.

Reverse engineering is a complex task. We described some challenges we met during transforming a PSM to another PSM. Some of them are specific to the Erlang programming language, while an interesting issue was the difficulty to distinguish between aspects of the component framework that are platform-independent (and hence may be realized in other platforms without further changes) and those that are platform dependent, where a platform in this context is OTP. The Translator gives a PSM that is structurally complete, but transformation to a structurally complete PIM should be done manually by developing a model for the component framework that is platform independent.

Another important issue is the difficulty to extract behavior and constraints automatically from the code. We could draw sequence diagrams manually by using the code, but they can’t be used by Rose (or any other tool) to generate code in other programming languages. Therefore we can't develop a computationally complete PIM or PSM.

The next steps in the study may be:

1. Study the possibility to develop a platform independent model for the component framework, and a Platform Description Model (PDM) that describes the framework realization.
2. Study the possibility to extract objects from the developed PIM (in the design model) to have a complete object-oriented class diagram. Neither Erlang nor C is object-oriented languages, while future development may be object-oriented.
3. Develop a similar translator for the C language.

Developing legacy wrappers is another approach when integrating legacy systems, which is not evaluated in this case and may be subject of future studies.

The study helped us to better understand the MDA approach to software development and to identify the problems and opportunities with the approach. Although organizations may find it difficult to use the MDA approach for their legacy systems, some aspects of the approach may already be integrated into their current practice.

Acknowledgement

We thank Ericsson in Grimstad for the opportunity to perform the case study.

References

[1] L. Ekeroth, P.M. Hedstrom, "GPRS Support Nodes", *Ericsson Review*, No. 3, 2000, pp. 156-169.

[2] For more details on Erlang and OTP, see www.erlang.se

[3] The INCO (INcremental and COmponent-based development) project is a Norwegian R&D project in 2001-2004: http://www.ifi.uio.no/~isu/INCO/

[4] ISO, RM-ODP [X.900] http://www.community-ML.org/RM-ODP/

[5] MDA Guide V1.0: http://www.omg.org/docs/omg/03-05-01.pdf

[6] P. Mohagheghi, R. Conradi, "Experiences with Certification of Reusable Components in the GSN Project in Ericsson, Norway", Proc. 4th ICSE Workshop on Component-Based Software Engineering: Component certification and System Prediction, ICSE'2001, Toronto, Canada, May 14-15, 2001, 27-31.

[7] Rational Unified Process: www.rational.com

[8] Selo, Warsun Najib, "MDA and Integration of Legacy Systems", MSc thesis, *Agder University College*, Norway, 2003.

P4. Object-Oriented Reading Techniques for Inspection of UML Models – An Industrial Experiment

Reidar Conradi[1], Parastoo Mohagheghi[2], Tayyaba Arif[1], Lars Christian Hegde[1], Geir Arne Bunde[3], and Anders Pedersen[3]

[1] Department of Computer and Information Science, NTNU, NO-7491 Trondheim, Norway
[2] Ericsson Norway - Grimstad, Postuttak, NO-4898 Grimstad, Norway
[3] Agder University College, NO-4876 Grimstad, Norway
conradi@idi.ntnu.no, parastoo.mohagheghi@eto.ericsson.se

Abstract

Object-oriented design and modeling with UML has become a central part of software development in industry. Software inspections are used to cost-efficiently increase the quality of the developed software by early defect detection and correction. Several models presenting the total system need to be inspected for consistency with each other and with external documents such as requirement specifications. Special Object Oriented Reading Techniques (OORTs) have been developed to help inspectors in the individual reading step of inspection of UML models. The paper describes an experiment performed at Ericsson in Norway to evaluate the cost-efficiency of tailored OORTs in a large-scale software project. The results showed that the OORTs fit well into an incremental development process, and managed to detect defects not found by the existing reading techniques. The study demonstrated the need for further development and empirical assessment of these techniques, and for better integration with industrial work practice.

1. Introduction

The Unified Modeling Language (UML) provides visualization and modeling support, and has its roots in object-oriented concepts and notations [4]. Using UML implies a need for methods targeted at inspecting object-oriented models, e.g. to check consistency within a single model, between different models of a system, and between models and external requirement documents. Detected defects may be inconsistencies, omissions or ambiguities; i.e. any fault or lack that degrades the quality of the model.

Typically software inspections include an individual reading step, where several inspectors read the artifacts alone and record the detected defects. An inspection meeting for discussing, classification and recording defects follows this step. Individual reading of artifacts (the target of this paper) strongly relies on the reader's experience and concentration. To improve the output of the individual reading step, checklists and special reading guidelines are provided. Special Object-Oriented Reading Techniques (OORTs) have been developed at the University of Maryland, USA, consisting of seven individual reading techniques (Sec. 2.2). In each technique, either two UML diagrams are compared, or a diagram is read against a Requirements Description.

Modeling in UML is a central part of software development at Ericsson in Grimstad. With increased use of UML, *review* and *inspection* of UML models are done in all development phases. While reviews are performed to evaluate project status and secure design quality by discussing broader design issues, formal inspections are part of the exit criteria for development phases. In the inspection process in Ericsson, individual inspectors read UML diagrams using different views, with checklists and guidelines provided for each type of view or focus.

Ericsson primarily wants to increase the *cost-efficiency* (number of detected defects per person-hour) of the individual reading step of UML diagrams, since inspection meetings are expensive and require participation of already overloaded staff. Ericsson further wants to see if there is any correlation between developer experience and number of defects caught during individual reading. Lastly, Ericsson wants to improve the relevant reading techniques (old or new) for UML diagrams, and to find out whether the new reading techniques fit into their incremental development process.

Before introducing the OORTs in industry, systematic empirical assessments are needed to evaluate the cost-efficiency and practical utility of the techniques. Following a set of student experiments for assessment and improvement of the techniques at The University of Maryland and NTNU [17][6], we conducted a small controlled experiment at Ericsson. The experiment was performed as part of two diploma (MSc) theses written in spring 2002 at the Agder University College (AUC) and The Norwegian University of Science and Technology (NTNU) [5][1]. The original set of OORTs from The University of Maryland were revised twice by NTNU for understandability, evaluated and re-evaluated on two sample systems, and then tailored to the industrial context.

The Ericsson unit in Norway develops software for large, real-time systems. The Requirements Descriptions and the UML models are big and complex. Besides, the UML models are developed and inspected *incrementally*; i.e. a single diagram may be inspected several times following successive modifications. The size of the inspected artifacts and the incremental nature of the software development process distinguish this industrial experiment from previous student experiments. The cost-efficiency of inspections and the types of detected defects were used as measures of the well-suitedness of the techniques. Other steps of the inspection process, such as the inspection meeting, remained unchanged.

Results of the experiment and qualitative feedback showed that the OORTs fit well into the overall inspection process. Although the OORTs were new for the inspectors, they contributed to finding more defects than the existing reading techniques, while their cost-efficiency was almost the same. However, the new techniques ought to be simplified, and questions or special guidelines should be added.

The remainder of the paper is structured as follows: Section 2 describes some state of the art and the new OORTs. Section 3 outlines the overall empirical approach to assess the OORTs. Section 4 summarizes the existing practice of reviews and inspections at Ericsson and some baseline data. Section 5 describes the experimental steps and results, analyzes the main results, and discusses possible ways to improve the new OORTs and their usage. The paper is concluded in Section 6.

2. The Object-Oriented Reading Techniques (OORTs)

2.1 A Quick State of the Art

Inspection is a technique for early defect detection in software artifacts [8]. It has proved to be effective (finding relatively many defects), efficient (relatively low cost per defect), and practical (easy to carry out). Inspection cannot replace later testing, but many severe defects can be found more cost-efficiently by inspection. A common reading technique is to let inspectors apply complimentary perspectives or views [2][3]. There are over 150 published studies, and some main findings are:

- It is reported a net productivity increase of 30% to 50%, and a net timescale reduction of 10% to 30% [9, p.24].
- Code inspection reduces costs by 39%, and design inspection reduces rework by 44% [11].
- Ericsson in Oslo, Norway has previously calculated a net saving of 20% of the total development effort by inspection of design documents in SDL [7].

As software development becomes increasingly model-based e.g. by using UML, techniques for inspection of models for completeness, correctness and consistency should be developed. Multiple models are developed for complex software systems. These models represent the same system from different views and different levels of abstraction.

However, there exist no documented, industrial-proven reading techniques for UML-based models [16]. The closest is a reported case study from Oracle in Brazil [13]. Its aim was to test the practical feasibility of the OORTs, but there was no company baseline on inspections to compare with. The study showed that the OORTs did work in an industrial setting. Five inspectors found 79 distinct defects (many serious ones), with 2.7 defects/person-hour (totally 29 person-hours, but excluding a final inspection meeting). Few qualitative observations were collected on how the OORTs behaved.

2.2. The OORTs

As mentioned, one effort in adapting reading techniques for the individual reading step of inspections to object-oriented design was made by the OORT-team at University of Maryland, USA [17]. The principal team members were:

- Victor R. Basili and Jeffrey Carver (The University of Maryland),
- Forrest Shull (The Fraunhofer Center – Maryland),
- Guilherme H. Travassos (COPPE/Federal University of Rio de Janeiro).

Special object-oriented reading techniques have been developed since 1998 to inspect (“compare”) UML diagrams with each other and with Requirements Descriptions in order to find defects. *Horizontal reading techniques* are for comparing artifacts from the same development phase such as class diagrams and state diagrams developed in the design phase. Consistency among artifacts is the most important focus here. *Vertical reading techniques* are for comparing artifacts developed in different development phases such as requirements and design. Completeness (traceability of requirements into design) is the focus. UML diagrams may capture either *static* or *dynamic* aspects of the modeled system. The original set of OORTs has seven techniques, as in Figure 1:

OORT-1: Sequence Diagrams vs. Class Diagrams (horizontal, static)
OORT-2: State Diagrams vs. Class Descriptions[1] (horizontal, dynamic)
OORT-3: Sequence Diagrams vs. State Diagrams (horizontal, dynamic)
OORT-4: Class Diagrams vs. Class Descriptions (horizontal, static)
OORT-5: Class Descriptions vs. Requirements Descriptions (vertical, static)
OORT-6: Sequence Diagrams vs. Use Case Diagrams (vertical, static/dynamic)
OORT-7: State Diagrams vs. (Reqmt. Descr.s / Use Cases) (vertical, dynamic)

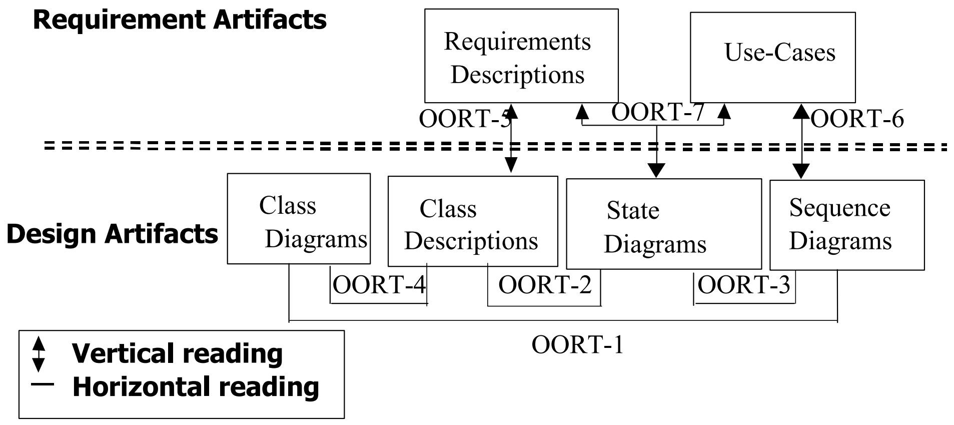


Figure 1 The seven OORTs and their related artifacts, taken from [18]

The techniques cover most diagrams when modeling a system with UML. In addition, Requirements Descriptions are used to verify that the system complies with the prerequisites. Each technique compares at least two artifacts to identify defects in them (but requirements and use cases are assumed to be defect-free here). The techniques consist of several steps with associated questions. Each technique focus the reader on different design aspects related to consistency and completeness, but not on e.g. maintainability and testability. In student experiments, each reader either did four "dynamic" OORTs or four "static" ones, and with OORT-6 in common. That is, we had two complementary *views*, a dynamic and a static one.

Defects detected by the techniques are classified either as Omission (missing item), Extraneous information (should not be in the design), Incorrect fact (misrepresentation of a concept), Ambiguity (unclear concept), Inconsistency (disagreement between representations of a concept), or Miscellaneous (any other defects). In [18], severity of defects may be either Serious (It is not possible to continue reading. It needs redesign), Invalidates (the defects invalidates this part of the document) or Not serious (needs to be checked).

[1] Class Descriptions include textual descriptions of goals and responsibilities of a class, list of functions with descriptions of each function, attributes, cardinalities, inheritance, and relations.

To get more familiar with the techniques, a short description of OORT-1 is given in the following: The goal of this technique is to verify that the Class Diagram for the system describes classes and their relationships consistently with the behaviors specified in the Sequence Diagrams. The first step is to identify all objects, services and conditions in the Sequence Diagram and underline them in different colors. The second step is to read the related Class Diagram and see whether all objects are covered, messages and services found, and constraints fulfilled. To help the reader, a set of questions is developed for each step.

3. The Overall Empirical Method

Developing a method solid enough to be used in the industry takes time and effort through various experiments and verification of results. A set of empirical studies at University of Maryland and NTNU has used the empirical method presented in [14] for improving a development process from the conceptual phase to industry. The method is divided into four studies where each study step has some questions that need to be answered before the next level can be reached:

1. Feasibility study -- Did the process provide usable and cost-effective results?
2. Observational study -- Did the steps of the process make sense?
3. Case study: Use in real life cycle -- Did process fit into the lifecycle?
4. Case study: Use in industry -- Did process fit into industrial setting?

Previous studies at The University of Maryland have emphasized steps 1-3, using students. There is also an undocumented student study from University of Southern California, where the OORTs were tailored to the Spiral Model, i.e. step 3. Previous student experiments at NTNU [6] have applied steps 1 and 2.

The mentioned case study at Oracle in Brazil was the *first* industrial study, emphasizing step 4 and feasibility. It applied more or less the original version of the OORTs, i.e. with no tailoring to the industrial context. Regrettably, we were not aware of this study before our experiment.

The study at Ericsson was the *second* industrial study, with emphasis on step 4 and with a direct comparison of Ericsson's existing inspection techniques. It used a revised and tailored version of the OORTs. We will call it an *experimen*t and not a case study, as it was very close to a controlled experiment.

4. The Company Context

The goal of the software development unit at Ericsson in Grimstad, Norway is to build robust, highly available and distributed systems for large, real-time applications, such as GPRS and UMTS networks. SDL and the proprietary PLEX languages have recently been replaced by UML and e.g. Java or C++. UML models are developed to help understanding the structure and behavior of the system, for communicating decisions among stakeholders, and finally to generate code to some extent [10].

The Ericsson inspectors are team members working on the same software system. They have extensive experience with and good motivation for inspections. The artifacts in the student experiments represented complete, although small systems. In contrast, Ericsson's UML models are developed incrementally and updated in each delivery with new or changed requirements. I.e., diagrams are inspected in increments when any complete revision is done. The artifacts at Ericsson are also of industrial caliber:

- The Requirements Descriptions are in many cases large and complex, including external telecommunication standards, internal requirement specifications, and/or change requests.
- The inspected UML diagrams are often huge, containing many classes, relationships or messages - indeed covering entire walls!

4.1. State of the Practice of Reviews and Inspections

Ericsson has a long history in inspecting their software artifacts; both design documents and source code. The inspection method at Ericsson is based on techniques originally developed by Fagan [8], later refined by Gilb [9], adapted for Ericsson with Gilb's cooperation, and finally tailored by the local development department. Below, we describe the existing Ericsson review and inspection process for UML diagrams.

A review is a team activity to evaluate software artifacts or project status. Reviews can have different degrees of formality; i.e. from *informal meetings* (to present the artifacts) and *walkthroughs* (to discuss design issues and whether the design meets the requirements) to *frequent reviews* (more formal intermediate checks for completeness and correctness). Reviews act as internal milestones in a development phase, while formal *inspections* are performed at the end of an activity and act as exit criteria.

Each inspection has an associated team. The team consists of a moderator, several inspectors, at least one author, and possibly a secretary. For optimal performance, Ericsson guidelines state that a team should consist of 5 to 7 persons. The moderator is in charge of planning and initiating the inspection process. He chooses the artifacts to be inspected (with incremental development also their versions), and assigns inspectors to different views (see below). Before the inspection meeting, inspectors individually read the artifacts and mark the defects, usually directly in the inspected artifact. Requirements Descriptions, UML diagrams and source code are usually printed out for easy mark-up. If a diagram is too large to be printed out, the inspector takes separate notes on the defects and related questions.

Ericsson uses *views* during inspections, where a view means to look at the inspected artifact with a special focus in mind. Examples are *requirement* (whether a design artifact is consistent with requirements), *modeling guideline* (consistency with such guidelines), or *testability* (is the modeled information testable?). For each view, the inspectors apply checklists or design rules to help discovering defects.

An example of a modelling guideline is: The interface class will be shown as an icon (the so-called "lollipop") and the connection to the corresponding subsystem, block or unit proxy class shall be "realize" and not "generalize". An example of a design rule is: A call back interface (inherited from an abstract interface) shall be defined on the block or subsystem level (visibility of the interface). Such guidelines and rules enforce that the design model will contain correct interfaces to generate IDL files.

Only two different classifications for severity of defects are used, Major and Minor. A Major defect (most common) will cause implementation error, and its correction cost will increase in later development phases. Examples include incorrect specifications or wrong function input. A Minor defect does not lead to implementation error, and is assumed to have the same correction cost throughout the whole process. Examples are misspelling, comments, or too much detail.

In spite of a well-defined inspection process and motivated developers, Ericsson acknowledges that the individual reading step needs improvement. For instance, UML orientation is poor, and inspectors spend too little time in preparatory reading - i.e. poor process conformance, see below.

4.2. Inspection Baseline at Ericsson

A post-mortem study of data from inspections and testing was done at the Ericsson development unit outside Oslo, Norway in 1998 [7]. The historical data used in this study is from the period from 1993 to 1998, and also covered data for code reviews and different test activities (unit test, function test, and system test). The results confirm that individual design reading and code reviews are the most cost-efficient (economical) techniques to detect defects, while system tests are the least cost-efficient.

While the cost-efficiency of inspections is reported in many studies, there is no solid historical data on inspection of UML diagrams, neither in the literature nor at Ericsson. As part of a diploma thesis at AUC, data from 38 design and code inspections between May 2001 and March 2002 were analyzed; but note that:

- Design (UML) and code inspections were *not* distinguished in the recorded data.
- In the first 32 inspections logs, only the *total* number of defects was reported, covering both individual reading and inspection meetings. Only the last 6 inspections had distinct data here.

Table 1 Ericsson baseline results, combined for design and code inspections

	%Effort Individual Reading	**%Effort Meeting**	**Overall Efficiency (def./ph)**	**Individual Reading Efficiency (def./ph)**	**Meeting Efficiency (def./ph)**
All 38 inspections	32	68	0.53	-	-
6 last inspections	24	76	1.4	4.7	0.4

The data showed that most of the effort is spent in inspection meetings, while individual reading is more cost-efficient. For the 6 last inspections:

- 24% of the effort is spent in individual reading, finding 80% of the defects. Inspection meetings took 76% of the effort but detected 20% of defects. Thus, individual reading is 12 times more cost-efficient than inspection meetings.
- Two of these inspections had an extra high number of defects found in individual reading. Even when this data is excluded, the cost-efficiency is 1.9 defects/person-hour for individual reading and 0.6 defects/person-hour for meetings, or a factor 3.

There has been much debate on the effect of inspection meetings. Votta reports that only 8% of the defects were found in such meetings [19]. The data set in this study is too small to draw conclusions, but is otherwise in line with the cited finding.

5. Ericsson Experiment and Results

The experiment was executed in the context of a large, real software project and with professional staff. Conducting an experiment in industry involves risks such as:

- The experiment might be assumed as time-consuming for the project, causing delay and hence being rejected. Good planning and preparation was necessary to minimize the effort spent by Ericsson staff. However, the industrial reality at Ericsson is very hectic, and pre-planning of all details was not feasible.
- The time schedule for the experiment had to be coordinated with the internal inspection plan. In fact, the experiment was delayed for almost one month.
- Selecting the object of study: The inspected diagrams should not be too complex or too trivial for running the experiment. The inspected artifacts should also contain most of the diagrams covered by the techniques.

PROFIT - PROcess improvement For IT industry – is a cooperative, Norwegian software process improvement project in 2000-2002 where NTNU participates. This project is interfaced with international networks on empirical software engineering such as ESERNET and ISERN. For the experiment at Ericsson, PROFIT was the funding backbone.

The OORTs had to be modified and verified before they could be used at Ericsson. Therefore the NTNU-team revised the techniques in two steps:

1. Comments were added and questions rephrased and simplified to improve understandability by making them more concise. The results in [1] contain concrete defect reports, as well as qualitative comments and observations.
2. The set of improved techniques were further modified to fit the company context. These changes are described in Section 5.2.

Students experienced that the OORTs were cost-efficient in detecting design defects for two sample systems, as the OORTs are very structured and offer a step-by-step process. On the other hand, the techniques were quite time-consuming to perform. Frustration and de-motivation can easily be the result of extensive methods. In addition, they experienced some redundancy between the techniques. Particularly OORT-5 and OORT-6 were not motivating to use. A lot of issues in OORT-5 and OORT-6 were also covered by OORT-1 and OORT-4. OORTs-6/7 were not very productive either.

The experiment was otherwise according to Wohlin's book [20], except that we do not negate the null hypotheses. The rest of this section describes planning and operation, results, and final analysis and comments.

5.1. Planning

Objectives: The inspection experiment had four industrial objectives, named **O1-O4**:

- **O1 – analyze cost-efficiency and number of detected defects,** with **null hypothesis *H0a:*** *The new reading techniques are as cost-efficient and help to find at least as many defects as the old R&I techniques*. ("Effectiveness", or fraction of defects found in inspections compared to all reported defects, was not investigated.).

- **O2 – analyze the effect of developer experience,** with **null hypothesis** ***H0b:*** *Developer experience will positively impact the number of detected defects in individual reading.*
- **O3 – help to improve old and new reading techniques for UML,** since Ericsson's inspection guidelines had not been properly updated after the shift in design language from SDL to UML. No formal hypothesis was stated here, and results and arguments are mostly qualitative.
- **O4 – investigate if the new reading techniques fit the incremental development process at Ericsson.** Again, qualitative arguments were applied.

Relevant inspection data: To test the two null hypotheses *H0a* and *H0b*, the *independent* variable was the individual reading technique with two treatments: either the existing review and inspection techniques (R&I) or the OORTs modified for the experiment. The *dependent* variables were the effort spent, and the number and type of detected defects in the individual reading step and in the inspection meetings (see below on defect logs). Data in a questionnaire (from the OORT-team at Maryland) over developer experience was used as a *context* variable. To help to evaluate objectives *O3* and *O4*, all these variables were supplemented with qualitative data from defect logs (e.g. comments on how the OORTs behaved), as well as data from observation and interviews.

Subjects and grouping: Subjects were the staff of the development team working with the selected use case. They were comprised of 10 developers divided in two groups, the *R&I-group* applying the previous techniques and the *OORT-group* applying the new ones. A common moderator assigned the developers to each group. A slight bias was given to implementation experience in this assignment, since Ericsson wanted all the needed views covered in the R&I-group (see however Figure 2 in 5.2). The R&I-group then consisted of three very experienced designers and programmers, one newcomer, and one with average experience. The OORT-group consisted of one team leader with good general knowledge, two senior system architects, and two with average implementation knowledge. Inspection meetings were held as usual, chaired by the same moderator. Since both groups had 5 individuals, the experimental design was balanced. Both groups had access to the same artifacts.

Changes to the OORTs: As mentioned, the OORTs were modified to fit Ericsson's models and documents, but only so that the techniques were comparable to the original ones and had the same goals. The main changes were:

- **Use Case Specifications:** Each use case has a large textual document attached to it, called a Use Case Specification (UCS), including Use Case Diagrams and the main and alternative flows. This UCS was used instead of the graphical Use Case Diagram in OORT-6 and OORT-7.
- **Class Descriptions:** There is no explicit Class Description document, but such descriptions are written directly in the Class Diagrams. In OORT-2, OORT-4 and OORT-5, these textual class descriptions in the Class Diagrams are used.
- **OORT-4:** Class Diagram (CD) vs. Class Description (CDe). The main focus of this technique is the consistency between CD and CDe. As Class Descriptions are written in the same Class Diagram, this technique seems unnecessary.

However, the questions make the reader focus on internal consistency in the CD. Therefore all aspects concerning Class Descriptions were removed and the technique was renamed to "*Class Diagram for internal consistency*".

- **OORT-5:** Class Description (CDe) vs. Requirements Descriptions (RD). Here, the RD is used to identify classes, their behaviors and necessary attributes. That is, the RD nouns are candidates for classes, the RD verbs for behaviors, and so on. The technique was not applicable in Ericsson, due to the large amount of text that should be read. But Ericsson has an iterative development process, where they inspect a small part of the system at one time. The UCS could substitute the RD for a particular part of the system, but the focus of the specification and the level of abstraction demanded major changes in the technique, which would make the technique unrecognizable. Therefore a decision was made to *remove OORT-5.* Thus, we had *six OORTs* to try out.

Defect Logging: To log defects in a consistent and orderly manner, one template was made for the R&I-group and a similar one for the OORT-group – both implemented by spreadsheets. For all defects, the inspectors registered an explanatory name, the associated artifact, the defect type (Omission, Extraneous etc.), and some detailed comments. The OORT-group also registered the technique that helped them to find the defect. Ericsson's categorization of Major and Minor was not applied (we regretted this during later analysis). These changes in defect reporting were the only process modification for the R&I-group. The amount of effort spent by each inspector, in individual reading and inspection meetings, was also recorded for both groups. We also asked for qualitative comments on how the techniques behaved.

5.2. Operation, Quantitative Results, and Short Comments

It was decided to run the experiment in April or May 2002, during an already planned inspection of UML diagrams for a certain use case, representing the next release of a software system. The inspected artifacts were:

- Use Case Specification (UCS) of 43 pages, including large, referenced standards.
- Class Diagram (CD), with 5 classes and 20 interfaces.
- Two Sequence Diagrams (SqD), each with ca. 20 classes and 50 messages.
- One State Diagram (StD), with 6 states including start and stop, cf. below.

Problem note 1: When the actual use case and its design artifacts were being prepared for the experiment, a small but urgent problem occurred: For this concrete use case (system) there was *no State Diagram (StD)!* Such diagrams are normally made, but not emphasized since no code is generated from these. Luckily, the UCS contained an Activity Diagram that was a hybrid of a StD and a data flow chart. Thus, to be able to use the OORTs in their proposed form, a StD was hastily extracted from this Activity Diagram. However, the StD was now made in the analysis and not in the design phase, so the reading in OORT-7 changed focus. The alternative would have been to drop the three OORTs involving State Diagrams, leaving us with only three OORTs. The R&I-group had access to, but did not inspect this StD.

The experiment was executed over two days. In the beginning of the first day, the NTNU students gave a presentation of the experimental context and setup. For the OORT-group, a short introduction to the techniques was given as well. Since we had few inspectors, they were told to use *all the available six OORTs (excluding OORT-5)*, not just four "dynamic" ones or four "static" ones as in previous experiments (again, we regretted this later).

Each participant filled out a questionnaire about his/her background (e.g. number of projects and experience with UML). The R&I-group was not given any information on the OORTs. The 10 participants in this experiment were the team assigned to the use case, so they had thorough knowledge of the domain and the UML models at hand.

When all participants had finished their individual reading, they met in their assigned teams for normal inspection meetings. During these meetings, each defect was discussed and categorized, and the moderator logged possible new defects found in the meetings as well. At the end of the meetings, a short discussion was held on the usability of the techniques and to generally comment on the experiment.

Problem note 2: One inspector in the OORT-group did only deliver his questionnaire, not his defect log. Thus the OORT-data represents 4, not 5 persons. The number of defects from the OORT group is therefore lower than expected (but still high), while the OORT effort and cost-efficiency data reflect the reduced person-hours.

Table 2 Summary of collected data on defects from the Ericsson experiment

	Indiv. read. defects	Meet. defects	Over -laps	% Indiv. read. defects	% Meet. defects	Person -hours Indiv. read.	Person-hours Meet.
R&I-group	17	8	0	68	32	10	8.25
OORT-group	38	1	8	97	3	21.5	9

Table 2 shows the number of distinctive defects found in individual reading and inspection meetings, both as absolute numbers and relative frequencies. It also shows the effort in person-hours for individual reading and meetings. Defects reported in more than one defect log are called *overlaps* (in column four), and 8 "overlap defects" were reported for the OORT-group.

The cost-efficiency (defects/person-hours) of the individual reading step, the inspection meetings and the average for both groups is shown in Table 3 below.

Table 3 Cost-efficiency of inspections as no. of detected defects per person-hour

	Cost-eff. Indiv.read. (defects/ph)	Cost-eff. Meeting (defects/ph)	Cost-eff. Average. (defects/ph)
R&I-group	1.70	0.97	1.37
OORT-group	1.76	0.11	1.28

Defects logs were used to make a summary of the distribution of defects over the defined defect types. Table 4 shows that the R&I-group registered most Incorrect fact, while the OORT-group found most Omission and Inconsistency.

Table 4 Defect Distribution on defect types

Defect Type	R&I-group Indiv.read.	R&I-group Meeting	OORT-group Indiv.read.	OORT-group Meeting
Omission	3	2	12	1
Extraneous	-	3	6	-
Incorrect fact	10	3	1	-
Ambiguity	-	-	5	-
Inconsistency	2	-	12	-
Miscellaneous	2	-	2	-
Total	17	8	38	1

Short comment: Incorrect facts reported by the R&I-group were mostly detected in the two Sequence Diagrams showing the interactions to realize the use case behavior. These defects were *misuse of a class or interface*, such as wrong order of operation calls or calling the wrong operation in an interface (Incorrect fact was originally defined as misrepresentation of a concept). The group argued that the interface is misrepresented in the Sequence Diagram, and thus the defects are of type Incorrect fact.

For the OORT-group the defects were also classified based on the question leading to find them. OORT-1 and OORT-2 helped finding most defects. OORT-7 did not lead to detection of any defects whatsoever.

Problem note 3: The inspectors mentioned that some defects were detected by more than one technique and only registered the first technique that lead to them. However, the techniques were time-consuming, and one of the developers did not do OORT-6 and OORT-7, while others used little time on these latter two.

As mentioned, the participants filled in a questionnaire where they evaluated their *experience* on different areas of software development on an ordinal scale from 0 to 5, where 5 was best. A total score was coarsely calculated for each participant by simply adding these numbers. The maximum score for 20 questions was 100. Figure 2 shows the number of defects reported by each participant and their personal score for 9 participants (data from the "misbehaving" fifth participant in the OORT-group was not included). The median and mean of these scores were very similar within and between the two groups, so the groups seem well balanced when it comes to experience. For the R&I-group, the number of reported defects increases with their personal score, while there is no clear trend for the OORT- group!

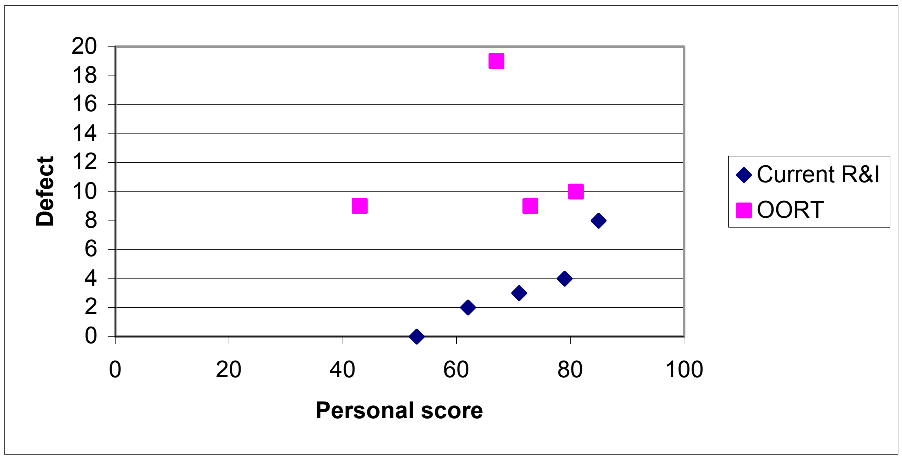


Figure 2 Relationship between numbers of defects reported in individual reading and personal scores ("experience") for 9 participants

5.3. Further comments, Interpretation and Analysis

Here, we first comment deeper on some of the results, also using qualitative feedbacks. Then we assess the objectives and hypotheses, and lastly analyze the validity threats. A general reminder is that the data material is very meager, so any conclusion or observation must be drawn with great care.

Comments on old vs. new reading techniques: All in all, the R&I-group only found 68% of their defects in individual reading. This is considerably less then the 98% of the defects found by the OORT-group in this step. The meeting was less prosperous for the latter group, which is the expected result. The R&I-group inversely detected 32% of their defects in the inspection meeting, which is high but not cost-efficient. However, the OORT-group spent twice the effort on individual reading, and therefore the cost-efficiency is almost the same. Furthermore, the OORTs were new for the inspectors, and this may hurt cost-efficiency.

The OORT-group found much more Omissions and Inconsistencies than the R&I-group. The OORTs are based on comparing UML diagrams with each other and with requirements, and this may result in finding many more Omissions and Inconsistencies. In contrast, the R&I techniques do not guide inspectors to find "defects", which do not degrade the behavior of the system. An example is possible Inconsistencies between a Class Diagram and a Sequence Diagram (in OORT-1), since no code is generated from a Sequence Diagram during design. However, Inconsistencies in other artifacts related to the State Diagram (as in OORT-2 and OORT-3) are important also for implementation.

The R&I-group detected 10 defects of type Incorrect fact, all being important for implementation, while the OORT-group detected only *one* such defect. The registered defects included both misrepresentation of concepts and misuse of them, such as interface misuse being commented for Figure 4. Finding Incorrect facts may be based on previous knowledge of the system, and inspectors in the R&I-group had better insight in implementation details. Another reason is, that for the inspected system, internal design guidelines and Class Descriptions contain information on the use of interfaces. Comparing these with the Sequence Diagrams may have helped finding violations to interface specifications, such as wrong order of operation calls. This technique is not currently in the set of OORTs, while the R&I techniques ask for conformance to such design guidelines.

One interesting result of the experiment was the total *lack of overlap* between defects found by the two groups. The N-fold inspection method [12] is based on the hypothesis that inspection by a single team is hardly effective and N independent teams should inspect an artifact. The value of N depends on many factors such as cost of additional inspections and the potential expense of letting a defect slip by undetected. Our results showed that each team only detected a fraction of defects as anticipated by the above method. This result is possibly affected by a compound effect of the two elements discussed earlier as well: slightly different background of inspectors and different focus of reading techniques. The latter meant, that the OORTs focused on consistency between UML diagrams and completeness versus requirements, while the R&I techniques focused on conformance to the internal guidelines. The experiment therefore suggests concrete improvements in the existing R&I techniques.

Lastly, defect severity (e.g. Major, Minor, and possibly Comment or as defined by the OORTs) should be included for both techniques. Defect types might also be made more precise – e.g. to distinguish Interface error, Sequencing error etc.

Comments on the new reading techniques: Some OORTs helped to detect more defects than others. The inspectors mentioned that some defects were found by more than one technique, and were therefore registered only once for the first OORT. Such redundancies should be removed.

Some UML diagrams of the inspected system contain "more" information than others. Modeling is also done differently than assumed in the original set of OORTs - cf. the "Ericsson" changes to OORT-4 and removal of OORT-5.

As mentioned, for the inspected system we had to improvise a State Diagram from an Activity Diagram already standing in the Use Case Specification. But again, making an explicit and separate State Diagram proved that the new OORTs really work: 16(!) defects were totally identified using OORT-2 and OORT-3, comparing the State Diagram with, respectively, Class Descriptions and Sequence Diagrams.

The participants in the OORT-group said it was too *time-consuming* for each to cover all the OORTs, and some (often the last) techniques will suffer from lack of attention. A possible solution is to assign only a subset of the techniques to each participant, similarly to Ericsson's *views* and to what was done in earlier student experiments. A more *advanced UML editor* might also catch many trivial inconsistencies, e.g. undefined or misspelled names, thus relieving human inspectors from lengthy and boring checks.

Finally, we should *tailor* the reading techniques to the context, i.e. project. For instance, the OORTs were successful in detecting Omissions and Inconsistencies by comparing UML diagrams with each other and with the requirements. But they did not detect e.g. misuse of interfaces and inconsistencies between the models and the internal guidelines. A natural solution is to include questions related to *internal guidelines and design rules*, and then e.g. compare Sequence Diagrams with class and interface descriptions as part of a revised OORT-1.

Evaluation of O1/H0a – cost-efficiency and number of defects: Our small sample prevents use of standard statistical tests, but we can anyhow assess *H0a* (and *H0b* below). The cost-efficiency of the old and new techniques seems rather similar, and in line with that of the baseline. The OORTs seem to help finding more defects in the individual reading step than the R&I techniques, respectively 38 and 17 defects. Even without defects (indirectly) related to the new State Diagram, 22 defects were reported using the OORTs. Thus the null hypothesis *H0a should be accepted*.

Evaluation of O2/H0b – effect of developer experience on number of defects: From Figure 2 we see that the number of reported defects from the individual reading step increases with the personal score for the R&I-group. This may indicate that the R&I techniques rely on the experience of the participants. But there is no clear relationship for the OORT-group. Thus the null hypothesis *H0b should be accepted for the R&I-group,* but we cannot say anything for the OORT-group. The effect for the OORT-group is surprising, but consistent with data from The University of Maryland and NTNU [6], and will be documented in Jeffrey Carver's forthcoming PhD thesis.

Evaluation of O3 – improvement of reading techniques for UML: The new OORTs helped Ericsson to detect many *defects not found* by their *existing* R&I techniques. However, both the old and new reading techniques varied a lot in their effectiveness to detect defects among different diagrams and diagram types. This information should be used to improve both sets of reading techniques. Actually, there were many comments on how to improve the OORTs, suggesting that they should be shortened and simplified, have mutual redundancies removed, or include references to internal design guidelines and rules. Thus, although the original set of OORTs had been revised by NTNU in several steps and then tailored for Ericsson, the experiment suggests further simplification, refinement, and tailoring.

Evaluation of O4 – will fit in the incremental development process: Although the OORTs were originally created to inspect entire systems, they work well for an *incremental* development process too. The techniques helped to systematically find inconsistencies between new or updated UML diagrams and between these diagrams and possibly changed requirements. That is, they helped inspectors to see the revised design model as a whole.

Validity Evaluation: Threats to experimental validity are classified and elaborated in [15] [20]. Threats to validity in this experiment were identified to be:

- *Internal validity*: There could be some compensatory rivalry; i.e. the R&I-group could put some extra effort in the inspection because of the experiment. Inversely, the OORT-group may do similar in a "Hawthorne" effect. Due to time/scheduling constraints, some participants in the OORT-group did not cover all the techniques properly, e.g. OORT-6 and OORT-7.
- *External validity*: It is difficult to generalize the results of the experiment to other projects or even to other companies, as the experiment was done on a single use case. Another threat was that the OORTs were adapted for Ericsson, but we tried to keep the techniques as close to the original set as possible.
- *Construct validity*: The OORT-group had knowledge of the R&I techniques and the result for them could be a mix of using both techniques.
- *Conclusion validity*: The experiment is done on a single use case and it is difficult to conclude a statistical relationship between treatment and outcome. To be able to utilize all the techniques, a simple State Diagram was extracted the day before the experiment. The R&I-group did not look at this particular diagram, while the OORT-group reported 16 defects related to this diagram and to indirectly related artifacts. The inspectors were assigned "semi-randomly" to the two groups, which roughly possessed similar experience. The adding of

ordinal scores to represent overall inspector experience is dubious, but this total score was only used qualitatively (i.e. is there a trend? - not how large it is).

6. Conclusions

The studied Ericsson unit incrementally develops software for large-scale real-time system. The inspected artifacts, i.e. Requirements Descriptions and UML models, are substantially larger and more complex than those used in previous academic experiments. For Ericsson it is interesting to see if these techniques could be tailored to their inspection needs in the individual reading step.

Below we sum up the objectives of the experiment and how they have been reached:

- **O1 and H0a – cost-efficiency and detected defects:** The cost-efficiency of the old R&I techniques and the new OORTs seems very similar. The new ones helped to find more than twice as many defects as the old ones, but with no overlaps with the defects found by the old techniques.
- **O2 and H0b – effect of developer experience on detected defects:** There is probably a positive trend for the old R&I techniques, but we do not know for the new ones. The result may term "expected", but the reasons are not quite understood.
- **O3 - improvement of old and new reading techniques:** Although the new OORTs have shown promising results, the experiment suggests further modifications of both general and specific issues. We have for both the old and the new reading techniques identified parts that could be included in the other.
- **O4 – fit into an incremental process:** To our surprise this went very well for the OORTs, although little attention and minimal effort was spent on this.

To conclude: In spite of very sparse data, the experiment showed a need for several concrete *improvements*, and provided many unforeseen and valuable *insights*. We also should expect a learning effect, both for the reading techniques and for Ericsson's inspection process and developers, as a result of more OORT trials. We further think that the evaluation process and many of the experimental results can be *reused* in future studies of inspections of object-oriented design artifacts in UML.

Some final challenges: First, how to utilize inspection data actively in a company to improve their inspection process? Second, how to convince the object-oriented community at large, with its strong emphasis on prototyping and short cycle time, to adopt more classic quality techniques such as inspections?

Acknowledgements

We thank Ericsson in Grimstad for the opportunity and help to perform the experiment with ten of their designers and several managers, who all were highly motivated. We also thank the original OORT-team in USA for inspiration and comments. The study was partially funded by two public Norwegian research projects, namely PROFIT (Sec. 5) and INCO (INcremental and COmponent-based development, done jointly by University of Oslo and NTNU). Thanks also goes to local colleagues at NTNU.

References

1. Arif, T., Hegde, L.C.: Inspection of Object-Oriented Construction. Diploma (MSc) thesis at NTNU, June 2002. See http://www.idi.ntnu.no/grupper/su/su-diploma-2002/Arif-OORT_Thesis-external.pdf.
2. Basili, V.R., Caldiera, G., Lanubile, F., and Shull, F.: Studies on reading techniques. Proc. Twenty-First Annual Software Engineering Workshop, *NASA-SEL-96-002*, pp. 59-65, Greenbelt, MD, Dec. 1996.
3. Basili, V.R., Green S., Laitenberger, O., Lanubile, F., Shull, F., Sørumgård, S., Zelkowitz, M. V.: The Empirical Investigation of Perspective-Based Reading, *Empirical Software Engineering Journal,* 1(2):133-164, 1996.
4. Booch, G., Rumbaugh, J., Jacobson, I.: *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.
5. Bunde, G.A., Pedersen, A.: Defect Reduction by Improving Inspection of UML Diagrams in the GPRS Project. Diploma (MSc) thesis at Agder University College, June 2002. See http://siving.hia.no/ikt02/ikt6400/g08/.
6. Conradi, R.: Preliminary NTNU Report of the OO Reading Techniques (OORT) exercise in course 7038 on Programming Quality and Process Improvement, spring 2000, v1.12. Oct. 2001, 80 p.
7. Conradi, R., Marjara, A., Hantho, Ø., Frotveit, T., Skåtevik, B.: A study of inspections and testing at Ericsson, Norway. *Proc.* PROFES'99, 22-24 June 1999, pp. 263-284, *published by VTT.*
8. Fagan, M. E.: Design and Code Inspection to Reduce Errors in Program Development. *IBM Systems Journal,* 15 (3):182-211, 1976.
9. Gilb, T., Graham, D.: Software Inspection. Addison-Wesley, 1993.
10. Jacobson, I., Christerson, M., Jonsson, P., Övergaard, G.: *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, revised printing, 1995.
11. Laitenberger, O., Atkinson, C.: Generalized Perspective-Based Inspection to handle Object-Oriented Development Artifacts. Proc. ICSE'99, Aug. 1999, *IEEE CS-Press*, pp. 494-503.
12. Martin, J., Tsai, W.T.: N-fold Inspection: A Requirements Analysis Technique. *Communications of the ACM*, 33(2): 225-232, 1990.
13. Melo, W., Shull, F., Travassos, G.H.: Software Review Guidelines, *Technical Report ES-556/01*, Aug. 2001, 22 p. Systems Engineering and Computer Science Department, COPPE/UFRJ, http://www.cos.ufrj.br (shortly reporting OORT case study at Oracle in Brazil).
14. Shull, F., Carver, J., Travassos, G.H.: An Empirical Method for Introducing Software Process. Proc. European Software Engineering Conference 2001 (ESEC'2001), Vienna, 10-14 Sept. 2001, *ACM/IEEE CS Press*, ACM Order no. 594010, ISBN 1-58113-390-1, pp. 288-296.
15. Sommerville, I.: *Software Engineering*. Addison-Wesley, 6th ed., 2001.

16. Travassos, G.H., Shull F., Carver J., Basili V.R.: Reading Techniques for OO Design Inspections, Proc. Twenty-Forth Annual Software Engineering Workshop, *NASA-SEL*, Greenbelt, MD, Dec. 1999, http://sel.gsfc.nasa.gov/website/sew/1999/program.html.

17. Travassos, G.H., Shull, F., Fredericks, M., Basili, V.R.: Detecting Defects in Object-Oriented Designs: Using Reading Techniques to Increase Software Quality. Proc. OOPSLA'99, p. 47-56, Denver, 1-5 Nov. 1999. In *ACM SIGPLAN Notices 34(10)*, Oct. 1999.

18. Travassos, G.H., Shull, F., Carver, J., Basili, V.R.: Reading Techniques for OO Design Inspections. *University of Maryland Technical Report CS-TR-4353*. April 2002 (OORT version 3), http://www.cs.umd.edu/Library/TRs/CS-TR-4353/CS-TR-4353.pdf.

19. Votta, L.G.: Does Every Inspection Need a Meeting? Proc. ACM SIGSOFT'93 Symposium on Foundation of Software Engineering (FSE'93), p 107-114, *ACM Press*, 1993.

20. Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A.: *Experimentation in Software Engineering, an Introduction*. Kluwer Academic Publishers, 2000.

P5. Using Empirical Studies to Assess Software Development Approaches and Measurement Programs

Parastoo Mohagheghi[1,2,3], Reidar Conradi[2,3]

[1]Ericsson Norway-Grimstad, Postuttak, NO-4898 Grimstad, Norway
[2] Department of Computer and Information Science, NTNU, NO-7491 Trondheim, Norway
[3] Simula Research Laboratory, P.O.BOX 134, NO-1325 Lysaker, Norway
parastoo.mohagheghi@ericsson.com
conradi@idi.ntnu.no

Abstract

In recent years, incremental and component-based software development approaches, and reuse have been proposed to reduce development time and effort, and to increase software quality. The activities in each increment of an incremental approach, and the interaction between incremental and component-based development is presented in the paper using an industrial example of a large-scale telecommunication system. The paper discusses difficulties in gathering data, since data from increments flow into each other, and the degree of change is high. Empirical studies can be useful to assess the approach to software development, and the quality of measurement programs. Establishing relationships between the development approach (incremental, component-based and reuse) and variables such as planning precision, modification rate, or reliability is the goal of our empirical study. The paper presents examples of metrics that are especially useful for such studies, and proposes improvements to the methods and tools for collecting data.

1. Introduction

The main reason for performing empirical studies in software engineering is to gather useful and valid results in order to understand, control, predict, and improve software development. A spectrum of empirical techniques is available, e.g. formal experiments, case studies, interviews, and retrospective analysis, even literature studies. In recent years, incremental and component-based software development, and reuse have been proposed to reduce development time and effort, and to increase software quality (especially usability and reliability). These approaches can be used separately or combined. However, we need empirical evidence in terms of e.g. increased productivity, higher reliability, or lower modification rate to accept the benefits of these approaches.

Ericsson in Grimstad-Norway started using the Rational Unified Process (RUP), an incremental, use-case driven software process, adaptable to different contexts, for developing two large-scale telecommunication systems in 2000. The developed systems are component-based, using an internally developed component framework, and have a high degree of reuse. We have performed several studies at Ericsson in 2001-2003. In this paper we use results of these studies to discuss how empirical studies can be useful to assess development approaches and measurement programs. We give examples on how development approaches have affected quality attributes, and what metrics are especially useful for assessing these approaches.

The remainder of this paper is organized as follows. Section 2 is a brief state of the art. Section 3 presents the Ericsson context and studies. We describe how empirical studies are useful in assessing development approaches in Section 4, and Section 5 discusses the impact of development approaches on measurement programs. The paper is concluded in Section 6.

2. A Brief State-of-the-Art

Iterative and Incremental development has been proposed as an efficient and pragmatic way to reduce risks from new technology, and from imprecise or changing requirements [4]. An increment contains all the elements of a normal software development project, and delivers a (possibly pre-planned) partially complete version of the final system. There is a confusion of terminology in this area (iterative, time-boxing, short interval scheduling etc.), or as we call it in the paper-*incremental development*.

Component-based Software Engineering (CBSE) involves designing and implementing software components, and assembling systems from pre-built components. Components are often developed based on a *component model*, which defines the standards and conventions for component developers [9]. Implementation of such a component model for providing run-time services for components is usually called a *component framework*. CBSE seems to be an effective way to reuse, since components are designed to be units of distribution. However, reuse covers almost any artifact developed in a software life cycle, including the software development process itself. Product lines are especially built around reuse of software architecture.

The basic idea with components is that the user only needs to know the component interface, and not the internal design. This property allows separating component interface design, and component internal design. Karlsson describes two alternatives for assigning functionality to increments in [11]: Features (or user functionality), and system functionality (like start, restart, traffic handling, etc). With CBSE a third alternative would be to have a component-oriented approach; i.e. either assigning components to increments, or designing interfaces of some components in an increment, and implementing them in another increment. KobrA [2] is an example of such process. The component-oriented approach can be combined with the other two, e.g. it can be combined with feature increments if functionality of a feature is too large for an increment.

Incremental development, CBSE, reuse and product-line development have all been in use for a while, and there are increasing number of studies that assess these development approaches by correlating the specific approach to attributes of software quality, such as reliability (e.g. in terms of defect-density), maintainability (e.g. in terms of maintenance effort), productivity (e.g. in terms of line of code per person-hour), delivery precision etc. See for example [12, 13, 15, 17]. However, generalizing the results of single studies is difficult because of the differences in contexts (type of the developed software, the organizational competence, scale etc.). For example, MacCormack et al. [12] have analyzed a sample of 29 Hewlett-Packards projects, and concluded that releasing an early prototype contributes to both lower defect-density, and higher productivity. Neufelder [15] has studied 17 organizations and correlated more than 100 parameters with defect-density. In her study, early prototyping does not correlate strongly with defect-density. On the other hand, both studies report that daily tests, incremental testing, and having test beds contribute strongly to lower defect-density. Results of these studies indicate that different development approaches may be associated with different quality attributes, and the impact of development approaches may vary in different contexts.

In order to assess development approaches and software quality attributes associated with those, we need valid data from measurement programs. *Measurement* is defined in [8] as the process by which numbers or symbols are assigned to attributes of entities in the real world in such a way to describe them according to clearly defined rules. *Measures* are the actual numbers or symbols being assigned to such attributes. The word *metrics* is both used to denote the field of measurement, and the schema that describes the measures.

3. The Ericsson Context

Ericsson in Grimstad-Norway has developed software for several releases of two large-scale telecommunication systems. The first system was originally developed to provide packet data capability to the GSM (Global System for Mobile communication) cellular network. A later recognition of common requirements with the forthcoming W-CDMA system (Wide-band Code Division Multiple Access) lead to reverse engineering of the developed architecture to identify reusable parts across applications [14], and to evolve the initial software architecture to an architecture that can support both systems.

The development process has evolved as well: The initial development process was a simple, internally developed one, describing the main phases of the lifecycle and the related roles and artifacts. After the first release, the organization decided to adapt RUP [16]. The two products (systems) are developed incrementally, using a component-based approach, and many artifacts are shared between these two products. New functionality is added to each release of the products, and each release goes through 5-7 increments. The size of each system (not including the system platform) is over 1000 NKLOC (Non-Commented Kilo Lines Of Code measured in equivalent C), and several hundred developers in different Ericsson organizations (up to 200 in Grimstad) have been involved in developing, and testing the releases.

3.1. Incremental and Component-Based Development in Practice

Karlsson describes some alternatives for defining increments and work allocation in [11]. We use his terminology to present our example. Figure 1 shows a view of activities during increments in a project, leading to a product release. Milestones (MS) are points in time when the project progress is evaluated. Ericsson has defined its own milestones that slightly differ from the standard RUP. At MS2, the project should have an approved requirement baseline. Any changes to requirements are afterwards handled by initiating a Change Request. The initial increment plan is made based on assigning use cases or features to increments (called for *feature increments* in [11], and in fact *use-case and feature increments* in our case). The duration of increments varies, but it is in the order of 6-12 weeks. For each increment:

1. It will be activities of requirements elicitation or refinement, analysis and design, implementation, integration and system testing for the current increment.
2. While software developed in increment *i* is being tested, increment *i+1* has started. Therefore each increment includes fault removal activities for previous increment(s). Faults should be removed both from the previous increments or releases, and the current one, and fault correction may introduce new faults.
3. Changes to the requirement baseline in form of Change Requests, may lead to deviation from the original increment plan, when it comes to effort and time-plan.
4. Several development teams work in parallel for implementing use cases and features assigned to the increment. Some of these teams may finish their work before others and start working on the next increment.
5. Several teams may update a component in an increment, since a component may be involved in several use cases or features, and work allocation is a combination of *increment responsibility* (a team is responsible for a use case in an increment) and *item responsibility* (each high-level component has a design coordinator that is responsible for following the item). These activities should be synchronized with each other and dependencies should be resolved.

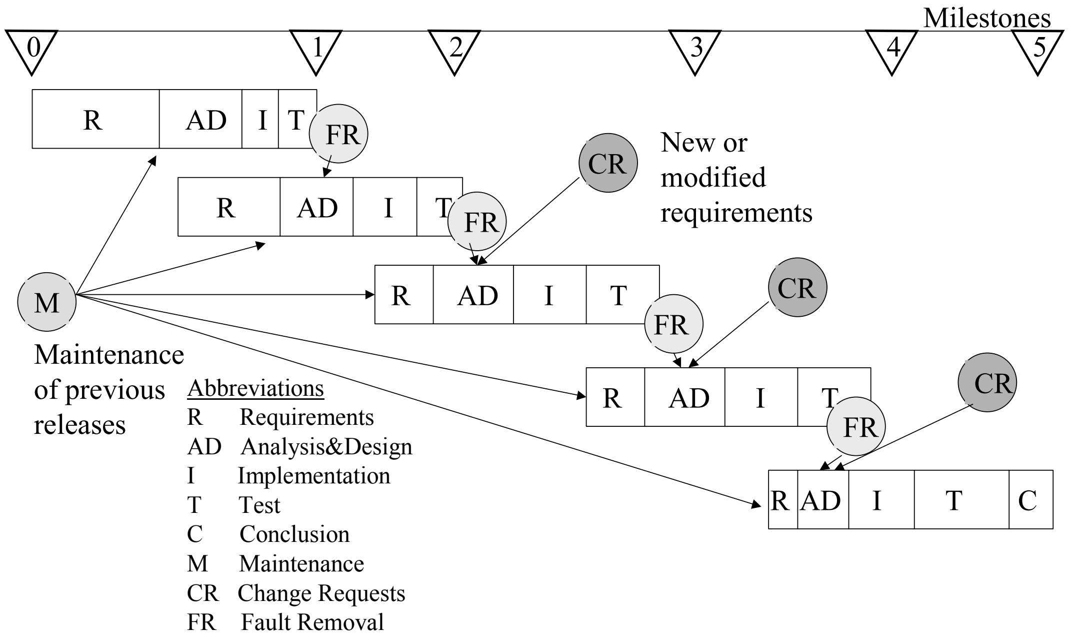


Figure 1 Increments and activities in each increment

What is specific to RUP is its use-case driven approach for requirement definition, design and test. But Ericsson had to combine use cases with features. A feature may be a non-functional requirement or a constraint that identifies two releases from each other, e.g. compliance to an interface or a standard.

The initial increment plan is based on effort estimated for implementing use cases or features defined in the requirement baseline at MS2. However, it is difficult to proceed according to the original increment plan, because of the stream of fault reports and change requests. It is also difficult to measure the actual effort used in each increment or on each requirement because of: (1) the effort used on fault removal for previous releases, and (2) in cases parallel increments (refer to point 5). A confounding factor is the system used to record effort. It collects effort used on delivered artifacts without recording the increment.

Although the idea behind incremental development is to deliver the final system in smaller parts, it was soon realized that too much functionality is delivered at the end of each increment, and each release, and many corrections should also be tested, which made integration and testing difficult. Therefore the project management developed an *integration plan* that described which features and corrections should be tested, and in which order. It is based on an *anatomy plan* that describes interconnections and dependencies of different functionalities. The integration plan turned out to be an effective mean to control the progress of both design and test.

3.2. Collected Data

The organization collects data according to a measurement program that covers both *direct measures* (such as measures of effort, duration of releases in calendar-weeks, software size, person-hours used in different activities, and number of faults or failures reported during a week) and *indirect measures* (calculated from direct measures, such as fault-density). The results are used to evaluate progress relative to project plans, and to assess software quality by measures such as inspection rate or the number of faults detected in each test phase. We argue that the measurement program is not updated for the incremental approach. For example it is not easy to find the effort spent in each increment or on each requirement.

There are also lots of data in different databases that are not linked to any specific metrics, and are not systematically analyzed. For example, the number of CRs during a project is followed up, but it is not analyzed which components are more change-prone (or less unstable). Parts of the system that are more change-prone should be designed in order to reduce maintenance effort. We analyzed some of these data (see S3 in Section 3.3) and observed weaknesses in the fault reporting and the change management system, so that presentation and analysis of data was not easy. For example, not all the fault reports include the name of faulty module, or have defined the type of fault.

3.3. Ericsson Experiments and Case Studies

The results of the following studies performed in 2001-2003 are used in this paper:

- **S1- Experiment on inspection of UML diagrams:** The goal was to compare two inspection techniques for inspection of UML diagrams; the Ericsson current technique and the new Object-Oriented Reading Techniques. The quality attribute was effectiveness in terms of the number of detected defects per person-hours used in the individual reading phase of UML diagrams [7]. The results showed that the two techniques were almost equally effective, but detected different type of defects.
- **S2- Estimation of effort based on use cases**: The goal is to extend an effort estimation technique based on use cases [1] in the context of reuse, and compare the results with experts estimations. The quality focus is the estimation precision. We used the technique on actual data from one release, with good results. We plan to assess the method using data from a second release during this year.

- **S3- Empirical assessment of quality attributes**: We have collected and analyzed historical data of three releases of the GPRS for GSM system. The quality focus is reliability and stability (or modification rate). We assessed some hypotheses using the available data, and will publish the results soon. For example, our results show that reused components are more reliable and stable than non-reused ones.
- **S4- Qualitative studies of the software process model (RUP) and the practice of reuse:** We studied RUP in the context of reuse and performed an internal survey on developers' attitude to reuse and RUP. We concluded that RUP does not have guidelines for product line engineering, and development for and with reuse. Results of the survey showed that developers are motivated for reuse, and consider reusable components to be more reliable and stable than non-reused ones (which is also proved by the results in S3).

Case studies (S2, S3 and S4) have the advantage of being performed in a real context, and the possibility to give feedbacks on collected data and results. Yet validity of case study results is difficult to assess: The researcher has little or no control over the confounding factors, he/she may have a researcher bias, and it may be difficult to generalize the results to other organizations [5].

4. Assessment of Development Approaches

There are methods such as Goal-Question-Metric (GQM) [3] and GQM/MEDEA [6] that are useful when a measurement process is about to be started, in order to determine which metrics to define. In our case, we have extensive data available from three releases of one of the products, but no explicit link between this data and the organizational goals. We therefore had to choose a bottom-up approach by collecting measures and analyzing these, and defining a set of hypothesis that could be assessed based on the available data. We don't present the hypotheses or the results in this paper, but present what kind of data may typically be the basis for such analyses.

Some observations and quantitative results that may be related to the development approaches (incremental, component-based and reuse) are:

1. Planning precision decreased from 91% to 78% over the three releases. The planning precision is defined as the absolute value of the actual time minus planned time (in number of weeks), divided by the planned time, and multiplied by 100, for each release. What observations could explain this?
   - Requirement stability (percentage of requirements that are not changed between MS2 and MS5 in Figure 1) decreased from 92% to 69% over the three releases. The **incremental** approach is chosen when the project foresees changing requirements or changing environment. But the remedy may reduce the threshold for accepting changes; i.e. managers are more willing to accept changes, in contrast to development approaches with rather frozen requirements.
   - Weber says that although **change** is part of the daily project life, change proposals occur more often than actual project changes [18]. In S3 we found the opposite: 68% of change requests are in fact accepted and implemented. The code is modified about 50% between two releases.

   - In S2, we realized the effect of **reuse** on effort-estimation. Many use cases are reused “as-is” or modified in a release, and effort-estimation methods must be able to account for this.
   - Assessing the effort-estimation model is difficult: Management estimated based on features or use cases assigned to increments, while the actual effort is recorded for components or other artifacts. This is the combination of **feature increments** (see Section 3.1), and **component** or **item-oriented** way of thinking from the time before incremental development.
2. Too much functionality is delivered at the end of each increment, and each release, which caused integration-bangs:
   - Qualitative feedbacks indicate that it is sometimes difficult to map requirements into increments of the right size, and many non-functional requirements could not be tested in the early increments. This is associated with the **incremental** approach.
   - **Components** use a component framework that should be developed early. Although design of the application components started in parallel with developing the framework, the functionality could not be tested in the test environment before the framework was ready.
3. The projects never reached the goal regarding Appraisal-to-Failure-Rate (AFR), which is defined as person-hours used for reviews and inspections, rework included, divided by person-hours used for test and rework. It is assumed that a higher AFR indicates focus on early fault detection:
   - In S1, we realized that many artifacts are modified in several increments, and it is not possible to inspect these every time something is modified. This is associated with the **feature increments** approach.

Empirical studies such as S3 could be useful in understanding and identification of relationships between variables, in order to assess development approaches. Establishing a relationship between the development approach and planning precision, modification rate, defect-density, productivity (in increments and totally) etc. can be subject of empirical studies in our case. Another goal is to adapt development approaches to the industrial context, and answer questions regarding increments’ functionality, work allocation, and adaptation of verification techniques such as inspections or testing to development approaches.

5. Assessment of Measurement Programs and Data Collection Methods

Performing empirical studies early in the life cycle of a project would help the organization to assess the quality of the measurement program and the collected data, and to improve it. The key is to find to what metrics could be related to the organization goals, what is not useful to measure, or what other metrics should be defined. Examples of such observations during our studies are:

- We realized that assessing the effort-estimation model is difficult, and the collected data on effort used on each artifact or component is useless unless the effort estimation model is changed. With the current estimation model, the useful data is the total effort for each release and the size of the delivered code (which were also used in our effort estimation model based on use cases in S2).

- Ericsson decomposes the system into subsystems, and each subsystem consists of several blocks (which in turn consists of units and software modules). Both subsystems and blocks have interfaces defined in the Interface Definition Language (IDL) and may be defined as components. During statistical analysis of the results in S3, we found that subsystems are too coarse-grained to be used as components, and give us too few data points to establish any relationship of statistical significance between size of them and quality attributes such as stability or reliability. On the other hand, we could show such relationships if blocks were chosen as components. Our empirical study has therefore been useful to decide the granularity of components for data collection.
- When a problem is first detected, the developer or tester fills a fault report using a web interface, and writes the name of the faulty component (or software module), if it is known. In many cases this information is not known when the fault report is initiated. As the field for the name of the faulty module is not updated later, tracing faults to software modules become impossible without parsing the entire version control system to find this information in the source code (where it is written every time the code is updated). The same is true for change requests, which originally include an estimate over the impact of the change request, and are not updated later with the actual effort used and the name of modified components. If the fault report or change management systems asked developers to insert data on the modified components when closing the reports, tracing between these two and the modified code would be much easier. In S3, we found that 22% of fault reports for a release did not give any subsystem name for the origin of the fault, only half of them had information on the block name, and very few on the software module name. We therefore could not use many of the fault reports in assessing hypotheses regarding reliability of components.

Some suggestions for improving the Ericsson's measurement program, fault reporting and change management systems, associated with the development approaches are:

Incremental development: To have better control on increment and release plans, it is important to have control over three factors: (a) functionality delivered in an increment or release, (b) parts of the system that have changed, and (c) the link between a) and b); i.e. *traceability* between requirements and deliveries. Our observations are:

- The development environment (Rational Rose associated with RUP for requirement definition and modeling, and mostly manually written code) does not have tools that provide traceability automatically. But there are tools that can find differences between files in the version control system. One possible solution would be to gather data on the modified model, and code at some pre-planned intervals, like on delivery dates, or before code for a use case or feature is merged into the final delivery.
- Measures of change such as percentage of modified code, percentage of modified requirements, change requests etc. are important to assess quality attributes such as productivity or defect-density. Unlike the waterfall model of development, it is not enough to measure these quality attributes once at the end of the project, but they should be measured for increments.

- The effort recording system should be updated so that we can measure person-hours used in each increment, and on each requirement to assess productivity and planning precision.

CBSE: It is important to define and measure quality attributes for components. We need metrics such as:

- Defect-density per component: Update fault reports with the name of the faulty component after correcting it.
- Component size and size of modified code in Lines of Code (LOC) to assess stability and reliability. LOC is a good measure of component size, which is easy to gather by automated tools.
- Change requests per component to assess stability or volatility. Update change requests with information on the modified components (and the actual effort).

Reuse: Metrics would be:

- Reuse percentage between releases to assess reuse gain (in productivity, stability, etc).
- Classification of components as reused, new, or modified.

Some general observations regarding the measurement programs and project plans are:

- Don't over-measure and don't gather data that you won't analyze.
- Project plans should have room for changing requirements.
- To assess the effectiveness of inspections and testing phases, record all faults in a single database with information on the detection phase (inspections, unit testing, etc). Today, these data are recorded using different tools. A single web interface that stores the data in a database would ease presentation and analysis of the data.
- Use data to improve software quality. As an example, data on number of faults for each component could be used to identify the most fault-prone ones early in the project and take action.
- Have realistic goals and modify them if necessary. Unachievable goals (such as for AFR) do not motivate.
- Establish a plan for benchmarking (comparing the measures with peer organizations) for future projects.
- Too many changes have negative impact on quality and planning precision. Use metrics such as requirement stability and modified lines of code to assess volatility.
- Be aware of the impact of the chosen development approaches. Learn from your own experiences and the results of other studies (although there are few published results from large-scale industrial projects).

We would also like to ask whether organizations are too afraid to draw conclusions based on their own experiences. Usually there are many confounding factors that make this difficult, and it is always easier to blame management or developers when a goal is not reached, than modifying the development approach or the goal. Reorganizations (and other organizational "noise") are also reasons why improvement works are not followed up.

6. Conclusions

Many organizations gather a lot of data on their software process and products. This data is not useful if it is not related to defined goals, not adapted to the development approach, or not analyzed at all. The incremental nature of development makes gathering data more difficult than earlier since data from increments flow into each other, and each increment is dealing with the past, the present and the future. We gave an example of activities in each increment in an industrial context, and presented some measurement results and project experiences that may be related to the incremental and component-based development approaches. Establishing a causal relationship between development approaches and variables such as stability and reliability could be subject of empirical studies, in order to assess these approaches. We discussed that methods for effort estimation, fault reporting or change control, and tools associated with them, should also be updated for the development approach. We also discussed why empirical studies are useful to assess measurement programs and gave examples of metrics that are useful based on the development approach. We think that organizations should put more effort in defining goals for measurement programs, assessing the quality and usefulness of the collected data, and assessing the development approaches based on empirical studies.

Acknowledgements

The work is done in the context of the INCO project (INcremental and COmponent-based Software Development [10]) a Norwegian R&D project in 2001-2004, and as part of the first author's PhD study. We thank Ericsson in Grimstad for the opportunity to perform the studies.

References

1. Anda, B.: Comparing Effort Estimates Based on Use Case Points with Expert Estimates. *Proc. The Empirical Assessment in Software Engineering (EASE 2002)*, Keele, UK, April 8-10, 2002.

2. Atkinson, C., Bayer, J., Bunse, C., Kamsties, E., Laitenberger, O., Laqua, R., Muthig, D., Paech, B., Wust, J., Zettel, J.: *Component-Based Product Line Engineering with UML*. Addison-Wesley, 2002.

3. Basili, V.R., Caldiera, G., Rombach, H.D.: *Goal Question Metrics Paradigm*. Encyclopedia of Software Engineering, Wiley, I (1994) 469-476

4. Boehm, B., Abst, C.: A Spiral Model of Software Development and Enhancement. *IEEE Computer*, 31(5):61-72, 1998.

5. Bratthall, L., Jørgensen. M.: Can you Trust a Single Data Source Exploratory Software Engineering Case Study? *The Journal of Empirical Software Engineering*, 7(2002):9-26.

6. Briand, L.C., Morasca, S., Basili, V.R.: An Operational Process for Goal-Driven Definition of Measures. *IEEE Trans. Software Engineering*, 28(12):1106-1125, 2002.

7. Conradi, R., Mohagheghi, P., Arif, T., Hegde, L.C, Bunde, G.A., Pedersen, A.: Object-Oriented Reading Techniques for Inspection of UML Models- An Industrial Experiment. Proc. of the 17th European Conference on Object Oriented Programming- ECOOP2003, *Springer-Verlag Berlin Heidelberg*, 2003, pp.483-501.

8. Fenton, N., Pfleeger, S.L.: *Software metrics: A Rigorous & Practical Approach.* 2nd ed, International Thomson Computer Press, 1997.

9. Heineman, G.T., Councill, W.T.: *Component-Based Software Engineering, Putting the Pieces Together*. Addison-Wesley, 2001.

10. INCO (INcremental and COmponent-based Software Development): http://www.ifi.uio.no/~isu/INCO/

11. Karlsson, E.A.: Incremental Development- Terminology and Guidelines. In *Handbook of Software Engineering and Knowledge Engineering*, Volume 1. World Scientific, 2002, pp.381-401.

12. MacCormack, A., Kemerer, C.F., Cusumano, M., Crandall, B.: Trade-offs between Productivity and Quality in Selecting Software Development Practices. *IEEE Software,* 20(5):78-85, 2003.

13. Malaiya, Y.K., Denton, J.: Requirements Volatility and Defect Density. *Proc. The International Symposium on Software Reliability Engineering (ISSRE'99)*, 1999, pp. 285-294.

14. Mohagheghi, P., Conradi, R.: Experiences with Certification of Reusable Components in the GSN Project in Ericsson, Norway. In Judith Stafford et al. (Eds.): Proc. 4th ICSE Workshop on Component-Based Software Engineering: Component Certification and System Prediction (ICSE'2001), Toronto, May 14-15, 2001, pp. 27-31. *SU-report 6/2001*, 5 p.15.

15. Neufelder, A.M.: How to Measure the Impact of Specific Development Practices on Fielded Defect Density. Proc. *The 11th International Symposium on Software Reliability Engineering (ISSRE'00),* 2000, pp.148-160.

16. Rational Unified Process: www.rational.com

17. Slaughter, S.A., Banker, R.D.: A Study of Effects of Software Development Practices on Software Maintenance Effort. *Proc. The International Conference on Software Maintenance (ICSM'96)*, 1996, pp.197-205.

18. Weber, M., Weisbrod, J.: Requirements Engineering in Automative Development: Experiences and Challenges. *IEEE Software*, 20(1):16-24, 2003.

19. Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A.: *Experimentation in Software Engineering: An Introduction.* Kluwer Academic Publishers, 2000.

P6. Different Aspects of Product Family Adoption

Parastoo Mohagheghi[1,2,3], Reidar Conradi[2,3]

[1] Ericsson Norway-Grimstad, Postuttak, NO-4898 Grimstad, Norway
[2] Department of Computer and Information Science, NTNU, NO-7491 Trondheim, Norway
[3] Simula Research Laboratory, P.O.BOX 134, NO-1325 Lysaker, Norway
parastoo.mohagheghi@ericsson.com
conradi@idi.ntnu.no

Abstract

Ericsson has successfully developed two large-scale telecommunication systems based on reusing the same software architecture, software process, and many other core assets. The approach to initiating a product family has been a lightweight approach, and many artifacts are evolved during product family adoption, although not to the same degree. The software architecture has evolved to support reuse and handling of variations, while the software process model is not updated for product family engineering and reuse. We discuss what works and doesn't work in the current process model, and why it is important to synchronize it with the practice of software development. Product family adoption has raised challenges in many aspects of software development such as requirement management, and measurement. These processes should also be evolved to fit the software development approach.

1. Introduction

Many organizations are using a product family engineering approach for software development by exploiting commonalities between software systems, reusing software architecture, and a set of core assets. The approach to start a product family and evolve it varies, depending on the context, and the term *product family* is used for a wide range of approaches to develop software with reuse. For example, the degree to which some reusable assets are identified before the first product is used to distinguish between heavyweight, and lightweight approaches to initiate a product family.

Ericsson has developed two large-scale telecommunication systems that share software architecture, software process model, and other core assets using a lightweight approach. The software architecture has evolved to an architecture that promotes reuse, and product family engineering. Although the software process model is evolved in parallel with product family adoption, it has not been adapted for this aspect of development, and lacks explicit guidelines for domain engineering and reuse. I.e. there is a gap between the software process model, the adapted Rational Unified Process (RUP), and the actual process (the practice of software development). The internally developed guidelines, existing knowledge, and expertise compensate to some degree for shortcomings in the process model. Adopting product family engineering has impact on many aspects of software development. If these aspects are not evolved harmoniously, conflicts may appear in areas such as requirement engineering where a product family approach is more feature-oriented, while RUP is use-case driven. Resolving these conflicts is part of the adoption process, and analyzing experiences is important for learning feedbacks.

The remainder of the paper is structured as follows. Section 2 describes some state of the art. Section 3 describes the Ericsson context, and Section 4 discusses the strengths, and weaknesses of the current process model. The paper is concluded in Section 5.

2. A Brief State-of-the-Art

Parnas wrote the first paper on development of systems with common properties in 1976. He wrote:" We consider a set of programs to constitute a *family*, whenever it is worthwhile to study programs from the set by *first* studying the common properties of the set, and *then* determining the special properties of the individual family members" [14]. He called these systems *program families*, while other terms are *system families*, *product lines,* or, as we prefer to call it-*product families*. Product families are built around *reuse*: reuse of requirements, software architecture and design, and implementation. Bosch writes, "the software product line approach can be considered to be the first intra-organizational software reuse approach that has proven successful" [3].

Several software development processes support product family engineering, see for example [1, 2, 5, 7, 8]. The Software Engineering Institute (SEI) defines three essential product family activities [13]:

1. D*omain engineering* for developing the architecture and the reusable assets (or development *for* reuse as called in [8]).
2. *Application engineering* to build the individual products (or development *with* reuse as called in [8]).
3. *Management* at the technical and organizational level.

In [10] approaches for introducing a product family are divided into *heavyweight,* and *lightweigh*t. In the heavyweight approach, commonalities are identified *first* by domain engineering, and product variations are foreseen. In the lightweight approach, a first product is developed, and the organization then uses mining efforts to extract commonalities. The choice of approach also affects cost and the organization structure. Krueger claims that the lightweight approach can reduce the adoption barrier to large-scale reuse, as it is a low-risk strategy with lower upfront cost [9]. Johnson and Foote write in [6] that useful abstractions are usually designed from the bottom up; i.e. they are discovered not invented.

If the approach to initiate a product family is a lightweight approach, the shared artifacts such as the software process should evolve in order to be reusable. By a *software process* we mean all activities, roles and artifacts that produce a software product, and a s*oftware process model* is a representation of it. These artifacts are not always evolved harmoniously and synchronously, and some of them are more critical for the success of the product family. The process of change is a composition of organizational, business, and technical factors.

3. An Industrial Example of Product Family Adoption

The General Packet Radio Service (GPRS) system provides a solution to send packet data over the cellular networks. GPRS was first developed to provide packet data capability to the GSM (Global System for Mobile communication) cellular network. A later recognition of common requirements with the forthcoming W-CDMA system (Wide-band Code Division Multiple Access) lead to reverse engineering of the developed architecture to identify reusable parts across applications, and to evolve the software architecture to an architecture that can support both products. This was a joint development effort across organizations for almost one year, with negotiations and renegotiations.

The initial software architecture is shown in the left part of Figure 1. Components are tightly coupled, and all use services of the platform (WPP), and a component that provides additional middleware functionality. Evolution of the software architecture was mainly done in two steps:

– Extracting the reusable components, and evolving the architecture into the one shown in the right part of Figure 1. Old components are inserted in the layers based on their reuse potential, and some are split into several new components in different layers.
– Removing coupling between components that break down the layered architecture. These removed couplings are shown with red dashed arrows in the left part of Figure 1. Components in the lower layers should be independent of components in the higher layers.

The reused components in the business-specific layer (that offers services for the packet switching networks), and the common services layer (includes a customized component framework for building robust real-time applications, and other services) stand for 60% of the code in an application, where an application in this context consists of components in the three upper layers. The size of each application is over 1000 NKLOC (Non-Commented Kilo Lines Of Code measured in equivalent C code).

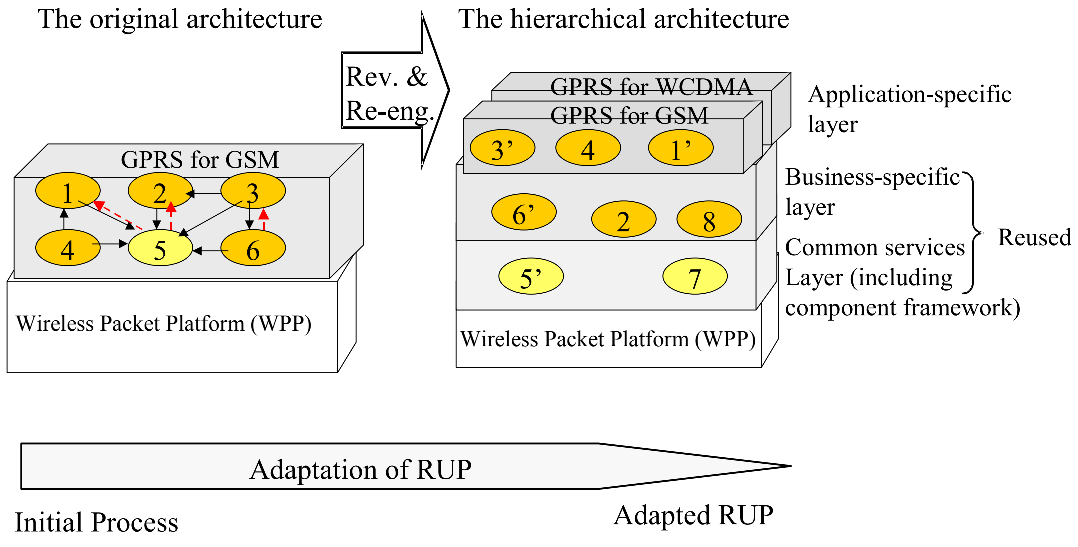

The original architecture
Rev. & Re-eng.
The hierarchical architecture
GPRS for GSM
1
2
3
4
5
6
Wireless Packet Platform (WPP)
GPRS for WCDMA
GPRS for GSM
3'
4
1'
Application-specific layer
6'
2
8
Business-specific layer
5'
7
Common services Layer (including component framework)
Reused
Wireless Packet Platform (WPP)
Adaptation of RUP
Initial Process
Adapted RUP

Figure 1 Evolution of the GSN software architecture and the software process model

The approach to product line adoption has been a *lightweight* approach. The first product was initially developed and released, and the commonalities between it, and the requirements for the new product lead to the decision on reuse. The products are developed incrementally, and new features are added to each release of the products. Several Ericsson organizations have been involved in development and testing.

The software process has been developed in parallel with the products. The first release of the GPRS for GSM product used a simple, internally developed software process, describing the main phases of the lifecycle and the related roles and artifacts. After the first release, the organization decided to adapt the Rational Unified Process (RUP) [15]. The adaptation is done by adding, removing or modifying phases, activities, roles, and artifacts in the standard RUP process. RUP is an *architecture-centric* process, which is an advantage when dealing with products using the same reference architecture. But RUP in its original form is not a process for product families, and we argue that it has not been adapted for this aspect of development:

- The main workflows (requirement, analysis and design, implementation and testing) are described as if there is a single product development, while configuration management activities handle several versions and several products.
- There is no framework engineering in the adapted RUP, and developing framework components (or in general reusable components) is an indistinguishable part of application engineering.

To provide the information needed for software developers, artifacts such as internally developed modeling guidelines, and design rules are linked to the workflows in RUP. We mean that there is a gap between the process model (the adapted RUP), and the practice of software development (the actual process).

4. What Works and Does not Work in the Software Process?

We have studied the software process, and performed a small survey in the Ericsson organization in Grimstad-Norway to understand developers' attitude towards reuse, and the software process model. We present some results of our study in this paper.

The adapted RUP has been in use for almost four years, and have some benefits:

1. RUP is architecture-centric, as mentioned. Software architecture plays the key role in engineering product families.
2. RUP is adaptable.
3. Rational delivers RUP together with a whole range of other tools for requirement management, configuration management etc.
4. The developed web pages for RUP are understandable.

We asked whether the lack of explicit reuse-related activities in the process model affects the reuse practice. The survey results indicate such impact. For example, developers mean that the reused components are not sufficiently documented, and assessing components for reuse is not easy.

Some suggestions for improving the process model for reuse are given in [12], and [16]. Some of the suggestions are easier to introduce than others. Example is adding the activity *Record reuse experience* to the *Conclusion Phase* (Ericsson has added the Conclusion Phase to the adapted RUP as the last phase of a project). On the other hand, distinguishing domain, and application engineering has impact on several workflows, and is more difficult to carry out.

Product family adoption has impact on all aspects of the software process and raises challenges that should be solved. Some of our observations are:

1. Requirement management for reusable components is difficult. The attempts to specify requirements in terms of use cases that should be included or extended in the application use cases (as proposed in [5]) was not successful as complexity grows, and dependencies become unmanageable. Use cases were therefore dropped for reusable parts, and replaced by textual documents that describe functionality and variation points.
2. There is a measurement program in the organization, but specific metrics for reuse, and product family engineering should be more stressed.
3. Requirements to each release of the systems are defined in terms of features, and it is features that distinguish releases, and products from each other, while RUP is use-case driven. Tracing from features to use cases, and later design, and deliveries is difficult.

We have started working on some of these issues like metrics. We have collected trouble reports and requirement changes from several releases, and defined hypotheses that can be verified based on the available data. Results of this study can be used to assess the development approach, and to improve the measurement program, as described in [11].

5. Conclusions

We described an industrial example of product family adoption, where the products have a high degree of reuse, and share a common software architecture and software process. The lightweight approach to adoption has been successful in achieving shorter time-to-market and lower development costs. The role of the software architecture in product family adoption has been critical. The software architecture distinguishes reusable components from application-specific components, and promotes reuse. The software process model has not evolved to the same degree, and does not reflect the practice. As the software is developed incrementally, and the development projects have been running for 5 years, the existing knowledge, and the internally developed guidelines compensate to some degree for shortcomings in the process model. We discussed strengths and shortcomings in the adapted RUP, and described some aspects of software development that are affected in adopting product family engineering. The inadequate adaptation of the software process model has impact on the reuse practice (such as insufficient documentation of reusable parts, and lack of metrics to evaluate reuse gains), and we think that the organization can benefit through more adapting it to product family engineering.

6. Acknowledgements

The work is done in the context of the INCO project (INcremental and COmponent-based Software Development [4]), a Norwegian R&D project in 2001-2004, and as part of the first author's PhD study. The survey on developers' attitude to reuse, and some

improvement suggestions regarding reuse are part of two MSc diploma theses [12, 16]. We thank Ericsson in Grimstad for the opportunity to perform the studies.

References

1. Atkinson, C, Bayer, J., Bunse, C., Kamsties, E., Laitenberger, O., Laqua, R., Muthig, D., Paech, B., Wüst, J., Zettel, J.: Component-based Product Line Engineering with UML. Addison-Wesley (2002)
2. Bosch, J.: Design and Use of Software Architecture: Adpoting and Evolving a Product-Line Approach. Addison-Wesley (2000)
3. Bosch, J.: Maturity and Evolution in Software Product Lines: Approaches, Artifacts and Organization. In Proc. of the Second Software Product Line Conference- SPLC2 (2002). Available at http://www.cs.rug.nl/~bosch/
4. INCO project: http://www.ifi.uio.no/~isu/INCO/
5. Jacobson, I., Griss, M., Jonsson, P.: Software Reuse: Architecture, Process and Organization for Business Success. ACM Press (1997)
6. Johnson, R.E., Foote, B.: Designing Reusable Classes. Journal of Object-Oriented Programming, 1(3): 26-49 (1998)
7. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, A.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Software Engineering Institute Technical Report CMU/SEI-90-TR-21, ADA 235785 (1990)
8. Karlsson, E.A. (Ed.): Software Reuse, a Holistic Approach. John Wiley & Sons (1995)
9. Krueger, C.: Eliminating the Adoption Barrier. IEEE Software, 19(4): 29-31 (2002)
10. McGregor, J.D., Northrop, L.M., Jarred, S., Pohl, K.: Initiating Software Product Lines. IEEE Software, 19(4): 24-27 (2002)
11. Mohagheghi, P., Conradi, R.: Using Empirical Studies to Assess Software Develoment Approaches and Measurement Programs. Forthcoming at the 2nd Workshop in Workshop Series on Empirical Software Engineering (WSESE'03), Rome-Italy (2003)
12. Naalsund, E., Walseth, O.A.: Decision Making in Component-Based Development. NTNU diploma thesis, 92 p. (2002) http://www.idi.ntnu.no/grupper/su/su-diploma-2002/naalsund_-_CBD_(GSN_Public_Version).pdf
13. Northrop, L.M.: SEI's Software Product Line Tenets. IEEE Software, 19(4):32-40 (2002)
14. Parnas, D.L.: On the Design and Development of Program Families. IEEE Trans. Software Eng., SE-2(1):1-9 (1976)
15. Rational Unified Process, http://www.rational.com
16. Schwarz, H., Killi, O.M., Skånhaug, S.R.: Study of Industrial Component-Based Development. NTNU pre-diploma thesis, 105 p. (2002) http://www.idi.ntnu.no/grupper/su/sif8094-reports/2002/p2.pdf

P7. An Industrial Case Study of Product Family Development Using a Component Framework

Parastoo Mohagheghi
Ericsson Norway-Grimstad, Postuttak, NO-4898 Grimstad, Norway
Department of Computer and Information Science, NTNU, NO-7491 Trondheim, Norway
Simula Research Laboratory, P.O.BOX 134, NO-1325 Lysaker, Norway
Phone: (+47) 37 293069, fax: (+47) 37 293501, e-mail: parastoo.mohagheghi@ericsson.com

Reidar Conradi
Department of Computer and Information Science, NTNU, NO-7491 Trondheim, Norway
Simula Research Laboratory, P.O.BOX 134, NO-1325 Lysaker, Norway
Phone: (+47) 73 593444, fax: (+47) 73 594466**,** e-mail: conradi@idi.ntnu.no

Abstract

Component-based software engineering, product family engineering, and reuse are increasingly used by software development organizations in order to achieve higher productivity, better software quality and shorter time-to-market. The paper describes a case study where two large-scale telecommunication systems are developed using a lightweight approach to product family adoption, and based on reusing a software architecture, a software process, a component framework and many other assets. The software architecture has evolved to a layered one that promotes reuse, and product family development. The internally developed component framework is part of the software architecture by defining rules and conventions for architecture and design. It is also part of the final product by providing run-time services for components. The component framework embraces many quality requirements either by implementing mechanisms that affect a quality requirement, or by taking design decisions for application developers, or a combination of both. The framework is realized as a package containing several subsystems, and is documented in UML models, textual descriptions, design rules, and programming guidelines. Developing a component framework is both similar to, and different from application engineering. The difference is usually mentioned to be requirement gathering from several applications, and handling of variability between products. Organizations should also put extra effort in documenting, and testing a component framework to make it reusable and reliable. If a component framework is developed in parallel with the applications using it, requirements of the framework are gradually discovered during design of applications, and the framework developers should solve the dilemma between early and late decision taking, and between being restrictive or flexible. Using a component framework will impact application engineering in many ways. Unlike component technologies like EJB or COM that are considered for realization, and implementation of components, component frameworks include reusable designs for a specific domain, and should be integrated early into the development process of applications. For the success of development with reuse, and in this case based on a component framework, it is crucial to evaluate the impacts early, and to adapt the development process.

Keywords. Product family, component framework, reuse, quality requirements, software architecture, software process.

1. Introduction

Many organizations are using a product family approach for software development by exploiting commonalities between software systems, and thus reusing a common software architecture, and a set of core assets. In this context, component frameworks are large-scale components that may be shared between applications. Ericsson has developed two products to deliver GPRS (General Packet Radio Service) to the GSM, and W-CDMA networks using a lightweight approach to product family adoption. The software architecture has evolved to a layered one that promotes reuse, and product family development. It includes an internally developed component framework that captures many of the quality requirements. Evolution to a product family has impact on many artifacts, and analyzing experiences is important for learning feedbacks.

The remainder of the paper is structured as follows: Section 2 describes some state of the art, and Section 3 describes the Ericsson context. Section 4 discusses why software processes should be adapted for development with a component framework, and some experiences from developing a component framework. The paper is concluded in Section 5.

2. Component Frameworks and Product Families

Components are another way to answer the challenge of modularity or decomposition of a system to smaller parts. Some other ways are modules (e.g. in Ada and procedural languages), and objects in object-oriented design. A component is an independently deliverable piece of functionality, providing access to its services through interfaces. Component-Based Software Engineering (CBSE) is concerned with assembly of systems from pre-built components, where components conform to a component model that defines rules, and conventions on how components interact [3,9]. Implementation of such a component model to offer run-time services for components is usually called a component framework. CBSE approaches are yet far from mature, but nevertheless, use of components is a clear trend in industry. One main reason is that CBSE offers an opportunity to increase productivity by reuse. Product family engineering is exploiting top-down reuse (reusing software architecture, and domain-specific frameworks), combined with bottom-up design to reuse existing components [4]. It is therefore considered as “the first intra-organizational software reuse approach that has proven successful” [5]. Several software development processes support product family engineering, and reuse, e.g. [2, 4, 7, 11, 13]. SEI defines the following three essential product family activities [17]:

1. Core asset development or *domain engineering* for developing the architecture, and the reusable assets (or development *for* reuse [13]).

2. *Application engineering* to build the individual products (or development *with* reuse [13]).
3. *Management* at the technical, and organizational level.

In practice the amount of domain engineering vs. application engineering varies, depending on the stability of the application domain, and maturity of the organization [5]. In [15], approaches for introducing a product family are divided into *heavyweight,* and *lightweigh*t. In the heavyweight approach, commonalities are identified *first* by domain engineering, and product variations are foreseen. In the lightweight approach, a first product is developed, and the organization then uses mining efforts to extract commonalities. The choice of approach also affects cost, and the organization structure. Krueger claims that the lightweight approach can reduce the adoption barrier to large-scale reuse, as it is a low-risk strategy with lower upfront cost [14]. Johnson and Foote write in [12] that useful abstractions are usually designed from the bottom up; i.e. they are discovered not invented.

Developing a component framework is both similar to, and different from application engineering. The difference is usually mentioned to be requirement gathering from several applications, handling of variability between products (e.g. in KobrA[2] by decision trees), and documentation of the framework for application developers. However, using a component framework (or in general frameworks; which covers earlier object-oriented frameworks as well) will impact application engineering. Frameworks include reusable designs for a specific domain (as mentioned by Gamma et al. [8]). Unlike component technologies like EJB or COM that are considered for realization, and implementation of components, frameworks define rules for architecture and design, and should be integrated early into the development process of applications.

3. The Ericsson Context

The GPRS system provides a solution to send packet data over the cellular networks. GPRS was first developed to provide packet data capability to the GSM (Global System for Mobile communication) cellular network. A later recognition of common requirements with the forthcoming W-CDMA system (Wide-band Code Division Multiple Access) lead to reverse engineering of the developed software architecture to identify reusable parts across applications, and to evolve the software architecture to one that can support both products. This was a joint development effort across organizations for almost one year, with negotiations, and renegotiations. We describe two aspects of product family adoption: Developing a reusable software architecture, and developing a reusable component framework as part of it.

3.1. Evolution of the Software Architecture

The left part of Figure 1 shows the initial software architecture. Components are tightly coupled, and use services of the platform (WPP, which is a high-performance packet switching platform developed by Ericsson in parallel with the products), and a central component, the Network Control Subsystem or NCS, that provides additional middleware functionality. Components have interfaces defined in the Interface Definition Language (IDL), and the broking mechanism of CORBA is extended for communication. Product family adoption was based on outlining a strategy for development with reuse by:

- Extracting the reusable components, and evolving the software architecture into the one shown in the right part of Figure 1. Old components are inserted in the layers based on their reuse potential, and some are split into several new components in different layers. Variation points are identified.
- Removing coupling between components that break down the layered software architecture (shown with red dashed arrows in the left part of Figure 1). Instead, components in the higher layers register a callback interface whenever they should be called by the lower layer components, for example when they should be notified on special events.
- Developing a component framework based on NCS. The whole component framework is reused as a component.

Three layers are defined on the top of the platform: 1) the application-specific layer contains components that are specific for application systems (GPRS for GSM, and GPRS for W-CDMA), 2) the business-specific layer contains components that offer services for packet switching cellular networks, and are shared between the two applications, 3) the common services layer includes the component framework, and components that may be reused in other contexts as well.

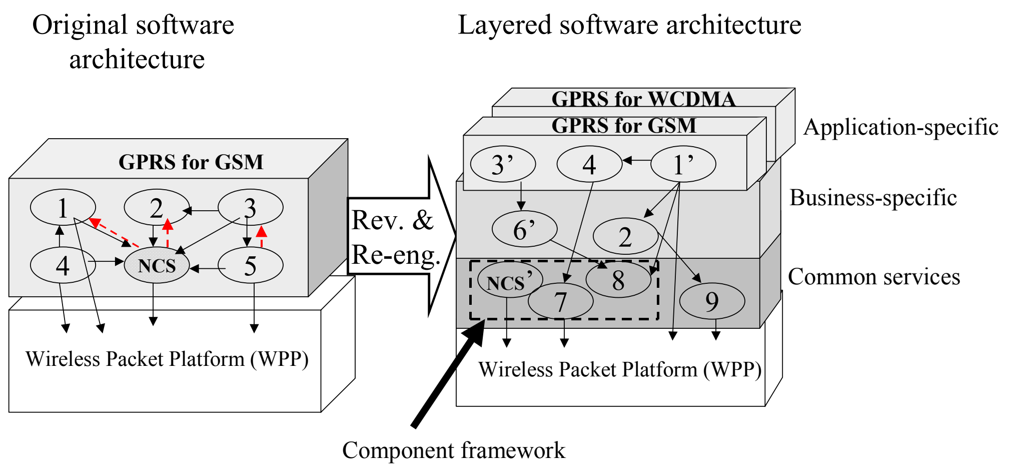


Figure 1 Evolution of the GSN software architecture

The original software architecture had one dimension based on the functionality of the components. The evolved software architecture has another dimension as well: the reuse dimension or generality. The common software architecture captures not only commonalities, but also variations between products, and has shown to be stable, and at the same time highly adaptable to new requirements. The reused components in the business-specific, and common services layers stand for 60% of the code in an application, where an application in this context consists of components in the three upper layers. The size of each application (not including WPP) is over 1000 NKLOC (Non-Commented Kilo Lines Of Code measured in equivalent C code). Software components are mostly developed internally. Software modules are written in C, Java, and Erlang (a functional language for programming concurrent, real-time, and distributed fault-tolerant systems).

GSN's approach to product family adoption has been a lightweight one: The first product was initially developed and released, and the commonalities between the developed product, and the requirements for the new product lead to the decision on reuse. The approach gave much shorter time-to-market for the second product, while the first one could still meet its hard schedules for delivery.

The software development process is an adaptation of the Rational Unified Process (RUP) [18]. In [16], we describe that the organization has developed several additional guidelines that assist developers to develop with reuse, but we mean that the software process model should be adapted more for reuse.

3.2. Component Framework and Quality Requirements

The component framework has several functionalities: It offers abstractions for hardware, and the underlying platform (WPP) for system functionality such as start or software upgrades, it offers run-time services such as transaction handling and broking, and it includes guidelines for building robust, real-time applications in a distributed multiprocessor environment. The framework is realized as a package containing several subsystems (components), and is documented in UML models, textual descriptions, design rules, and programming guidelines. It is part of the software architecture by defining rules and conventions for design. By providing run-time services for applications, it is part of deployment, and the delivered product as well.

Component frameworks are designed to ensure that systems using these will satisfy some quality requirements [9]. A quality requirement specifies an attribute of software that contributes to its quality where software quality is defined to be "the degree to which software possesses a desired combination of attributes", e.g. reliability, or interoperability [IEEE-1061]. The internally developed component framework embraces quality requirements either by implementing mechanisms that affect a quality requirement, or by taking design decisions for application developers, or a combination of both. For example, the reliability of a system improves by increased fault-tolerance, where the goal is to isolate faults, and preventing system failures in the presence of active faults, and also the subsequent system recovery. The component framework has mechanisms for both software and hardware fault-tolerance. Software fault-tolerance is handled by means such as starting separate threads for each user in order to isolate faults, replication of data, and persistent data storage. Hardware fault-tolerance is handled by hardware redundancy combined with reconfiguration of the system. Applications should register their desired hardware, and redundancy options in the component framework at start, which in turn handles reconfiguration in case of any hardware failure.

4. Discussion

A layered software architecture is discussed in the literature as an architectural style that increases maintainability by reduced coupling between components [4, 11]. It also classifies components for both component developers, and component assemblers. In addition to the software architecture, the internally developed component framework is shared between applications. The advantage is enhanced quality since the component framework is tested in more than one application. The disadvantage is the growing complexity of the framework, and possible trade-offs if requirements from several applications are in mutual conflict with each other.

What we observe in practice is that any software process should be adapted for development based on a component framework or even a component technology; either developed in-house or a commercial one. Cheesman et al. [6] describe such adaptation of a software process based on UML, Advisor [1], and RUP, and with a realization in EJB. However, domain-specific component frameworks should be integrated into the earlier phases of the development process; i.e. from requirement definition, and to analysis & design, testing, deployment, and documentation.

Developing a component framework is a complex task, and we list some challenges and experiences here. Some of these are especially related to the fact that the component framework was developed in parallel with applications using it:

- Requirements of the component framework were discovered gradually during design of the application components, rather than being explicitly specified in the beginning. The lightweight approach to reuse let to discover the main requirements to the component framework during developing the first product. But variation points are identified when requirements for several products are considered. Therefore, it is important to have a software architecture that is maintainable, i.e. changeable.
- If most of the design decisions are taken first, and captured in the framework, the risk is to have a software architecture that is not suitable for the problem. If the decisions are left to later phases, application developers may develop diverting solutions to the same problem, which is in conflict with the philosophy of the product family approach. I.e. there is a dilemma between early and late decision taking, and between being restrictive (enforcing many rules on application developers), and flexible.
- Some quality requirements cannot be assessed until the system is fully built. The developed component framework had to be optimized in several iterations for requirements such as performance.
- The software process was adapted in parallel with developing the products, and the software process model could occasionally not keep pace with development [16].
- Special testing and simulation tools had to be developed in order to improve testability of the applications based on the component framework.

We performed a small survey in the organization in spring 2002 with 9 developers, and asked their opinion on reuse, and the adapted RUP process. The results showed that design was considered as the most important artifact to reuse (other alternatives were requirements, code, test data, and documentation), and reused components were considered to be more stable and reliable (is also confirmed by an empirical study of defects). On the other hand, the developers wanted better documentation of the reused components and the component framework.

5. Conclusions

Rothenberger et al. [19] have analyzed several earlier reuse studies, and performed a principle component analysis to find the so-called "reuse success factors", where success measures are defined in terms of reuse benefits (e.g. reduction in cost or development time), strategic impact (reaching new markets), and software quality (reduction in defects). They concluded that software quality could be achieved based on project similarity, and common architecture. To gain high reuse benefits and strategic impact, three other dimensions must also be added, which are management support, formalized process, and planning & improvement. In our case study, many of these success factors are in place; i.e. management support, common architecture, and project similarity. The other two factors (formalized process, and planning & improvement) have medium degree of achievement, and could be subjects of improvement to achieve higher reuse benefits.

We discussed that software processes should be adapted for reuse, and for development based on a component framework, and presented some experiences related to developing a component framework. Adoption to a product family, and developing component frameworks are beneficial if the domain and projects have high reuse potential. However, a holistic approach is required since the adoption impacts all the aspects of software development.

6. Acknowledgements

The work is done in the context of the INCO project (INcremental and COmponent-based Software Development [10]), a Norwegian R&D project in 2001-2004, and as part of the first author's PhD study. We thank Ericsson in Grimstad for the opportunity to perform the study.

References

[1] Advisor, Sterling Software Component-Based Development Method, http://www.sterling.com/cool

[2] Atkinson, C., Bayer, J., Bunse, C., Kamsties, E., Laitenberger, O., Laqua, R., Muthig, D., Paech, B., Wüst, J., Zettel, J.: *Component-based Product Line Engineering with UML*. Addison-Wesley, 2002.

[3] Bachman, F., Bass, L., Buhman, C., Comella-Dorda, S., Long, F., Robert, J., Seacord, R., Wallnau, K.: Volume II: Technical concepts of Component-based Software Engineering. *SEI technical report CMU/SEI-2000-TR-008*. http://www.sei.cmu.edu/

[4] Bosch, J.: *Design and Use of Software Architecture: Adpoting and Evolving a Product-Line Approach*. Addison-Wesley, 2000.

[5] Bosch, J.: Maturity and Evolution in Software Product Lines: Approaches, Artifacts and Organization. *Proc. of the Second Software Product Line Conference*- SPLC2, 2002. Available at http://www.cs.rug.nl/~bosch/

[6] Cheesman, J., Daniels, J.: *UML Components, A Simple Process for Specifying Component-Based Software*. Addison Wesley, 2001.

[7] Clements, P., Northrop, L.M.: *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.

[8] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 22nd printing, 2001.

[9] Heineman, G.T., Councill, W.T.: *Component-Based Software Engineering, Putting the Pieces Together*. Addison-Wesley, 2001.

[10] INCO project (INcremental and COmponent-based engineering), http://www.ifi.uio.no/~isu/INCO/

[11] Jacobson, I., Griss, M., Jonsson, P.: *Software Reuse: Architecture, Process and Organization for Business Success*. ACM Press, 1997.

[12] Johnson, R.E., Foote, B.: Designing Reusable Classes. *Journal of Object-Oriented Programming*, 1(3): 26-49, 1988.

[13] Karlsson, E.-A. (Ed.): *Software Reuse, a Holistic Approach*. John Wiley & Sons, 1995.

[14] Krueger, C.: Eliminating the Adoption Barrier. *IEEE Software*, 19(4): 29-31, 2002.

[15] McGregor, J.D., Northrop, L.M., Jarred, S., Pohl, K.: Initiating Software Product Lines. *IEEE Software*, 19(4):24-27, 2002.

[16] Mohagheghi, P., Conradi, R.: Different Aspects of Product family Adoption. Forthcoming at the 5th International Workshop on Product Family Engineering, PFE-5, Siena-Italy, November 4-6, 2003.

[17] Northrop, L.M.: SEI's Software Product Line Tenets", *IEEE Software*, 19(4):32-40, July-August 2002.

[18] Rational Unified Process, http://www.rational.com

[19] Rothenberger, M.A., Dooley, K.J., Kulkarni, U.R., Nada, N.: Strategies for Software Reuse: A Principal Component Analysis of Reuse Practices. *IEEE Trans. Software Eng.*, 29(9): 825-837, September 2003.

P8. An Empirical Study of Software Reuse vs. Defect-Density and Stability

Parastoo Mohagheghi[1,2,3], Reidar Conradi[2,3], Ole M. Killi[2], Henrik Schwarz[2]
[1]*Ericsson Norway-Grimstad, Postuttak, NO-4898 Grimstad, Norway*
[2]*Department of Computer and Information Science, NTNU, NO-7491 Trondheim, Norway*
[3]*Simula Research Laboratory, P.O.Box 134, NO-1325 Lysaker, Norway*
parastoo.mohagheghi@ericsson.com, conradi@idi.ntnu.no, henrik@schwarz.no

Abstract

The paper describes results of an empirical study, where some hypotheses about the impact of reuse on defect-density and stability, and about the impact of component size on defects and defect-density in the context of reuse are assessed, using historical data ("data mining") on defects, modification rate, and software size of a large-scale telecom system developed by Ericsson. The analysis showed that reused components have lower defect-density than non-reused ones. Reused components have more defects with highest severity than the total distribution, but less defects after delivery, which shows that that these are given higher priority to fix. There are an increasing number of defects with component size for non-reused components, but not for reused components. Reused components were less modified (more stable) than non-reused ones between successive releases, even if reused components must incorporate evolving requirements from several application products. The study furthermore revealed inconsistencies and weaknesses in the existing defect reporting system, by analyzing data that was hardly treated systematically before.

1. Introduction

There is a lack of published, empirical studies on large industrial systems. Many organizations gather a lot of data on their software processes and products, but either the data are not analyzed properly, or the results are kept inside the organization. This paper presents results of an empirical study in a large-scale telecom system, where particularly defect-density, and stability are investigated in a reuse context. Software reuse has been proposed e.g. to reduce time-to-market, and to achieve better software quality. However, we need empirical evidence in terms of e.g. increased productivity, higher reliability, or lower modification rate to accept the benefits of reuse.

Ericsson has developed two telecom systems that share software architecture, components in reusable layers, and many other core assets. Characteristics of these systems are high availability, reliability, and scalability. During the lifetime of the projects, lots of data are gathered on defects, changes, duration time, effort, etc. Some of these data are analyzed, and results are used in the improvement activities, while some others remain unused. Either there is no time to spend on data analysis, or the results are not considered important or linked to any specific improvement goals. We analyzed the contents of the defect reporting system (containing all reported defects for 12 product releases), and the contents of the change management system. For three of these releases, we obtained detailed data on the size of components, and the size of modified code. We have assessed four hypotheses on reuse, and reused components using data from these releases. We present detailed results from one of these releases here. The quality focus is defect-density (as the number of defects divided by lines of code), and stability (as the degree of modification). The goal has been to evaluate parameters that are earlier studied in traditional reliability models (such as module size and size of modified code) in the context of reuse, and to assess the impact of reuse on software quality attributes.

Results of the analysis show that reused components have lower defect-density than non-reused ones, and these defects are given higher priority to solve. Thus reuse may be considered as a factor that improves software quality. We did not observe any relation between defect-density or the number of defects as dependent variables, and component size as the independent variable for all components. However, we observed that non-reused components are more defect-prone, and there is a significant correlation between the size of non-reused components, and their number of defects. This must be further investigated. The study also showed that reused components are less modified (more stable) than non-reused ones, although they should meet evolving requirements from several products.

Empirical evidence for the benefits of reuse in terms of lower defect-density, and higher stability is interesting for both the organization, and the research community. As the data was not collected for assessing concrete hypotheses, the study revealed weaknesses in the defect reporting system, and identified improvements areas.

This paper is organized as follows. Section 2 presents some general concepts, and related work. Section 3 gives an overview of the studied product, and the defect reporting system. Section 4 the research method, and hypotheses. Hypotheses are assessed in Section 5. Section 6 contains a discussion, and summary of the results. The paper is concluded in Section 7.

2. Related work

Component-Based Software Engineering (CBSE) involves designing and implementing software components, assembling systems from pre-built components, and deploying systems into their target environment. The reusable components or assets can take several forms: subroutines in library, free-standing COTS (Commercial-Off-The-Shelf) or OSS (Open Source Software) components, modules in a domain-specific framework (e.g. Smalltalk MVC classes), or entire software architectures, and their components forming a product line or system family (the case here). CBSE, and reuse promise many advantages to system developers and users such as:

- Shortened development time, and reduced total cost, since systems are not developed from scratch.
- Facilitation of more standard, and reusable architectures, with a potential for learning.
- Separation of skills, since much complexity is packaged into specific frameworks.

- Fast access to new technology, since we can acquire components instead of developing them in-house.
- Improved reliability by shared components – etc.

These advantages are achieved in exchange for dependence on component providers, vague trust to new technology, and trade-offs for both functional requirements, and quality attributes.

Testing is the key method for dynamic verification (and validation) of a system. A system undergoes testing in different stages (unit testing, integration testing, system testing etc), and of different kinds (reliability testing, efficiency testing etc). Any deviation from the system's expected function is usually called for a *failure.* Failures observed by test groups or users are communicated to the developers by means of *failure reports*. A *fault* is a potential "flaw" in a hardware/software system that causes a failure. The term *error* is used both for execution of a "passive" fault leading to erroneous (vs. requirements) behavior or system state [6], or for any fault or failure that is a consequence of human activity [2]. Sometimes, the term *defect* is used instead of faults, errors or failures, not distinguishing between active or passive faults or human/machine origin of these. *Defect-density* or *fault-density* is then defined as the number of defects or faults divided by the size of a software module.

There are studies on the relation between fault-density and parameters such as software size, complexity, requirement volatility, software change history, or software development practices – see e.g. [1, 3, 5, 9, 13, 14]. Some studies report a relation between fault-density and component size, while others not. The possible relation can also be decreasing or increasing fault-density with growing size. Fenton et al. [3] have studied a large Ericsson telecom system, and did not observe any relation between fault-density and module size. When it comes to relation between the number of faults and module size, they report that size weakly correlates with the number of pre-release faults, but do not correlate with post-release faults. Ostrand et al. [14] have studied faults of 13 releases of an inventory tracking system at AT&T. In their study, fault-density slowly decreases with size, and files including high number of faults in one release, remain high-fault in later releases. They also observed higher fault-density for new files than for older files.

Malaiya and Denton [9] have analyzed several studies, and present interesting results. They assume that there are two mechanisms that give rise to faults. The first is how the project is partitioned into modules, and these faults decline as module size grows (because communication overhead, and interface faults are reduced). The other mechanism is related to how the modules are implemented, and here the number of faults increases with the module size. They combine these two models, and conclude that there is an "optimal" module size. For larger modules than the optimal size, fault-density increases with module size, while for smaller modules, fault-density decreases with module size (the economy of scale).

Graves et al. [5] have studied the history of change of 80 modules of a legacy system developed in C, and some application-specific languages to build a prediction model for future faults. The model that best fitted to their observations included the change history of modules (number of changes, length of changes, time elapsed since changes), while size and complexity metrics were not useful in such prediction. They also conclude that recent changes contributed the most to the fault potential.

There are few empirical studies on fault-density in the context of reuse. Melo et al. [10] describe a student experiment to assess the impact of reuse on software quality (and productivity) using eight medium-sized projects, and concluded that fault-density is reduced with reuse. In this experiment, reused artifacts are libraries such as C++ and GNU libraries; i.e. COTS and OSS artifacts. Another experiment that shows improvement in reliability with reuse of a domain-specific library is presented in [15].

High fault-density before delivery may be a good indicator of extensive testing rather than poor quality [3]. Therefore, fault-density cannot be used as a de-facto measure of quality, but remaining faults after testing will impact reliability. Thus it is equally important to assess the effectiveness of the testing phases, and build prediction models. Probably such a model includes different variables for different types of systems. Case studies are useful to identify the variables for such models, and to some extent to generalize the results.

3. The Ericsson context

3.1. System description

Our study covers components of a large-scale, distributed telecom system developed by Ericsson. We have assessed several hypotheses using historical data on defects, and changes of these systems that are either published by us, or will be published. This paper presents some of the results that are especially concerned with software reuse.

Figure 1 shows the high-level software architecture of the systems. This architecture is gradually developed to allow building systems in the same system family. This was a joint development effort across teams and organizations in Norway and Sweden for over a year, with much discussion and negotiation [12]. The systems are developed incrementally, and new features are added to each release of them. The two systems A and B in Figure 1 share the system platform, which is considered as a COTS component developed by another Ericsson organization. Components in the middleware, and business specific layers are shared between the systems, and are hereby called for *reused components* (reused in two distinct products and organizations, and not only across releases). Components in the application-specific layer are specific to applications, and are called for *non-reused components*.

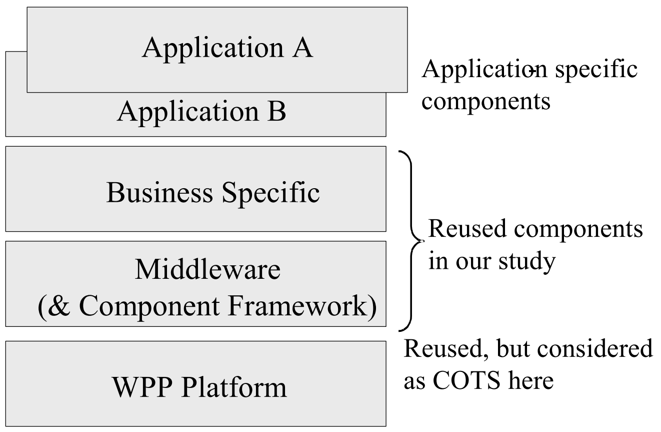

Application A
Application B
Application specific components
Business Specific
Middleware (& Component Framework)
Reused components in our study
WPP Platform
Reused, but considered as COTS here

Figure 1 High-level architecture of systems

The architecture is component-based, and all components in our study are built in-house. Several Ericsson organizations in different countries have been involved in development, integration, and testing of the systems. But what a component is in this discussion?

Each system is decomposed hierarchically into subsystems, blocks, units, and modules (source files). A *subsystem* presents the highest level of encapsulation used, and has formally defined (provided) interfaces in IDL (Interface Definition Language). It is a collection of blocks. A *block* has also formally defined (provided) interfaces in IDL, and is a collection of lower level (software) units. Subsystems, and blocks are considered as components in this study; i.e. high-level (subsystems) and lower-level (blocks) components. Since communication inside blocks are more informal, and may happen without going through an external interface, blocks are considered as the lowest-level components.

The systems' GUIs are programmed in Java, while business functionality is programmed in Erlang and C. Erlang is a functional language for programming concurrent, real-time, distributed fault-tolerant systems. We have data on defects and component size of 6 releases of one system (and several releases of other systems in the same system family releases). We present a detailed study of one of these releases in this paper. We obtained the same results with data from 2 other releases as well, but the data for this special release is more complete, and this release is the latest version of the system on the time of the study. The release in our study consisted of 470 KLOC (Kilo Lines of non-commented Code), where 64% is in Erlang, 26% in C, and the rest in other programming languages (Java, Perl, etc). Sometimes the term *equivalent code* is used for the size of systems developed in multiple programming languages. To calculate the "equivalent" size in C, we multiplied the software size in Erlang with 3.2, Java with 2.4, and IDL with 2.35, as the practice is in the organization. However, we found that other studies use other numbers. For example, Doug implemented 21 identical programs in C and Erlang, and reported an equivalent factor of 1.46 [16]. Based on the results of this study, we came to another factor (2.3) that must be further assessed. However, the results did not show any significant difference using pure LOC or equivalent ones.

All source code (including IDL files) is stored in a configuration management system (ClearCase). A product release contains a set of files with a specific label for the release in this system.

3.2. Trouble reports

When a defect is detected during integration testing, system testing or later in maintenance, a Trouble Report (TR) is written, and stored in a TR database using a web interface. Besides, if requirement engineering, or analysis and design of iteration *n* find defects in software delivered in iteration *n-1*, a TR will also be written. If a defect is reported multiple times, it reports problems observed due to the same fault, and is considered as a *duplicate*.

A TR contains the following fields: header with a number as identifier, date, product (system name), release, when the defect is detected (analysis and design, system test etc), severity, a defect code (coding, documentation, wrong design rule applied etc), assumed origin of the defect, estimated number of person-hours needed to correct the defect, identifier of another TR that this one is a duplicate of (if known), and a description. Three different severities are defined: A (most serious defects with highest priority that brings the system down or affects many users), B (affects a group of users or restarts some processes), and C (all other defects that do not cause any system outage). TRs are written for all types of defects (software, hardware, toolbox, and documentation), and there should be only one problem per TR.

All registered TRs are available as plain text files. We created a tool in C# that traversed all the text files, extracted all the existing fields, and created a summary text file. The summary was used to get an overview of the raw data set, and to decide which fields are relevant for the study. The exploration revealed a lot of inconsistencies in the TR database, e.g. fields are renamed several times, apparently from one release to the other. For example, a subsystem is stored as 'ABC' or 'abc' or 'ABC_101-27'. Another major weakness of the current defect reporting system is the difficulty to track defects to software modules without reading all the attached files (failure reports, notes from the testers, etc) or parsing the source code. Each TR has a field for software module, but this is only filled if the faulty module is known when the TR is initiated, and is not updated later. These inconsistencies show that data had hardly been systematically analyzed or used to a large extent before.

After selecting the fields of interest, another tool in C# read each TR text file, looked for the specified fields, and created a SQL insert statement. We verified the process by randomly selecting data entries, and cross checking them with the source data.

We inserted data from 13 000 TRs in a SQL database for 12 releases of systems. Around 3000 TRs were either duplicated or deleted. The release of system A in this study had 1953 TRs in the database, which are used for assessment of hypotheses in this paper. This release was in the maintenance phase on the date of this study (almost 8 months after delivery). TRs report both pre-delivery and post-delivery defects (from maintenance). 1539 TRs in our study were initiated pre-delivery (79%), while 414 TRs (21%) were post-delivery defects.

4. Research method and hypotheses

The overall research question in our study is the impact of reuse on software quality. To address this research question, we have to choose some attributes of software quality. Based on the literature search, and a pre-study of the available data, we chose to focus on defect-density, and stability of software components in the case study. There are inherently two limitations in this design:

1. Are defect-density and stability good indicators of software quality?
2. Can we generalize the results?

To answer the first question, we must assess whether defect-prone components stay defect-prone after release, and in several releases, and build a prediction model. This is not yet done.

The second limitation has two aspects: definition of the population, and limitations of case study research. Our data consists of non-random samples of components, and defect reports of a single product. Formal generalization is impossible without random sampling of a well-defined population. However, there are arguments for generalization on the background of cases [4]. The results may at least be generalized to other releases of the product under study, and products developed by the same company when the case is a probable one. On the other hand, if we find evidence that there is no co-variation between reuse, and quality attributes, the results could be a good example of a falsification case, which could be of interest when considering reuse in similar cases.

We chose to refine the research question in a number of hypotheses. A hypothesis is a statement believed to be true about the relation between one or more attributes of the *object of study,* and the *quality focus.* Choosing hypotheses has been both a top-down, and a bottom-up process. Some goal-oriented hypotheses and related metrics were chosen from the literature (top-down), to the extent that we had relevant data. In other cases, we pre-analyzed the available data to find tentative relations between data and possible research questions (bottom-up).

Table 1 presents 4 groups of hypotheses regarding reuse vs. defect-density and modification rate, and the alternative hypotheses for two of them; i.e. **H1,** and **H4**. For the other two groups of hypotheses, the null hypotheses state that there is no relation between the number of defects or defect-density, and component size. The alternative hypotheses are that there is a relation between the number of defects or defect-density with component size. Table 1 also shows an overview of the results. Section 5 presents the details of data analysis, and other observations.

Table 1 Research Hypotheses and results

HypId	Hypothesis Text	Result
H1	**H01:** Reused components have the same defect-density as non-reused ones.	Rejected
	HA1: Reused components have lower defect-density than non-reused ones.	Accepted
H2	**H02-1:** There is no relation between number of defects and component size for all components.	Not rejected
	H02-2: There is no relation between number of defects and component size for reused components.	Not rejected
	H02-3: There is no relation between number of defects and component size for non-reused components.	Rejected
H3	**H03-1:** There is no relation between defect-density and component size for all components.	Not rejected
	H03-2: There is no relation between defect-density and component size for reused components.	Not rejected
	H03-3: There is no relation between defect-density and component size for non-reused components.	Not rejected
H4	**H04:** Reused and non-reused components are equally modified.	Rejected
	HA4: Reused components are modified more than non-reused ones.	Rejected

5. Data analysis

We used Microsoft Excel and Minitab for data visualization, and statistical analysis. Statistical tests were selected based on the type of data (mostly on ratio scale). For more description of tests, see [11] and [17].

Most statistical tests return a *P-value* (the observed significance level), which gives the probability that the sample value is as large as the actually observed value if the null hypothesis (H0) is true. Usually, H0 is rejected if the P-value is *less* than a significance level (α) chosen by the observer. Historically, significance levels of 0.01, 0.05 and 0.1 are used because the statistical values related to them are found in tables. We present the P-values of the tests to let the reader decide whether to reject the null hypotheses, and give our conclusions as well.

The *t-test* is used to test the difference between two population means with small samples (typically less than 30). It assumes normal frequency distributions, but is resistant to deviations from normality, especially if the samples are of equal size. Variances can be equal or not. If the data departs greatly from normality, non-parametric tests such as *Wilcoxon test,* and *Mann-Whitney test* should be applied. Mann-Whitney test is the non-parametric alternative to the two-sample t-test, and tests the equality of two populations' medians (assumes independent samples, and almost equal variances).

Regression analysis helps to determine the extent to which the dependent variable varies as a function of one or more independent variables. The regression tool in Excel offers many options such as residual plots, results of an ANOVA test (Analysis of Variance), R^2, the adjusted R^2 (adjusted for the number of parameters in the model), and the significance of the observed regression line (P-value). R^2 and the adjusted R^2 show how much of the variation of the independent variable is explained with the variation of the dependent variable. Again it is up to the observer to interpret the results. We consider the correlation as low if the adjusted R^2 is less than 0.7.

Chi-square test is used to test whether the sample outcomes results from a given probability model. The inputs are the actual distribution of samples, and the expected distribution. Using Excel, the test returns a P-value that indicates the significance level of the difference between the actual, and expected distributions. The test is quite robust if the number of observations in each group is over 5.

5.1. H1: Reuse and defect-density

The quality focus is defect-density. We study the relation between component type (reused vs. non-reused), and defect-density.

H01: Reused components have the same defect-density as non-reused ones.
HA1: Reused components have lower defect-density than non-reused ones.

Results: Size of the release is almost 470 KLOC, where 240 KLOC is modified or new code (MKLOC= Modified KLOC). 61% of the code is from the reused components. Only 1519 TRs (from 1953 TRs) have registered a valid subsystem name, and 1063 TRs have registered a valid block name. We calculated defect-density using KLOC and MKLOC, and also using equivalent C-code. We do not present the results for equivalent C-code, but the conclusions were the same.

To compare the mean values of the two samples (reused, and non-reused components), we performed one-tail t-tests assuming zero difference in the means. However, the number of subsystems is low, which gave too few data points, and relatively high P-values. For example $P(T<t)$ one-tail=0.36 for #TRs/KLOC, which means that there is a probability of 36% that the observed difference is just coincidental. The same analysis on the block level gives results of statistical significance.

Table 2 No. of TRs for subsystems and blocks

Component	#TRs all	No. of Reused Comp.	%TRs Reused	No. of Non-reused Comp.	%TRs Non-reused
Subsystems	1519	9	44%	3	56%
Blocks	1063	29	41%	20	59%

Table 3 shows means, medians, and variances of defect-density. We tested the samples for normality, and the assumption of equal variances. The assumption of normality is violated for reused components and #TRs/KLOC, and the variances are unequal. For defect-density of modified code, distributions are not normal, but have almost equal variances. Table 4 shows results of the statistical tests. The t-test is applied since it is robust to the violation of normality, but a non-parametric test is also applied which has no assumption on distribution.

Table 3 Descriptive statistics for defect-density of blocks

Defect-density	Mean	Median	Variance
#TRs/KLOC, Reused	1.32	0.76	1.70
#TRs/KLOC, Non-Reused	3.01	2.44	4.39
#TRs/MKLOC, Reused	3.50	1.78	21.26
#TRs/MKLOC, Non-Reused	5.69	3.73	21.76

Table 4 Summary of the results of t-tests

P-values	t-test	Mann-Whitney
P(T<=t) one-tail [#TR/KLOC]	0.002	0.000
P(T<=t) one-tail [#TR/MKLOC]	0.055	0.020

The P-values in Table 4 are low (lower than 0.1), which means that the reused blocks have in average lower defect-density than the non-reused ones. We add that 46% of TRs have not registered any block name. Therefore the values for defect-density are not absolute, and they would be higher if all TRs had a valid subsystem or block name.

Table 5 shows the distribution of TRs over severity for blocks (2 of the blocks did not register the severity). The expected values may be calculated by multiplying '% of all' with the actual number; e.g. we can expect that 0.31*435=134.85 of TRs for reused blocks to be of severity A. As shown in Table 5, reused blocks have higher number of severity A defects than expected, while non-reused ones have lower number (167 compared with 190 expected). We performed a Chi-square test to evaluate whether the observed distribution is significantly different from the expected one. The returned P-value is 0.001; i.e. reused blocks have more defects with severity A (the highest priority defects) than expected from the total distribution. The same result is obtained if we perform the test with subsystems.

Table 5 TRs and severity classes for blocks

Severity	Reused	Non-Reused	% of all
A	160	16	31%
B	226	361	55%
C	49	98	14%
Sum	435	626	1

We also tested whether the distribution of TRs is different for pre- and post-delivery defects. The result is in the favor of reused blocks; i.e. they have significantly fewer defects after delivery than expected, with P-values equal to 0.003 and 0.002 for subsystems, and blocks.

5.2. H2 and H3: Reuse vs. component size and defects/defect-density

In this section, we study whether the number of defects or defect-density is correlated with the component size, and whether the result is different for reused, and non-reused components. We defined six null hypotheses in two groups in Table 1. The alternative hypotheses state that there is a relation between the variables of study.

Results: We first examine the relations graphically, and then perform regression analysis. A scatter plots with KLOC on the x-axis, and #TRs on the y-axis for blocks is shown in Figure 2, showing also regression lines, and polynomial functions of order 2 for reused and non-reused blocks. We have similar plots for MKLOC. The gradient of the regression line is higher for non-reused subsystems and blocks, indicating that non-reused blocks are more defect-prone. Table 6 shows a summary of the regression results.

Table 6 Regression results for #TRs and component size

	Subsystem	Block
Adjusted R–Square [KLOC]	0.631	0.491
Regression line, P-value [KLOC]	0.001	0.000
Adjusted R–Square [MKLOC]	0.643	0.590
Regression line, P-value[MKLOC]	0.001	0.000

A study of residual plots confirmed that points are evenly distributed on the both sides of the regression line, and thus the regression analysis is valuable. The P-values for the regression lines are low, meaning that the probability for a random correlation is very low. However, the adjusted R^2 values are also low (between 49-64%), which indicate a weak correlation.

We performed the same analysis, but this time with reused and non-reused components separately. The adjusted R^2 was low for reused components (less than 0.70). However, the regression analysis for non-reused components had higher adjusted R^2, as shown in Table 7. Results in Table 7 indicate that there is a relation between the size of the non-reused components, and #TRs. Plots also indicate that #TRs grow with the component size for this group.

Table 7 Regression results for #TRs and the component size, non-reused blocks

Adjusted R–Square [KLOC]	0.715
Adjusted R–Square [MKLOC]	0.633

The conclusion is that we don't reject **H02-1,** and **H02-2**. For non-reused components, we observe a relation between the number of TRs and the component size, and **H02-3** is therefore rejected. The same results are achieved using one-way ANOVA tests.

Referring to [9], we could explain the rejection of **H02-3** if non-reused blocks were larger than reused ones, but this is not true in our case. Reused blocks are in fact larger (as shown in Figure 2, and verified by statistical tests), and the result should be explained by other factors such as type of functionality or programming language. For reused blocks, Erlang is the dominant programming language, while C is dominant for non-reused blocks. We have studied the type of defects in Erlang and C units, and found that C units have more intra-component defects (defects within a component) than Erlang units, Therefore the number of defects can increase with component size. This needs further study.

Figure 3 shows a plot of defect-density, and component size for blocks. When we plot with defect-density instead of the number of defects, the points are scattered more, and there is no obvious relation between component size, and defect-density. Results for regression analysis between #TRs/KLOC, and size in KLOC for blocks and subsystems is shown in Table 8.

Table 8 Regression results for #TRs/KLOC and component size in KLOC

	Subsystem	**Block**
Adjusted R–Square [KLOC]	0.036	0.000
Regression line, P-value [KLOC]	0.553	0.455

The P-values for the regression lines are high, while the adjusted R^2 values are low, indicating no relation. Similar results were obtained when we performed the analysis for reused, and non-reused components separately. We conclude that there is no relation between defect-density and component size, and **H03-1**, **H03-2**, and **H03-3** are not rejected.

5.3. H4: Reuse and stability

Each release of the system adds some features to the previous release, and some bugs are fixed, and therefore the code is modified between releases. As reused components must fulfill requirements for two products, we may assume that they are modified more than non-reused components, and therefore are more fragile.

H04: Reused and non-reused components are equally modified.
HA4: Reused components are modified more than non-reused ones.

Results: We define MOD = Size of new or modified code/Total Size of the component. We calculated MOD, visualized the results in a scatter plot, and performed t-tests, and ANOVA to evaluate whether there is a significant difference between means of reused and non-reused components. Table 9 shows means, medians, and variances.

Table 9 Descriptive statistics for MOD of blocks

Defect-density	Mean	Median	Variance
MOD, Reused	0.43	0.43	1.87E-2
MOD, Non-Reused	0.57	0.60	2.04E-2

Our study showed that blocks are 49% modified totally; 43% for reused, and 57% for non-reused ones (with KLOC). The distribution is not normal for reused blocks, but variances are not significantly different. A scatter plot with KLOC on the x-axis, and MKLOC on the y-axis for blocks is shown in Figure 4. The gradient of the regression line is larger for non-reused blocks, indicating that they are modified more than reused ones. A two-tail t-test confirms that the difference in means is not zero, with P-value=0.001. A one-tail t-test for blocks assuming equal variances (we test for the hypothesis that reused blocks are modified more than non-reused ones), gives a P-value equal to 0.999. Results show that we can reject both **H04** and **HA4**, and conclude that non-reused components are more modified than reused ones, despite these being specific to one system. One explanation could be that non-reused components have more external interfaces than the reused ones. This must be further studied. We have data for MOD in earlier releases of the product, and it seems to be relative stable between releases.

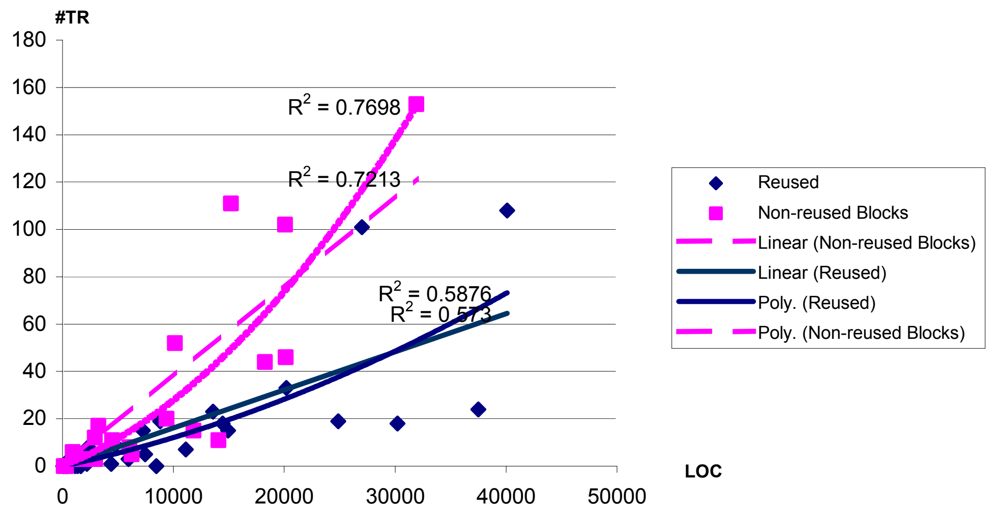


Figure 2 Relation between #TR and LOC for blocks

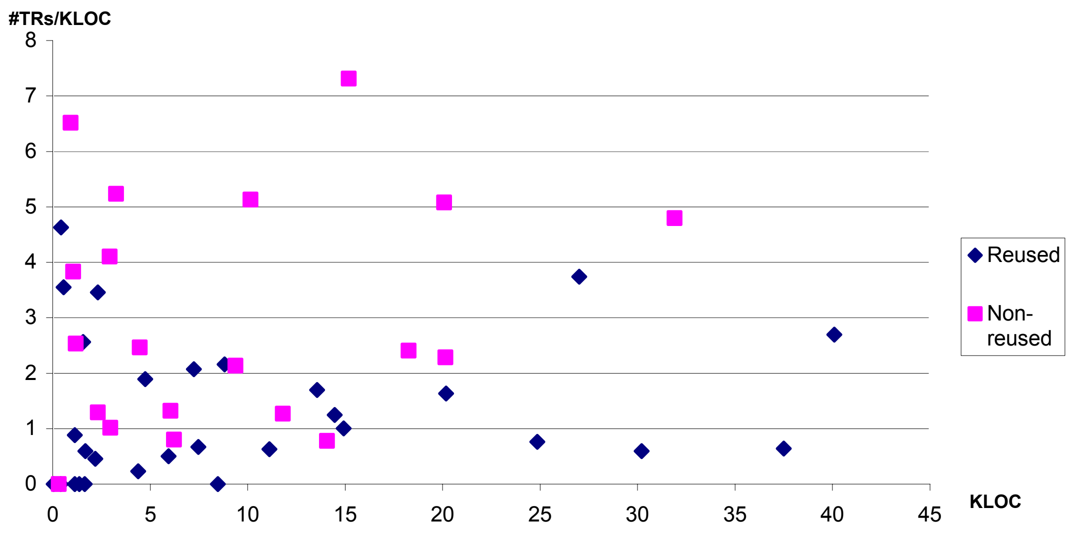


Figure 3 Relation between #TRs/KLOC and KLOC for blocks

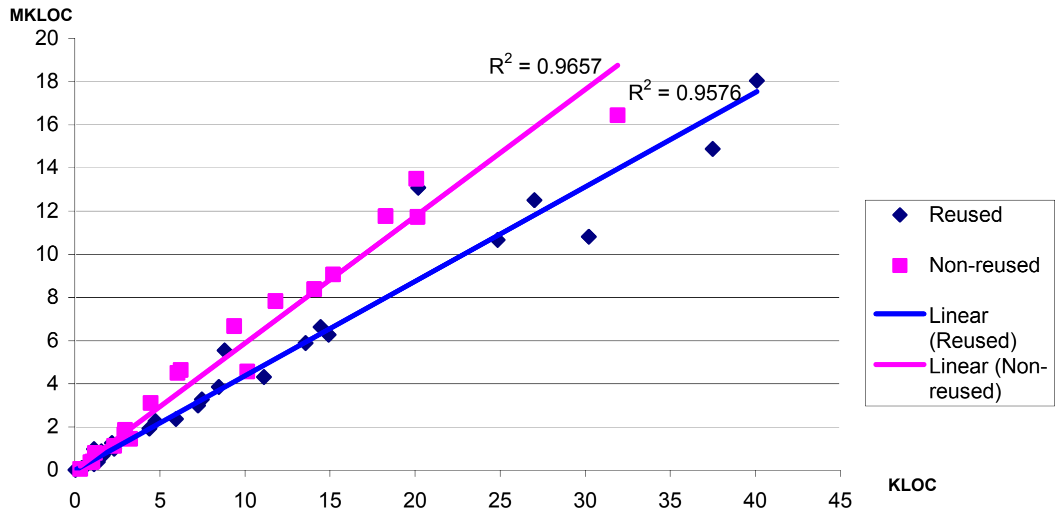


Figure 4 Relation between size of components and size of modified code for blocks

6. Summary and discussion of results

Wohlin et al. [17] describe four types of validity threats in empirical studies. In our study, these threats are:

Conclusion validity: A threat, or more a confounding factor, would be if reused and non-reused components had very different functionality and constraints. For example, non-reused components have user interfaces while some reused components handle other interfaces with complex protocols. Another threat would be if more experienced developers worked with one type of the components. This is not considered as a threat, since the components under study are developed within the same development unit, and by almost homogenous teams. A third threat is that TRs report defects mainly discovered during integration testing, system testing, and maintenance. We don't include data from inspections, and unit testing because this data is not in the same database.

Internal validity: Missing, inconsistent, or wrong data is a threat to internal validity-but mostly missing data. Sometimes gaps in data are systematically related to the behavior to be modeled, or to the nature of the problem. In our case, missing data is because of the process of reporting defects, which does not ask the tester to fill in the missing fields in the reports. This is not considered to be related to the nature of the problem or to introduce a systematic bias. Ways to handle missing data are e.g. mean substitution, regression substitution, or just trying with the existing data, and be aware of the lost efficiency of tests. We chose the last strategy because of two reasons. In some cases we could verify that the distribution of data is not significantly different if all or fractions of data are used. For example, the distribution of defects over severity classes for all data was almost the same as data for subsystems or blocks (which have missing points). The second reason is that our dataset is not too small for comparing the means or correlations. Some other statistical tests are more sensitive to missing data.

Construct validity: We use defect-density and stability as software quality indicators. The weaknesses of this assumption are discussed in Sections 2, and 4. Nevertheless these measures are used broadly in studies.

External validity: The external validity of all hypotheses is threatened by the fact that the entire data set is taken from one company. Our dataset consists of a non-random sample of defect reports (1953 from almost 10,000), and all components of a single release of one product. Two other releases are assessed with similar results. In Section 4, we discussed the possibility to generalize the results to other releases of the same product, to other products in the same company, and possibly to other companies in the same domain, where the case can be considered as a probable one. There are few published results from such large-scale products to compare with the results.

The remainder of this section discusses the results. Rejection of a null hypothesis does not mean that the inverse is accepted automatically, but we discuss when evidence for such conclusion exists.

H1: *Reuse and defect-density.* Our results showed that reused components have lower defect-density than non-reused ones (almost 50% less). The difference was less for modified code. Melo et al. [10] report fault-densities from 0.06 for components that are reused verbatim, 1.50 for slightly modified components, and 6.11 for completely new components in their experiment. They concluded also that reused components have lower fault-density. We observed that reused components had more severity A defects than expected from the total distribution, but fewer post-delivery defects. This could mean that defects of these components are given higher priority to fix.

H2: *Number of defects and component size.* We did not observe any significant relation between the number of defects, and component size for all the components as a group. We conclude there are other factors than size that may explain why certain components are more defect-prone. One factor may be whether the component is reused or not. When reused and non-reused components were analyzed separately, we did not observe any relation between size and the number of defects for reused components, while larger non-reused components had significantly higher number of defects, which may indicate that it is better to break these components down to smaller ones. Factors such as type of functionality or programming language may explain the result.

H3: *Defect-density and component size*. The plots, and regression analysis showed no relation between these two factors. The result is the same for all components, and for reused, and non-reused ones.

H4: *Reuse and stability*. Our results showed that reused components are in fact modified less than non-reused ones; i.e. when components are reused across several products, they don't get more fragile, although they should meet requirements from several systems. Stability is important in systems that are developed incrementally, and over several releases.

Based on **H1** and **H4**, we observe that packaging shared functionality into reusable components reduces defect-proneness and improves stability (thus decreasing the need for modifications). An internal survey of 9 developers in the same organization by us in spring 2002 indicated that developers consider reused components to be more reliable, and stable, in line with the quantitative results. These attributes may be interdependent as other studies show that modified code has more defects than old code [5, 10, 14]. The results of hypothesis testing cannot be used to build causal models, but rather be combined with other types of studies to discuss causes.

We did not observe a significant relation between defect-density, and component size. That is, defect-density cannot be used to predict the number of defects in a component, so other parameters should be studied.

The study also showed the weaknesses of the defect reporting system that has lead to inconsistencies and difficulties in analyzing, and presenting data. A better solution for quality managers (and researchers) would be if the defect reports had automatically been stored in a SQL database from the existing web interface. We may wonder if it is possible to develop a "standard", minimal metrics (TR schema), which all projects at Ericsson could use. Many interesting hypotheses on reuse, and various software properties were impossible to answer due to the lack of sufficient, and/or relevant data. On the other hand, amassing empirical data without any specific goals or with no post-processing is almost worse than not collecting any data!

7. Conclusions and future work

This study has "data-mined" defect reports, and associated data that had hardly been analyzed or used to a large extent before. We don't claim that the results are surprising. However, there are few published results on the impact of *reuse* on quality attributes in large industrial projects, so this study is a contribution in that context.

The hypotheses could be used to make a prediction model for future systems in the same environment or for maintaining the current system. For Ericsson, the results of this empirical study may be used to achieve better quality by identifying more defect-prone components (we have not presented the detailed results here), and by taking actions such as inspections or restructuring the components (e.g. split or merge to exploit economy of scale). Higher stability, and lower defect-density of reused components clearly show the industrial advantage of reuse. All these insights represent explicit knowledge based on own data, and thus important for deciding future approaches around reuse. Results can also be used as a baseline for comparison in future studies on software reuse.

8. Acknowledgements

The work is done in the context of the INCO project (INcremental and COmponent-based Software Development [7], a Norwegian R&D project in 2001-2004), as part of the first author's PhD study, and the MSc thesis of Killi and Schwarz [8] in spring 2003. We thank Ericsson for the opportunity to perform the study, and help to collect the data.

9. References

[1] Banker, R.D., Kemerer, C.F., ”Scale Economics in New Software Development”, *IEEE Trans. Software Engineering*, 15(10), 1989, pp. 1199-1205.

[2] Endres, A., Rombach, D., *A Handbook of Software and Systems Engineering; Empirical Observations, Laws and Theories*, Pearson Addison-Wesley, 2004.

[3] Fenton, N.E., Ohlsson, N., “Quantitative Analysis of Faults and Failures in a Complex Software System”, *IEEE Trans. Software Engineering*, 26(8), 2000, pp. 797-814.

[4] Flyvbjerg, B., *Rationalitet og Magt I- det konkretes videnskab*, Akademisk Forlag, Odense, Denmark, 1991.

[5] Graves, T.L., Karr, A.F., Marron, J.S., Siy, H., Predicting Fault Incidence using Software Change History. *IEEE Trans. Software Engineering*, 26(7): 653-661, July 2000.

[6] *IEEE Standard Glossary of Software Engineering Terminology*, IEEE Std 610.12, 1990.

[7] INCO project (INcremental and COmponent-based Software Development), http://www.ifi.uio.no/~isu/INCO/

[8] Killi, O.M., Schwarz, H., ”An Empirical Study of Quality Attributes of the GSN System at Ericsson”, MSc thesis, 109 pages, NTNU, June 2003, available on:http://www.idi.ntnu.no/grupper/su/su-diploma-2003/index.html

[9] Malaiya, K.Y., Denton, J., “Module Size Distribution and Defect Density”, *Proc. 11th International Symposium on Software Reliability Engineering- ISSRE'00*, 2000, pp. 62-71.

[10] Melo, W.L., Briand, L.C., Basili, V.R., “Measuring the Impact of Reuse on Quality and Productivity on Object-Oriented Systems”, Technical Report CS-TR-3395, University of Maryland, 1995, 16 pages.

[11] Mendenhall, W., Sincich, T., *Statistics for Engineering and the Sciences*, Prentice Hall, 1995.

[12] Mohagheghi, P., Conradi, R., “Experiences and Challenges in Evolution to a Product line”, *Proc. 5th International Workshop on Product Line Development- PFE 5*, 2003, Springer LNCS 3014, pp. 459-464,.

[13] Neufelder, A.M., “How to Measure the Impact of Specific Development Practices on Fielded Defect Density”, *Proc. 11th International Symposium on Software Reliability Engineering (ISSRE'00)*, 2000, pp. 148-160.

[14] Ostrand, T.J., Weyuker, E.J., "The Distribution of Faults in a Large Industrial Software System", *Proc. The International Symposium on Software Testing and Analysis (ISSTA'02)*, ACM SIGSOFT Software Engineering Notes, 27(4): 55 – 64, 2002.

[15] Succi, G., Benedicenti, L., Valerio, A., Vernazza, T., "Can Reuse Improve Reliability?", Fast abstract in *International Symposium on Software Reliability Engineering (ISSRE'98)*, 1998, available on http://www.chillarege.com/fastabstracts/issre98/98420.html

[16] The Great Computer Language Shootout, 2003, http://www.bagley.org/~doug/shootout

[17] Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A., *Experimentation in Software Engineering: An Introduction*, Kluwer Academic Publishers, 2000.

P9. A Study of Developer Attitude to Component Reuse in Three IT Companies

Jingyue Li[1], Reidar Conradi[1,3], Parastoo Mohagheghi[1,2,3], Odd Are Sæhle[1], Øivind Wang[1], Erlend Naalsund[1], and Ole Anders Walseth[1]

[1] Department of Computer and Information Science,
Norwegian University of Science and Technology (NTNU),
NO-7491 Trondheim, Norway
{jingyue, conradi}@idi.ntnu.no
[2] Ericsson Norway-Grimstad, Postuttak, NO-4898 Grimstad, Norway
{parastoo.mohagheghi}@ericsson.com
[3] Simula Research Laboratory, P.O.BOX 134, NO-1325 Lysker, Norway

Abstract

The paper describes an empirical study to investigate the state of practice and challenges concerning some key factors in reusing of in-house built components. It also studies the relationship between the companies' reuse level and these factors. We have collected research questions and hypotheses from a literature review and designed a questionnaire. 26 developers from three Norwegian companies filled in the questionnaire based on their experience and attitudes to component reuse and component-based development. Most component-based software engineering articles deal with COTS components, while components in our study are in-house built. The results show that challenges are the same in component related requirements (re)negotiation, component documentation and quality attributes specification. The results also show that informal communications between developers are very helpful to supplement the limitation of component documentation, and therefore should be given more attention. The results confirm that component repositories are not a key factor to successful component reuse.

1. Introduction

Systematic reuse is generally recognized as a key technology for improving software productivity and quality [17]. With the maturity of component technologies, more and more companies have reused their software (program) in the form of components. Component reuse consists of two separate but related processes. The first deals with analysis of the application domain and development of domain-related components, i.e. *development-for-reuse*. The second process is concerned with assembling software system from prefabricated components, i.e. *development-with-reuse*. These two processes are tightly related, especially in reusing in-house built components. The number of components and the ratio of reused components to total components will determine the reuse benefits (e.g. improved productivity and quality) [11][23].

To investigate the current state of practice and challenges for *development-with-reuse* in the IT industry, and to investigate the relationship between companies' reuse level and some key factors in reusing of in-house components, an empirical study was performed as part of two Norwegian R&D projects. These projects were SPIKE (Software Process Improvement based on Knowledge and Experience) [29] and INCO (INcremental and COmponent-based development) [30]. From the literature review, we defined several research questions and hypotheses. A questionnaire was designed to investigate these questions. Developers from three Norwegian IT companies filled in the questionnaire based on their experience and attitudes to component reuse.

From the results of the survey, we found some new challenges in component reuse and component-based development based on in-house built components. The results support some commonly held beliefs and contradict others.

As the sample size of current research is still small, this study cannot provide statistically significant tests on hypotheses, and is therefore a pre-study. Later studies will be undertaken with refined hypotheses and on a larger sample.

The reminder of the paper is structured as follows: Section 2 presents some general concepts. Section 3 describes the research approach. Section 4 presents the survey results. Section 5 gives a detail discussion on the survey results. Conclusion and future research are presented in Section 6.

2. Component reuse and component-based development

Software reuse can take many different forms, from ad-hoc to systematic [16]. In the broad definition of reuse, it includes reusing everything associated with software projects, such as procedures, knowledge, documentation, architecture, design and code. In our research, we focus on systematic reuse of software code. The code reuse literature has identified reuse practice and success factors through several case studies and surveys. A major reuse effort is the REBOOT (Reuse Based on Object-Oriented Techniques) consortium [25]. This effort was one of the early reuse programs that recognized the importance of not only the technical, but also the organizational aspects of reuse [18]. As more experience become available from industrial studies, non-technical factors, such as organization, processes, business drivers and human involvement, appeared to be at least as important as technological issues [15][19].

Following the success of the structured design and OO paradigms, component -based software development has emerged as the next revolution in software development [27]. More and more IT companies have started to reuse code by encapsulating it into components. Whitehead defines a component as: *A software component is a separable piece of executable software, which makes sense as a unit, and can interoperate with other components, within some supporting environment. The component is accessible only via its interface and is capable of use 'as-is', after any necessary installation and configuration procedures have been carried out* [28].

Component-based development is assumed to have many advantages. These include more effective management of complexity, reduced time to market, increased productivity, improved quality, a greater degree of consistency and a wider range of usability [4][13]. It also brings many challenges, because it involves various stakeholders and roles, such as component developers, application developers, and customers. Different stakeholders and roles have different concerns [3], and face different issues and risks [2][27].

Component-based development differs from traditional development, where the usual approach is for stakeholders to agree upon a set of requirements and then build a system that satisfies these requirements from scratch. Component-based development builds application by reusing existing components. Available components may not be able to satisfy all the requirements. Therefore, component-based projects must have flexibility in requirements, and must be ready to (re)negotiate the requirements with the customer. Moreover, components are intended to be used 'as-is'. If some additional functionality is required, 'glue-code' is needed to be built to meet the differences between the requirement and component functionality. Another important feature of component-based development is the strong focus on the quality attributes (such as reliability, performance, and security etc.) and related testing. A major effort has to be put into checking how components perform, how well they interact, and to make sure that they are indeed compatible. Components may be developed in-house, acquired as COTS (commercial-off-the-shelf) [3], or even as OSS (Open Source Software) [5]. Most current research on component-based software engineering focuses on COTS-based development. Because COTS users cannot access the source code and must rely on vendors to give technical support, COTS-based development is assumed to be more challenging. Therefore, there is little research on the challenges based on in-house built components.

3. Research approach

The difference between development based on in-house built components and development based on COTS is that the former is related very tightly with *development-for-reuse*. Component reuse is generally an incremental procedure. The company will build some reusable components in the beginning. In case of successful reuse, more and more code will be encapsulated into reusable components. The more reusable components are developed, the more complex will the development process be, and more support is required from the organization [8]. Our motivation is to investigate the relationship between companies' reuse level and some key factors in component-based development so that company with low reuse level can make necessary software process improvements when moving to a higher reuse level.

3.1 Research questions

To reuse in-house components successfully, developers must follow three basic steps [19]:

- Formulate the requirements in a way that supports retrieval of potentially useful reusable components.
- Understand the retrieved components.
- If the retrieved components are sufficiently 'close' to the needs at hand and are of sufficient quality, then adapt them.

From these steps, we selected several key factors. For step 1, we focus on the efficiency of component related requirements (re)negotiation and the value of component repository. For step 2, we study how knowledge about components can be transferred from a component provider to a component user. For step 3, our study focuses on definition and reasoning of quality attributes of components.

There is little research on the need for requirements (re)negotiation when components are built in-house. People assume that owning source code of in-house built components allows them to do any changes to meet the customers' requirements. However, components are intended to be used 'as-is', even it is built in-house. So, our first research question is:

RQ1. Does requirements (re)negotiation for in-house components really work as efficiently as people assume?

Crnkovic et al. have proposed that to successfully perform the component-based requirements (re)negotiation, a vast number of possible component candidates must be available, as well as tools for finding them [9]. Companies with a higher reuse level usually have more component candidates, more experience, and better experience than companies with a lower reuse level. So, our second research question is:

RQ2. Does the efficiency of component related requirements (re)negotiation increase with more in-house built components available?

To investigate this question, we formalized a null hypothesis H01 and an alternative hypothesis HA1 as follows:

H01. There is no relationship between the companies' reuse level and the efficiency of component related requirements (re)negotiation.

HA1. There is a positive relationship between the companies' reuse level and the efficiency of component related requirements (re)negotiation.

Concerning a component repository, Frakes claimed that it should not be given much attention, at least initially [12]. So, our third research question is:

RQ3. Does the value of component repository increase with more reusable components available?

To investigate this opinion more deeply, a null hypothesis H02 and an alternative hypothesis HA2 was proposed:

H02. There is no relationship between the companies' reuse level and the value of component repository.

HA2. There is a positive relationship between the companies' reuse level and the value of component repository.

A complete specification of a component should include its functional interface, quality characteristics, use cases, tests, etc. While current component-based technologies successfully manage functional interfaces, there is no satisfactory support for expressing quality parts of a component [9]. So, our fourth research question is:

RQ4. How can a component user acquire sufficient information about relevant components?

Berglund claimed that growing reusable software components will create a new problem, i.e. the *information-overload problem*. Therefore, learning which component to use and how to use them become the central part of software development [1]. Companies with a higher reuse level usually have more reusable components than companies with lower reuse level. So, our fifth research question is:

RQ5. Does the difficulty of component documentation and component knowledge management increase with increasing reuse level?

To study this question, we formalize null hypothesis H03 and alternative hypothesis HA3:

H03. There is no relationship between the companies' reuse level and developers' satisfaction with component documentation.

HA3. There is a negative relationship between the companies' reuse level and developer' satisfaction with component documentation.

One key issue in component-based development is *trust*, i.e. we want to build trustworthy systems out of parts for which we have only partial knowledge [7]. Current component technologies allow systems builders to plug components together, but contribute little to ensure how well they will play together or to fulfill certain quality properties. So, the sixth research question is:

RQ6. Do developers trust the quality specification of their in-house built components? If the answer is no, how can they solve this problem?

3.2 The questionnaire

The questionnaire included five parts. The questions in the first part were used to investigate the *reuse level* of the companies. The definition of *reuse level* in this study is the number of reused components vs. the number of total components in the organization. The other four parts were organized based on the four key factors. Each question in the questionnaire was used to study one or more research questions. The details of questions are showed in the following Table 1. The correspondences between the questions in the questionnaire and research questions are showed in Table 2. To increase the reliability of our survey, the questionnaire also included the definition of concepts used in the questionnaire, and the questions about the respondents' personal information.

3.3 Data collection

The study was performed in three Norwegian IT companies. Data collection was carried out by NTNU PhD and MSc students. Mohagheghi, Naalsund, and Walseth performed the first survey in Ericsson in 2002. In 2003, Li, Sæhle and Wang performed the survey reusing the core parts of the questionnaire in two other companies (i.e. EDB Business Consulting and Mogul Technology). We selected those three companies because they have experience on component reuse and would like to cooperate with NTNU in this research. The respondents are developers in these three companies. They answered the questionnaires separately. The questionnaires were filled in either by hand or electronically (as a Word file). The MSc students provided support with possible problems in answering the questionnaire.

Table 1 Questions in the questionnaire

Reuse level
Q1. What is the reuse level in your organization? Q2. To what extend do you feel affected by reuse in your work?
Component related requirements (re)negotiation
Q3. Are requirements often changed/ (re)negotiated in typical develop projects? Q4. Are requirements usually flexible in typical projects? Q5. Do the component related requirements (re)negotiation processes work efficiently in typical projects?
Value of component repository
Q6. Would the construction of a reuse repository be worthwhile?
Component understanding
Q7. Do you know the architecture of the components well? Q8. Do you know the interface of the components well? Q9. Do you know the design rules of the components well? Q10a. Is the existing design/code of reusable components sufficiently documented? Q10b. If the answer of Q10a is 'sometimes' or 'no', is this a problem? Q10c. If the answer of Q10a is 'sometimes' or 'no', what are the problems with the documentation? Q10d. If the answer of Q10a is 'sometimes' or 'no', how would you prefer the documentation? Q10e. What is your main source of information about reusable components during implementation? Q10f. How do you decide whether to reuse a component 'as-is', 'reuse with modification' or 'make a new one from scratch'?
Quality attributes specification of components
Q11. Are specifications for components' quality attributes well defined? Q12. Do you test components after modification for their quality attributes before integrating them with other components?

Table 2 Correspondence between Questions in the questionnaire and Research Questions

	RQ1	RQ2	RQ3	RQ4	RQ5	RQ6
Q1-Q2		X	X		X	
Q3-Q5	X	X				
Q6			X			
Q7-Q10f				X	X	
Q11-Q12						X

Below, we briefly characterize these three companies and respondents.

3.3.1 Companies

Ericsson Norway-Grimstad started a development project five years ago and has successfully developed two large-scale telecommunication systems based on the same architecture and many reusable components in cooperation with other Ericsson organization. Their two main applications share more than 60% of ca. 1M lines of code [22].

EDB Business Consulting in Trondheim (now Fundator) is an IT-consultant firm which helps its customers to utilize new technology. It started to build reusable components from 2001. They have built some reusable components based on the Microsoft .Net in their eCportal framework (i.e. a web-application framework) 1.0 & 2.0. These components have been successfully reused in their new e-commence applications.

Mogul Technology (now Kantega) in Trondheim has large customers in the Norwegian finance- and bank sector. The main responsibilities are development and maintenance of the customers' Internet bank application. The application was originally a monolithic system. After several years in production, the customer itself took initiative to reengineer the old system to a component-based solution based on EJB component model in 2002. At the time of the survey, some components have been created and reused in their new Internet bank system.

3.3.2 Respondents

There were 200 developers at Ericsson in Grimstad, where we sent out 10 questionnaires to developers in one development team and got 9 filled-in questionnaires back. There were 20 developers in EDB Business Consulting in Trondheim, and we gathered 10 filled-in questionnaires back out of 10. We distributed 10 questionnaires to 22 developers at Mogul Technology in Trondheim and got 7 back. Those developers were selected because their work was related to component reuse, and they could assign effort to participate in the survey. This is non-probability sampling, which is based on convenience. Most participants in this survey have a solid IT background. 6 of 26 respondents have MSc degree in computer science and all others have a bachelor degree in computer science or telecommunication. More that 80% of them have more than 5 years of programming experience. The details of their position and their experience in the current organization are summarized in the following Table 3.

4. Survey results

In this section, we summarize the result of the survey. All the following statistical analyses are based on valid answers, i.e. *Don't Know* answers are excluded. The statistical analysis tool we used is SPSS Version 11.0.

4.1 Different reuse level in these companies

First, we wanted to know the reuse level in those three companies. Q1 and Q2 were asked to get the answer based on developers' subjective opinion on this issue. The result of Q1 is showed in Fig. 1, and the result of Q2 is showed in Fig. 2. From Fig. 1 and Fig. 2, we can see that most developers in Ericsson think that the reuse level in their company is very high or high. Most developers in EDB regard the reuse level in their company is high or medium. Most developers in Mogul think that the reuse level in their company is medium or little.

4.2 Component related requirements (re)negotiation

Questions Q3-Q5 were asked to investigate RQ1.We can see that no respondents to Q3 believe that the requirements were never changed/ (re)negotiated. Only 8% of respondents to Q4 think the requirements of their typical project are not flexible. However, only 48% of respondents to Q5 think component related requirements (re)negotiation works well. To study RQ2 and test the hypothesis H01, the correlation between the reuse level and response to Q5 is studied. We assign ordinal values to Ericsson, EDB and Mogul to represent their different reuse levels based on the responses to Q1 and Q2 (Ericsson = 3, EDB = 2, Mogul = 1). We also assign ordinal value to the answer of Q5 (Yes = 3, Sometimes = 2, No =1). The result of correlation between them using one-tailed *Spearman Rank Correlation Coefficient* analysis is .112, and the significance is .306. This shows that there is no significant statistical relationship between the reuse level and the efficiency of component related requirements (re)negotiation.

Table 3 Background of the respondents

Company	**Position and working experience in the organization**
Ericsson Norway-Grimstad	2 system architects, 7 designers. 1 person has 13 years of experience 7 persons have experience from 2-5 years, 1 person has 9 months of experience,
EDB Business Consulting in Trondheim	1 project manager, 5 developers and 4 IT consultants. 1 person has 17 years of experience 8 persons have experience from 3-8 years, 1 person has 2 years of experience.
Mogul Technology in Trondheim	6 developers and 1 maintainer (previous developer). 1 person has 10 years of experience, 6 persons have experience from 2-5 years.

4.3 Value of component repository

From the answer of Q6, we found that 71% of respondents in Mogul and EDB regard constructing a component repository as worthwhile, against 57% in Ericsson. To study RQ3 and test hypothesis H02, the relationship between the answer of Q6 and the reuse level is studied. We use the same ordinal number mapping as previously. The result of correlation between them using one-tailed *Spearman Rank Correlation Coefficient* analysis is -.124, and significance is .297, which shows that there is no obvious relationship between them.

4.4 Component understanding

Questions Q7-Q10f were used to investigate RQ4. For Q7, Q8 and Q9, the results show that 67% of the respondents think the component structure is well understood, 61% say that the component interfaces are understood, and 63% regard the design rules of components are also well understood. But for the responses to question Q10a, no one thinks that the design/code of components is well documented, 73% think that they are sometimes well defined, and 27% believe that they are not well documented.

Furthermore, the answers to questions Q10b and Q10c indicate that 86% believe that insufficient component documentation is a problem, e.g. documentation is not complete, not updated, and difficult to understand, etc. From responses to Q10d and Q10f, we can see that the preferable way of documentation is web pages. Some of the developers' knowledge of how to use components comes from informal communication sources, for example, previous experience, suggestions from local experts, etc. To study RQ5 and test hypothesis H03, the association between reuse level and response to Q10a is studied. We use the same ordinal number mapping as previously. The result of correlation between them using one-tailed *Spearman Rank Correlation Coefficient* analysis is -.469, and significance is .014, which shows that there is a weak negative relationship between them. It means that the higher the companies' reuse level, the less satisfied a developer is with the component documentation.

4.5 Quality attributes of components

Question Q11 and Q12 were used to investigate RQ6. From the responses to these questions, we see that 70% of the participants regard the design criteria for quality requirements are not well defined, and 87% will test the quality attributes of components after component modification, before integrating them into the system.

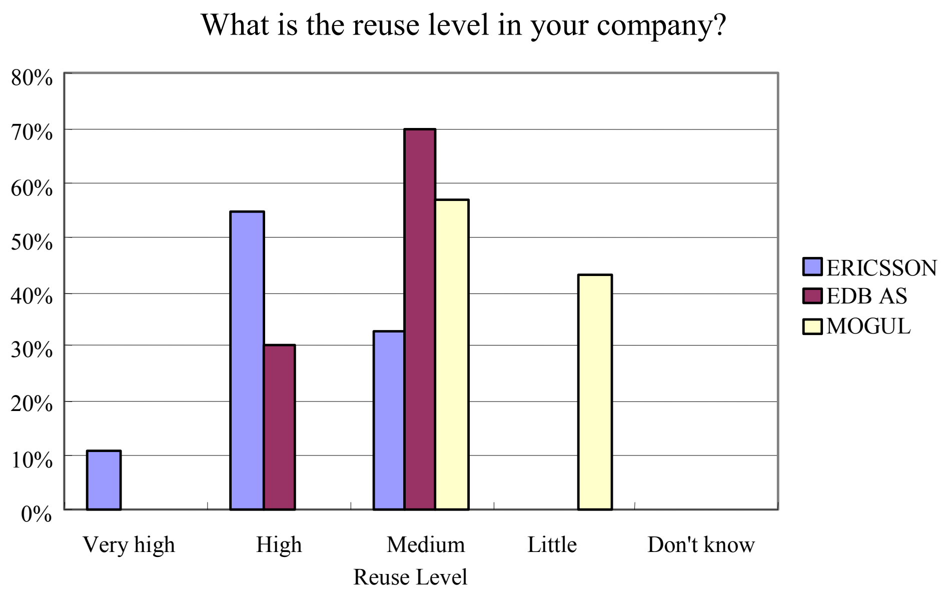


Figure 1 Result of the question "What is the reuse level in your company?"

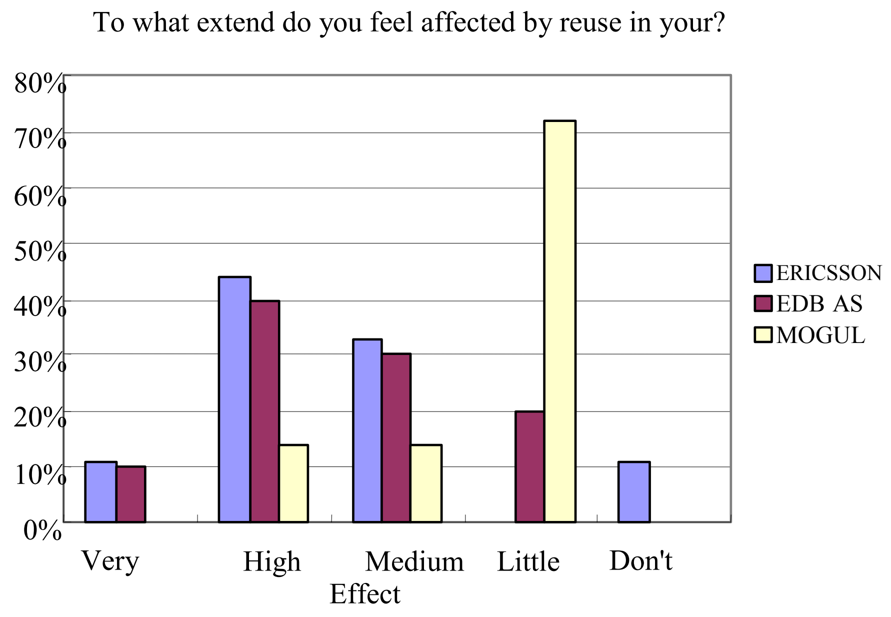


Figure 2 Result of the question "To what extend do you feel affected by reuse in your work?"

5. Discussions

Based on the result of the survey, we discuss our research questions and hypotheses, and discuss the limitations and threats to validity.

5.1 Component related requirements (re)negotiation

Much research focus on how to improve the efficiency of component related requirements (re)negotiation in COTS-based development [20][24][26]. The main reason is that people think the challenges in requirements (re)negotiation are due to the lack of access to source code, to timely vendor supports, or to the lack of engineering expertise to modify the integrated components [26]. In our case, the components are mostly built in-house. The above constrains on COTS components are not considered as challenges with built in-house components. From the responses to question Q3-Q5, we found that although 92% think that requirements of their typical projects are flexible, less than half think the component related requirements (re)negotiation in their typical projects works well.

Since components are intended to be used 'as-is', it is possible that an in-house reusable component meeting all the requirements will not be found. So, even though the components are built in-house, requirements (re)negotiation is necessary. For research question RQ1, we do not want to claim that the requirements (re)negotiation based on in-house components is more difficult than COTS-based components. We just want to emphasize that requirements (re)negotiation based on in-house components is also important but not efficient.

From the test result on H01, we cannot find a statistically significant relationship between the reuse level and the efficiency of component related requirements (re)negotiation. So, we cannot reject null hypothesis H01. Our conclusion to RQ2 is that when IT companies change from a low reuse level to a higher reuse level, they probably cannot expect that component-based requirements (re)negotiation becomes easier and more efficient.

5.2 Component repository

Some researchers have claimed that repository is important, but not sufficient for successful reuse [18][21]. Our data confirms that developers are positive, but not strongly positive to the value of component repository. So, this result gives future support to the previous conclusion.

From the test result on H02, we can see that there is no statistically significant relationship between developers' positive attitude to a component repository and reuse level. So, we cannot reject null hypothesis H02. Our conclusion to RQ3 is that companies are not expected to invest in a repository to increase reuse.

5.3 Component understanding

Transferring component knowledge from the component developer to the component user is critical for successful component reuse. The answers of Q7-Q9 show that most developers understand the components in detail. However, the answers of Q10a-Q10c show that no one believes that the components are well documented because the documents are either incomplete or not updated. So, our question is *"How can developers still understand the components without good documentation?"* From the answers to question Q10e and Q10f, we found that most developers got the knowledge of components from informal channels, such as previous experience and local experts. The most important feature of a component is the separation of its interface from its implementation. The component implementation is only visible through its interface. Moreover, current component documentation technologies cannot describe all the information the developer required, such as performance, reliability, and security etc. Therefore, informal knowledge transfer should be considered to supplement the insufficiency of formal component documentation and specification. This point was showed in other empirical studies as well [6][10]. For research question RQ4, we found that informal knowledge transfer is especially important in the component reuse. One possible solution is to have special interest groups or mailing lists for a components (or group of similar components) so that component users can share knowledge and experience of component usage.

From the test result on H03, we found a weak negative relationship between reuse level and developers' satisfaction with the documentation. We reject the null hypothesis H03 and accept the alternative hypothesis HA3, It means the higher the companies' reuse level, the less satisfied a developer is with components' documentation. Marcus et al. concluded that combine reuse education and training provided for staff with other reuse activity can lead to all the success of reuse [18]. Our conclusion to RQ5 implies that when a company moves from a low reuse level to high level, more effort should be spent on the component documentation and component knowledge management.

5.4 Quality attributes of components

Component-based development relies on the availability of high quality components to fill roles in a new intended system. When components are created or changed, we must ensure that they do not only fulfill the functional requirements, but also quality requirements. For research question RQ6, we found that most developers are not satisfied with the specification of components' quality attributes and therefore cannot use this information. Therefore, how can we model quality properties of both components and systems, and reason about them, particularly in the early stage of system development is still a key challenge in component-based development.

5.5 Threats to validity

We now discuss the possible validity threats in this study. We use the definition given by Judd et al. [14].

Construct validity In our case, the main construct issue applies to the variables chosen to characterize the data set. The independent variable, i.e. reuse level, is the most sensible one. The results of questions Q1 and Q2 give a qualitative and consistent value on this variable.

Internal validity A major threat to this validity is that we have not assessed the reliability of our measurement. Most variables are measured on a subjective ordinal scale. An important issue for future studies is to ensure the reliability and validity of all measurement. In this survey, we gave clearly specified concepts in the questionnaire and provided support to possible misunderstanding. These methods partly increased the reliability.

External validity The small sample size and lack of randomness in the choice of companies and respondents are threats to external validity. In general, most empirical studies in industry suffer from non-representative participation, since companies that voluntarily engage in systematic improvement activities must be assumed to be better-than-average.

Conclusion validity This study is still a pre-study. Future studies will be implemented to give more statistically significant results.

6. Conclusion and future work

This study has investigated challenges related to four key factors for development based on in-house components, especially in development-with-reuse. These factors are component related requirements (re)negotiation, component repository, component understanding and components' quality attribute specification. Another contribution is that we compared three IT companies with different reuse levels to study the possible trend and challenges in these factors when more and more code will be encapsulated as reusable components inside a company.

- For *component-based requirements (re)negotiation*, the results of research questions RQ1 and RQ2 show that requirements (re)negotiation for in-house built components is important but not efficient. The efficiency will probably not increase with higher reuse level.
- For the *component repository*, the results of research question RQ3 confirm that a component repository is not a key factor for successful reuse. Furthermore, the potential value of a component repository will probably not increase with higher reuse levels.

- For *component understanding,* the results of research questions RQ4 and RQ5 show that most developers are not satisfied with the component documentation, and developers' satisfaction with component documentation will probably decrease with higher reuse level. The results also show that informal communication channels, which developers can get necessary information about the components through, should be given more attention.
- For *components' quality attribute specification*, the result of research question RQ6 shows that developers still need to spend much effort on testing, as they cannot get relevant information from component specifications.

The main limitation of our survey is that it depends on the subjective attitudes of developers, and with few companies and participants involved. Later studies are planned to be undertaken with more precise quantitative methods and on more companies with more distinct reuse levels. Case studies will also be undertaken to follow the change of companies from lower reuse level to higher reuse level to future investigate our research questions.

7. Acknowledgements

This study is supported by the SPIKE and INCO projects. We thank the colleagues in these projects, and all the participants in the survey.

References

1. Erik Berglund: Writing for Adaptable Documentation. Proceedings of IEEE Professional Communication Society International Professional Communication Conference and Proceedings of the 18th ACM International Conference on Computer Documentation: Technology & Teamwork, Cambridge, Massachusetts, September (2000) 497–508.
2. Pearl Brereton: Component-Based System: A Classification of Issues. IEEE Computer, November (2000), 33(11): 54–62
3. Alan W. Brown: The Current State of CBSE. IEEE Software, September/October (1998) 37–46.
4. 4. Alan W. Brown: Large-Scale Component-Based Development. Prentice Hall, (2000).
5. Alan. W. Brown and Grady. Booch: Reusing Open-source Software and Practices: The Impact of Open-source on Commercial Vendors. Proceedings: Seventh International Conference on Software Reuse, Lecture Notes in Computer Science, Vol. 2319. Springer, (2002) 123–136.
6. Reidar Conradi, Tore Dybå: An Empirical Study on the Utility of Formal Routines to Transfer Knowledge and Experience. Proceedings of European Software Engineering Conference, Vienna, September (2001) 268–276.
7. Bill Councill and George T. Heineman: Component-Base Software Engineering and the Issue of Trust. Proceedings of the 22nd International Conference on Software Engineering, Limerick, Ireland, June (2000) 661–664.

8. Ivica Crnkovic and Magnus Larsson: A Case Study: Demands on Component-based Development. Proceedings of the 22nd International Conference on Software Engineering, Limerick, Ireland, June (2000) 21–31.

9. Ivica Crnkovic: Component-based Software Engineering - New Challenges in Software Development. Proceedings of 25th International Conference on Information Technology Interfaces, Cavtat, Croatia, June (2003) 9–18.

10. Torgeir Dingsøyr, Emil Røyrvik: An Empirical Study of an Informal Knowledge Repository in a Medium-Sized Software Consulting Company. Proceedings of 25th International Conference on Software Engineering, Portland, Oregon, USA, May (2003) 84–92.

11. W. B. Frakes: An Empirical Framework for Software Reuse Research. Proceedings of the Third Annual Reuse Workshop, Syracuse University, Syracuse, N.Y. (1990).

12. W. B. Frakes, C.J. Fox: Sixteen Questions about Software Reuse. Communication of the ACM, June (1995), 38(6): 75–87.

13. Ivar, Jacobson, Martin Griss, Patrick Jonsson: Software Reuse-Architecture, Process and Organization for Business Success. Addison Wesley Professional, (1997).

14. C.M. Judd, E.R. Smith, L.H. Kidder: Research Methods in Social Relations. Sixth edition, Holt Rinehart and Winston, (1991).

15. Y. Kim and E.A. Stohr: Software Reuse: Survey and Research Directions. Journal of Management Information System, (1998), 14(4): 113–147.

16. C. Kruger: Software Reuse. ACM Computing Surveys, (1992), 24(2): 131–183.

17. N. Y. Lee, C. R. Litecky: An Empirical Study on Software Reuse with Special Attention to Ada. IEEE Transactions on Software Engineering, September (1997), 23(9): 537–549.

18. Marcus A. Rothenberger, Kevin J. Dooley and Uday R. Kulkarni: Strategies for Software Reuse: A Principal Component Analysis of Reuse Practices. IEEE Transactions on Software Engineering, September (2003), 29(9): 825–837.

19. H. Mili, F. Mili, A. Mili: Reusing Software: Issues and Research Directions. IEEE Transactions on Software Engineering, June (1995), 21(6): 528–561.

20. M. Morisio, C.B. Seaman, A. T. Parra, V.R. Basili, S.E. Kraft, S.E. Condon: Investigating and Improving a COTS-Based Software Development Process. Proceeding of 22nd International Conference on Software Engineering, Limerick, Ireland, June (2000) 31–40.

21. Maurizio Morisio, Michel Ezran, Colin Tully: Success and Failure Factors in Software Reuse. IEEE Transactions on Software Engineering, April (2002), 28(4): 340–357.

22. Parastoo Mohagheghi and Reidar Conradi: Experiences with Certification of Reusable Components in the GSN Project in Ericsson, Norway. Proceedings of the 4th ICSE Workshop on Component-Based Software Engineering: Component Certification and System Prediction. Toronto, May (2001) 27–31.

23. Jeffrey S. Poulin: Measuring Software Reuse-Principles, Practices, and Economic Models. Addison-Wesley, (1997).

24. Vijay Sai: COTS Acquisition Evaluation Process: The Preacher's Practice. Proceedings of 2nd International Conference on COTS-based software systems, Lecture Notes in Computer Science, Vol. 2580. Springer, 2003, Ottawa, Canada, February (2003) 196–206.

25. Guttorm Sindre, Reidar Conradi, and Even-Andre Karlsson: The REBOOT Approach to Software Reuse. Journal of System Software, (1995), 30(3): 201–212.

26. Vu N. Tran, Dar-Biau Liu: Application of CBSE to Projects with Evolving Requirements- A Lesson-learned. Proceeding of the 6th Asia-Pacific Software Engineering Conference (APSEC' 99) Takamatsu, Japan, December (1999) 28–37.

27. Padmal Vitharana: Risks and Challenges of Component-based Software Development. Communications of the ACM, August (2003), 46(8): 67–72.

28. Katharine Whitehead: Component-Based Development: Principles and Planning for Business Systems. Addison-Wesley, (2002).

29. http://www.idi.ntnu.no/grupper/su/spike.html

30. http://www.ifi.uio.no/~isu/INCO/

P10. An Empirical Study of Software Change: Origin, Acceptance Rate, and Functionality vs. Quality Attributes

Parastoo Mohagheghi[1,2,3], *Reidar Conradi*[2,3]
[1]*Ericsson Norway-Grimstad, Postuttak, NO-4898 Grimstad, Norway*
[2]*Department of Computer and Information Science, NTNU, NO-7491 Trondheim, Norway*
[3]*Simula Research Laboratory, P.O.Box 134, NO-1325 Lysaker, Norway*
parastoo@idi.ntnu.no, *conradi@idi.ntnu.no*

Abstract

The paper presents results from an empirical study of change requests in four releases of a large-scale telecom system that is developed incrementally. The results show that earlier releases of the system are no longer evolved. Perfective changes to functionality and quality attributes are most common. Functionality is enhanced and improved in each release, while quality attributes are mostly improved, and have fewer changes in forms of new requirements. The share of adaptive/preventive changes is lower, but still not as low as reported in some previous studies. Data for corrective changes (defect fixing) have been reported by us in other studies. The project organization initiates most change requests, rather than customers or changing environments. The releases show an increasing tendency to accept change requests, which normally impact project plans. Changes related to functionality and quality attributes seem to have similar acceptance rates. We did not identify any significant difference between the change-proneness of reused and non-reused components.

1. Introduction

An important study object in empirical software engineering is *software maintenance*, being prevalent and thus costly in most software systems. Earlier studies have tried to study maintenance aspects, such as the ratio between different categories of maintenance activities, the origin of changes, or the impact of changes. These questions need updated answers given the emergence of new development approaches, such as incremental and iterative development. While incremental means that the project scope is (discovered and) covered in steps, iterative means that the developed assets are improved gradually during iterations. As many software projects are developed incrementally and iteratively, the subject of software change is relevant not only in the maintenance phase, but also in evolution between releases. Another aspect is the increasing use of component-based development (CBD) and software reuse, and the question that whether maintainability have improved.

This article describes the results of analyzing change requests (CRs) from four releases of a large telecom system developed by Ericsson over a three-years period. CRs cover any change in the requirements or assets from the time of requirement baseline.

We study some related factors, and assess five hypotheses, concerning the category of changes (perfective etc.), their origin, their acceptance rate, and their relation to reuse. We look at perfective, adaptive and preventive changes that characterize evolution. Corrective changes have been analyzed by us elsewhere [12].

The results show that earlier releases of the system are no longer evolved, and functionality is enhanced and improved in each release. Quality attributes are mostly improved, and have fewer changes in forms of new requirements. Most CRs are initiated internally by the project organization, and the acceptance rate of CRs has been increasing over time. When it comes to reuse, there was no significant difference between the change-proneness (number of CRs per KLOC) of reused and non-reused components. However, our earlier study of corrective changes shows that reused lower-level components are more stable (less modified code), and have fewer defects than are non-reused ones

The remainder of this paper is organized as follows. Section 2 includes a description of some related work. Section 3 presents the Ericsson context, and the available data. Section 4 describes the research method, and hypotheses. Section 5 presents the results, which are discussed further in Section 6. Section 7 contains the conclusion.

2. Related work

2.1. Concepts for software change

Lehman's first law of software evolution says that "an E-type program that is used must be continuously adapted else it becomes progressively less satisfactory" [7]. An E-type program is a software system that solves a problem in the real world. The growth of a system may be observed (measured) in many ways, for example by the amount of modified code between releases or per interval of time, the number of modules, the volume of change-logs, or even the number of system releases per time unit [14]. In other words, the granularity of change data varies. While lower level granularity provides most detailed information, it is more difficult to gather.

When a software system is still under development, requirements of the system may change, and requirement volatility may impact the project performance (e.g. schedule or cost overruns) or the quality of the software product (e.g. increasing defect-density). These impacts have been subject of empirical studies; see e.g. [5] [9] [18]. Other studies investigate modifications to software after it has gone into production; i.e. the maintenance phase, e.g. [10] [14]. The main challenge of empirical studies in this field is to have access to consistent records of software changes over time due to the longitudinal nature of the study.

Changes may be categorized as corrective, adaptive, perfective, or even preventive. *Corrective maintenance* refers to defect repair. *Adaptive maintenance* means adapting to a new environment or a new platform. *Perfective maintenance* is both used for implementing new or changed requirements, and for improving system performance; i.e. both functional enhancements and non-functional optimizations [16]. *Preventive maintenance* is sometimes used about internal restructuring or reengineering in order to ease later maintenance. Sometimes these terms are defined differently, making comparison of studies difficult. For example, Mockus et al. used adaptive maintenance to cover enhancements, and perfective to cover optimization [10]. Some others argue that it is maintenance when we correct errors, but it is *evolution* when we respond to other changes. Bennett and Rajlich distinguish between development, evolution and maintenance [4]. In their terminology, development lasts until a system is delivered to production. When a system is in production but still growing, it is in the evolution phase. They also provide a model for incremental evolution called for the *versioned staged model* shown in Figure 1, comparing to the simple staged model for evolution. In this model, after release of a version it is no longer evolved, only serviced. All new requirements will be placed on the new version.

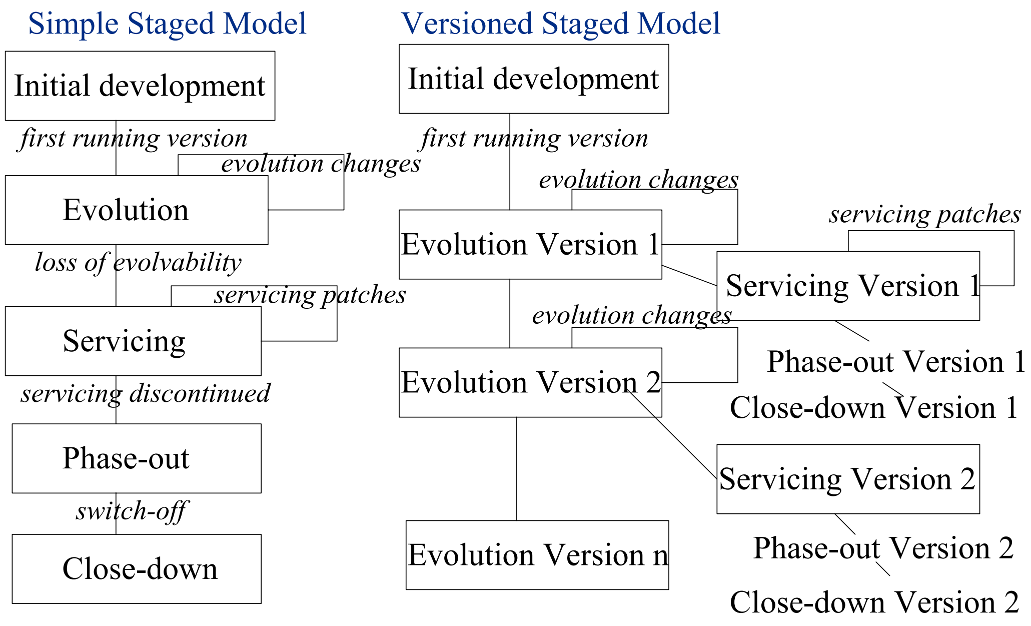


Figure 1 The incremental (versioned staged model) of software evolution from [4]

2.2. Results of previous studies

Damian et al. [5] describe results of a survey on the impact of improving the *pre-delivery* requirement engineering process on several factors. Zowghi and Nurmuliani [18] have similarly performed a survey among 430 software-developing companies in Australia on the impact of changing requirements on project performance regarding schedule and cost. The survey results show a negative correlation between the degree of requirement volatility, and both schedule and cost performance. Earlier work by Stark et al. [17] confirms this result.

One of the first studies on the distribution of *post-delivery* maintenance activities is reported in 1978 [8]. Based on the results of a survey among maintenance managers, Lientz et al. reported that 17.4% of the maintenance effort was categorized as corrective, 18.2% as adaptive, 60.3% as perfective, and 4.1% as other. Jørgensen [6] has observed that if the amount of corrective work is calculated based on interviews, it will be as twice as the actual work reported in such logs. I.e. the amount of corrective work may be exaggerated in interviews.

Schach et al. [14] have analyzed detailed data from 3 software products on the level of modules, and change-logs. The products were a 12 KLOC real-time product, a subset of Linux consisting of 17 kernel modules and 6506 versions, and GCC (GNU Compiler

Collection) consisting of nearly 850 KLOC. For these products, the distribution of maintenance categories was over 50% for corrective, 36-39% for perfective, and 2-4% for adaptive maintenance. In other words, the distribution is very different from the results reported by Lientz et al.

Mockus et al. [10] have used historical data of change requests of a multi-million-line telecom software system, and report the results both on the change-log level and on LOC (Lines of Code) added, deleted or modified. They report that adding new features (perfective changes) accounted for 45% of all changes, followed by corrective changes that accounted for 34%, while restructuring of the code accounted for 4% of changes (mostly preventive changes). Although comparisons of results are not easy between these two studies because of different categorizations, both indicate a large portion of corrective changes, as well as perfective changes for new requirements.

Algestam et al. [1] report a study in Ericsson of a large telecom system. Reusing components and a framework resulted in increased maintainability evaluated in cost of implementing change scenarios, improved testability, easier upgrades, and also increased performance. The impact of software reuse, especially exploiting COTS (Commercial-Off-The-Shelf) components, is studied e.g. in [2], and [3].

Organizations typically have a change management process to accommodate for requirement or artifact changes. In incremental development, each release may have changes in requirements or deliveries, and is undergoing an evolution phase. Evolution of a system should therefore be studied in two phases: during a release, and between successive releases. None of the studies above have taken the step to the versioned staged model shown in Figure 1, or separated these two phases, that may have different characteristics. The studies have also not separated functionality and quality attributes.

2.3. Research questions

Costs related to software evolution and maintenance activities can exceed development cost. Changes influence project performance and product quality. The impact of development approaches on software evolution and maintenance is also important to assess. Incremental and CBD are new approaches with few empirical studies on their impact on software evolution and maintenance. The following research questions are identified for this study:

RQ1: Do the majority of changes originate from external factors or from the project organization itself?

RQ2: Are changes mostly due to functional enhancements, or optimization of quality attributes?

RQ3: What is the impact of changes in terms of effort, size of modified code, or type of components?

RQ4: In which phase of the project are changes mainly introduced?

3. The Ericsson context

Ericsson in Grimstad-Norway has developed software for several releases of two large-scale telecom systems. The systems are characterized by high performance, high availability, scalability, frequent hardware and software upgrades, and distribution of software over multiple processors.

3.1. Overview of the products and the development process

The first system was originally developed to provide packet data capability to the GSM (Global System for Mobile communication) cellular network. A later recognition of common requirements with the forthcoming W-CDMA system (Wide-band Code Division Multiple Access) lead to reverse engineering of the original software architecture to identify reusable parts across the two systems. The two systems (or products) called for *A* and *B* in Figure 2, are developed incrementally, and new features are added to each release of the systems. The architecture is component-based, and all components in our study are built in-house. The higher-level components are subsystems (consisting of blocks, which are the lower level components) with almost 90 KLOC on the average. Both systems *A* and *B* contain top-level components respectively from Application *A* or *B*, as well as shared and reused components from the business-specific and middleware layers. At the bottom, there is a Wireless Packet Platform (WPP) serving as a pre-provided operating system.

The development process has evolved as well: The initial development process was a simple, internally developed one, describing the main phases of the lifecycle and the related roles and artifacts. After the first release, the organization decided to adapt the Rational Unified Process (RUP) [13]. Each release goes through 5-7 iterations. Multiple programming languages are used; Erlang and C are dominant, Java is used for GUIs, and Perl and other languages are used for minor parts. The size of each system (not including the system platform) is over 1000 NKLOC (Non-Commented Kilo Lines Of Code measured in equivalent C, see [12] for more details) in the last releases. Several hundred developers in different Ericsson organizations have been involved in developing, and testing the releases. Our data covers 4 releases of system *A*, where business-specific and middleware components are reused in two applications in release 4.

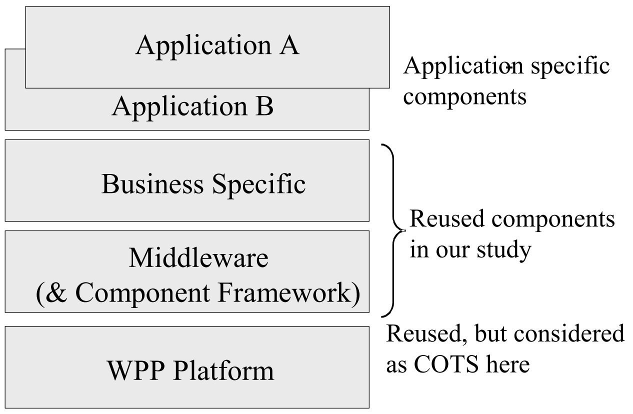

Application A
Application B
Application specific components
Business Specific
Middleware (& Component Framework)
Reused components in our study
WPP Platform
Reused, but considered as COTS here

Figure 2 High-level architecture of systems A and B

3.2. Change Requests (CRs)

RUP is an incremental software process with four phases: Inception, Elaboration, Construction, and Transition. Each phase concludes activities from several workflows; i.e. Requirement Management (RM), Analysis and Design (A&D), Implementation and Test, and may include one or several iterations. The original product, and project requirements for each release are stated in a textual document called the ARS

(Application Requirement Specification). Requirements in the ARS are refined iteratively during the inception and elaboration phases, resulting in artifacts such as use case models, use case specifications, supplementary specifications documents (for non-functional requirements), and statement of compliances towards standards that the system must fulfill. The ARS and the detailed set of the requirements are baselined at the end of initial iterations, and again at the end of the elaboration phase. Changes after this milestone are proposed through formalized CRs. Examples of proposed changes are:

- Add, change or delete functionality (perfective functional enhancements).
- Propose an improvement of a quality attribute (perfective quality optimizations).
- Implement a cost reduction.
- Solve an anticipated problem with major design impact (preventive).

CRs reflect coarse-grained changes that should be formally approved, and can significantly impact the contents of a release, cost or schedule. Small changes in implementation or documentation are done in each release by respective teams without issuing a CR. Figure 3 shows the phases and states in the CR handling process. CCB stands for the *Change Control Board*, who is responsible for taking decisions for approval or rejection of a CR. While the organization keeps a statistics over the number and states of the CRs, no systematic study of the CRs is done previous to this study.

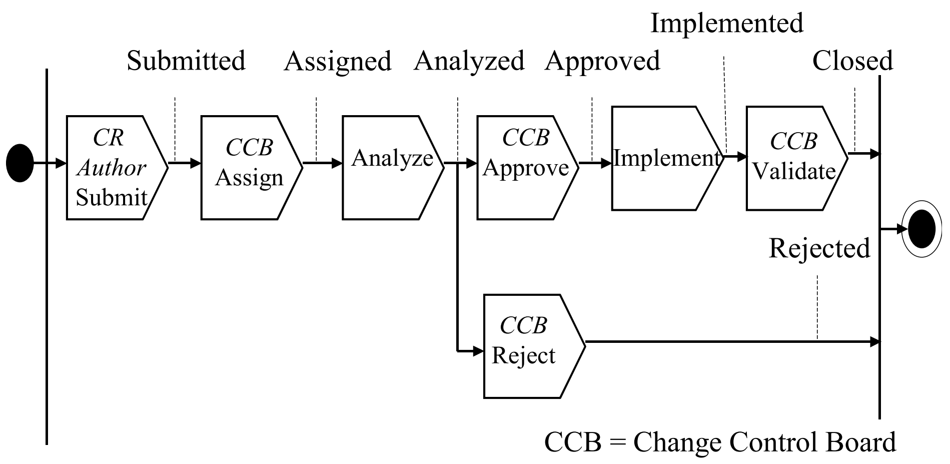


Figure 3 Flow of a CR and its states

CRs were at the beginning written in FrameMaker. Recently, these are written in MS-Word. The templates have changed several times, and not all fields are filled-in. The current template includes the following information: Title, revision history, baseline affected, documents/artifacts affected, description of the current situation and the proposed change, consequences of acceptance or rejection, and an estimate of the needed effort to implement the CR.

Note that CRs may be issued pre- or post-delivery. Our CRs fall in the category of perfective (enhancements and optimizations), adaptive (towards the WPP platform or standards), and preventive changes (reorganizations).

3.3. Change Request data

The first set of CRs was extracted from the version control system on January 2003 by a team of two NTNU students. This set included 165 CRs, issued from June 2000 to June 2002. 15 of these CRs handled in fact deviations to the process, and were omitted from the rest of this study. The status and a short summary of the CRs are given in html pages controlled by the CCB. We created a tool in C# that parsed the html pages, and inserted relevant fields in a Microsoft SQL database. For other fields that were not given in the html pages, the students read the CRs and inserted the data manually in the database. The first author checked these data later, and one or two fields in totally 24 CRs (of 150 such) were changed after this second check, being considered a small modification. A second set of CRs was extracted in November 2003 by the first author, and included 19 CRs issued from October 2002 to November 2003. These CRs were inserted manually in the database. Thus, we totally have 169 CRs for 4 releases of system *A*, as shown in Table 1. Release 3 did not include any new functionality, but was a new configuration in order to separate nodes in the system.

Table 1 Overview of all 169 CRs

	Rel. 1	Rel. 2	Rel.3	Rel. 4
Pre-delivery	10	37	4	99
Post-delivery	0	0	0	19
SUM	10	37	4	118

The number of CRs has increased dramatically as the product evolves from release 1 to 4. This increase is partly because the CR handling process has matured over time. For instance, some changes of release 1 were handled informally. However, because of the growing complexity of the releases, it is not unexpected that there would be more changes to the requirements or products over time, and the time frame in which a product is “under evolution” increases. We notice that evolution of releases 1-3 has stopped. These releases were delivered to the market, put in the servicing phase, and will be phased out after a while. Release 4 still evolved at the time of study as new CRs were issued. We also studied the date of initiation of CRs, which showed that most CRs in each release are initiated in a short time after requirement baselining.

Data on estimated cost or needed effort is not used in the study since we don’t have data on actual cost or effort. Data on effected components is coarse-grained as discussed later. Otherwise, the data set is considered to be reliable for the study.

We have described results of a study on corrective maintenance (Trouble Reports or TRs) in [12]. The data for that study covered TRs for these 4 releases until January 2003. We mention that the number of TRs were 6 for release 1, 602 for release 2, 61 for release 3, and 1953 for release 4. Again, not all TRs for release 1 were stored in this database. We note the same increase in the number of TRs as for CRs as shown in Table 1.

4. The research method and hypotheses

We tested five hypotheses on the available data. Choosing hypotheses has been both a top-down, and a bottom-up process. Some goal-oriented hypotheses were chosen from the literature (top-down), to the extent that we had relevant data. In other cases, we pre-analyzed the available data to find tentative relations between data and possible research questions (bottom-up) as in an exploratory research. Table 2 shows the hypotheses, their relations to research questions (RQ) defined in Section 2.3, and their grouping.

Table 2 The five research hypotheses

Hyp. group	Hyp. Id	Hyp. Text	RQ
Origin	H01 HA1	Pre-implementation, and post-implementation CRs have equal proportions. Most CRs are for post-implementation changes.	4
	H02 HA2	Quality attributes, and functionality have equal proportions of CRs. Most CRs are due to quality attributes, rather than to functionality.	2
	H03 HA3	Customers and changing environments initiate as many CRs as the project organization. Customers and changing environments initiate most changes.	1
Acceptance	H04 HA4	CRs that are accepted, and CRs that are rejected have equal proportions. Most CRs are accepted.	3
Reuse-CBD	H05 HA5	Reused and application components are equally change-prone. Application components are more change-prone than are reused ones.	3

A short description of background for each hypothesis is given below. Apart from assessing the five hypotheses, we will study some relationships between these, e.g. what class of CRs are mostly accepted or rejected.

H01-HA1: CRs have a field that indicates whether a CR specifies a change in requirements (new, modified, or removed requirement as stated in the ARS or other requirement specification documents) *before* implementation, or a change to the product or documentation *after* a requirement is first implemented and verified. Some CRs have left this information out, and are instead classified by us. **H01** states that the proportions of CRs for requirement changes, and CRs for modifications of the product are equal. The alternative hypothesis **HA1** states that most changes are post-implementation changes to the product. As the organization use much effort in the CR handling process, it is important to assess whether this effort is because of unstable requirements, or iterative improvement of solutions.

H02-HA2: CRs may also be categorized on whether they deal with functionality or with quality (non-functional) attributes. This information is extracted from the description and the consequences of approval or rejection. Earlier studies do not differ properly between these two sub-categories of perfective changes. The practice is that functional requirements are specified well, and thus changes in those would be more obvious. **H02** states equal proportion, while the alternative hypothesis **HA2** states that most changes are related to quality attributes, rather than to functionality, in line with HA1.

H03-HA3: CRs may be initiated internally by the project organization in order to improve or enhance the product, or externally by the customers or due to changing environments (external factors). Damian et al. [5] write that changes in requirements often arise from external events originating outside the organization, such as unpredictable market conditions or customer demands. **H03** states that there is no difference between proportions of CRs in these two groups. **HA3** states that the Domain's claim is true.

H04-HA4: Waterfall development requires stable requirements, while incremental approaches are more open to changes. We want to assess the stability of requirements, and the product. As we don't have data on the actual impact of CRs in terms of modified Lines of Code in some normalized form, it is difficult to assess the absolute impact. Therefore we chose to study the share of CRs being accepted or rejected. **H04** states that the proportions are equal, while **HA4** states that most CRs are accepted. If most CRs get accepted and implemented, the organization should be prepared to account for additional resources to handle and implement these CRs.

H05-HA5: We define change-proneness as #CRs/KLOC. Components in the business-specific and middleware layers were reused in two systems in release 4. **H05** states that there is no difference in change-proneness of these components. The alternative hypothesis would be that one component type is more change-prone than is the other. As application components are "customer-close", we may assume that these are more change-prone, as stated in **HA5**.

5. Data analysis and assessment of hypotheses

Table 3 shows a summary of the assessment of hypotheses. Here *NR* stands for *Not Rejected*, while *R* means *Rejected*. The remainder of this section describes the detailed results.

Table 3 Assessments of hypotheses

Hyp. Id	Result	Conclusion
H01	NR	The difference between proportions of CRs issued before or after implementation is not significant.
H02	R	Most perfective changes are due to quality attributes, not functionality.
H03	NR	Most CRs are originated inside the organization.
H04	R	Most CRs are accepted and implemented.
H05	NR	Reused and application components are equally change-prone.

We used Microsoft Excel and Minitab in statistical tests. The confidence level is 95% in all tests, which means that we reject the null hypotheses if the observed significance level (P-value) is less than 5%. The 5% significance level is the default value in most tools, and in practice much higher P-values may be accepted. We therefore present the distributions, and the P-values to let the reader decide, as well as presenting our conclusions.

5.1. H01-H03: Origin of CRs

Table 4 below shows a classification of CRs for all four releases. We see that most changes are optimizations of solutions, followed by new requirements after baseline. We could also compare the share of requirement changes (47.3%) before implementation, to the share of later modifications in solutions or documentation (52.1%). We tested whether the proportions are equal vs. greater proportion of CRs for modifications (one proportion test in Minitab for proportion=0.5 vs. proportion>0.5). The P-value is 0.322; i.e., it is 32% possible that the observed difference is by chance. The conclusion is that we cannot reject **H01**.

Table 4 Distribution of 169 CRs over pre- and post-implementation changes

	New Req.	Modified Req.	Removed Req.	Modified Solution	Modified Doc.	Other
No	46	25	9	70	18	1
%	27.2	14.8	5.3	41.4	10.7	0.6

Table 5 shows the distribution of CRs over evolution categories, and a more detailed distribution over CR-focus or *reason*. Note in Table 5, that the sum of numbers is 187 (and the sum of percentages is over 100%), as 18 of 169 CRs have indicated *two* reasons for requesting the change. Also note that preventive changes to improve file structure or to reduce dependencies between software modules and components may later impact quality attributes such as maintainability. For the systems in study, there is great emphasis on quality attributes, reflected in the large number of CRs for this group.

For perfective CRs, the proportion of functional and quality attributes CRs are 35% (40/114) and 65% (74/114). We performed a one proportion test in Minitab that gives a P-value of 0.01, which means that we can be 99% sure that the difference is significant, and may reject **H02** in favor of **HA2**.

The contents of Tables 4, and 5 are combined in Figure 4. Perfective functional CRs have almost equal distribution between new requirements, and modified solutions. Perfective quality attributes and preventive CRs are mostly modified solutions. Adaptive CRs are mostly new or modified requirements.

23 of 169 of the CRs are issued because of customer demands. If we exclude these CRs, and the 35 CRs due to adaptive changes, the overwhelming group (111 of 169) is still CRs that originate inside the project organization to enhance or optimize the products. The one proportion test in Minitab shows that the proportion of CRs due to external factors (customers and changing environments) is 34%, and the P-value is 1.000. Hence the proportion of CRs due to external factors is definitely lower than is the other group, opposed to **HA3**.

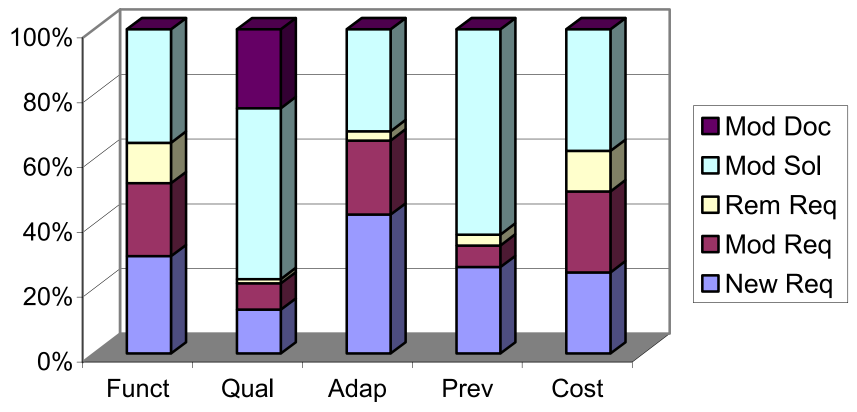


Figure 4 Evolution categories vs. requirements or solutions

Table 5 Distribution of CRs over maintenance categories and CR-focus

Evolution category	**CR-focus (*reason*)**	**No.**	**Accepted**	**Examples**
Perfective/ Functional	Functionality	40	26	Business or middleware functions
Perfective/ Quality Attributes SUM = 74	Performance	29	16	Storage, throughput
	Documentation	21	13	Understandability, customer documents
	Availability	11	7	Increasing up-time
	Testability/ Maintain-ability	11	6	Remote testing, monitoring alarms
	Security	2	1	Protecting contents
Adaptive SUM = 35	Standards	18	13	Compliance, new standards
	Interfaces	2	1	External interfaces
	WPP-upgrades	13	7	Change the platform
	WPP-adaptation	2	2	Adapting code to WPP changes
Preventive SUM = 30	System	19	7	Builds, configuration
	Re-structuring	11	7	Models, dependencies between entities, file structure
Other	Cost	8	3	Saving money/effort
TOTAL SUM		187		

5.2. H04: Acceptance rates of CRs

Figure 5 shows the acceptance rates of the CRs as of November 2003. 99 CRs were *accepted* (approved, implemented, or closed), while 68 CRs are *rejected* (including those cancelled). Performing a one-proportion test gives a P-value of 0.015, which means that the difference is significant. Hence, **H04** is rejected in favor of **HA4**.

Figure 6 shows the distribution of CR classes and accepted vs. rejected states. All CRs that requested to remove a requirement are accepted, while the group that has the highest rejection rate is new requirements.

We performed Chi-Square tests to study whether there is any relation between CR categories as defined in Table 5, and acceptance rates. The P-value of the test is 0.253 (DF=4), which indicated no relation. However, it is interesting to note that the maximum acceptance rate is for those with CR-focus Standards (72%), while the minimum rate is for those with System and Cost (38% both). The others vary between 50 and 60%. As our system should comply with international standards to be competitive and to inter-work with systems from other telecom operators, it is not surprising that changes due to Standards are mostly accepted. However, the low acceptance rate for Cost is surprising. The number of associated CRs is only 8, although many other CRs also impact cost (for example CRs that ask for removal of requirements). We cannot conclude otherwise that (low) cost is not a strong enough reason to accept a CR by itself.

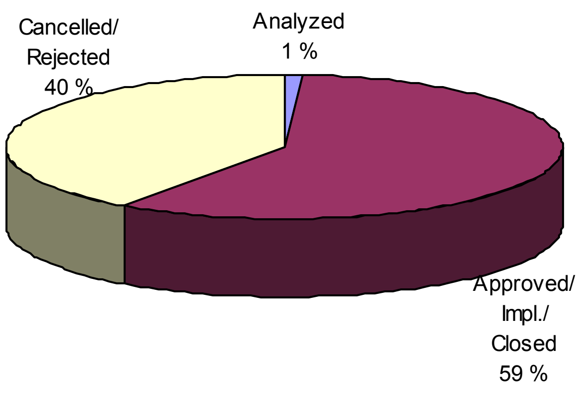


Figure 5 Acceptance rate of 169 CRs

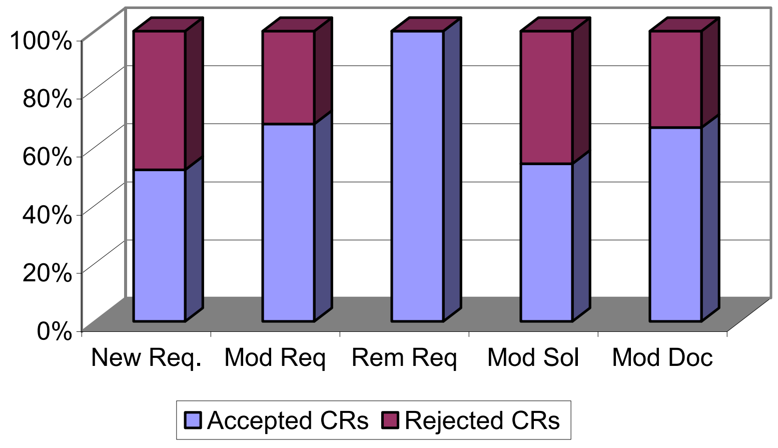


Figure 6 Acceptance rates and CR classes

Finally, we want to analyze whether the four releases vary significantly in acceptance rates of the CRs. Release 3 only has four CRs and hence cannot contribute to any significant conclusion. The overall results show that the acceptance rates of CRs have been increasing over the lifetime of the product as shown in Figure 7.

The organization has already studied *requirement volatility* in high-level requirements (those stated in the ARS), i.e. if they change after baseline. While this rate is 10% for Release 1, it is almost 30% for Release 3. I.e. both results indicate that the product is getting more change-prone over releases, or more changes are allowed.

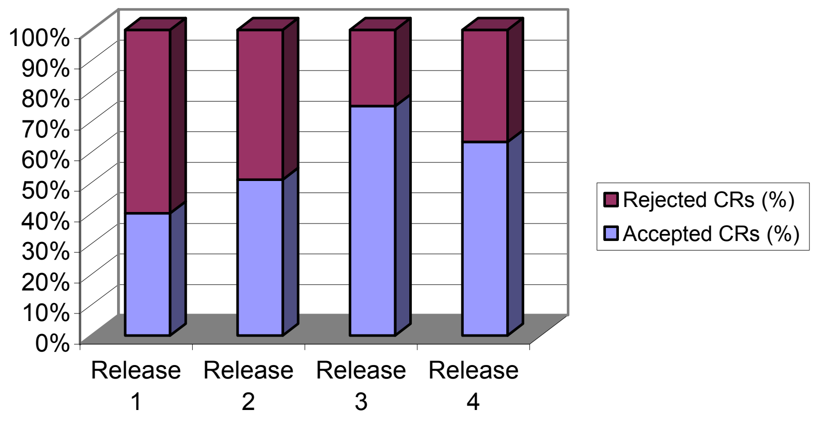


Figure 7 Relative distribution of CR states over the product releases (total of 169 CRs)

5.3. H05: Software reuse and CBD

Reuse of these components started from release 4, when development of system *B* started. Regrettably, only 81 of 118 CRs of release 4 have registered athe affected component name in the CR. Besides, CRs only register higher-level components, i.e. subsystems that consist of several related blocks.

System *A* consists of 3 application subsystems, 4 subsystems in the business-specific later, and 6 subsystems in the middleware layer (thus 10 reusable subsystems) in this release. Figure 8 shows the distribution of #CRs per KLOC for subsystems. We have one outlier, which is a small subsystem, handling configuration and tools, and which is left out from the statistical test. We performed a two-tailed t-test, which showed no significant difference in means for reused vs. non-reused components; i.e. P(T<=t) two-tail was 0.61, and **H05** can not be rejected.

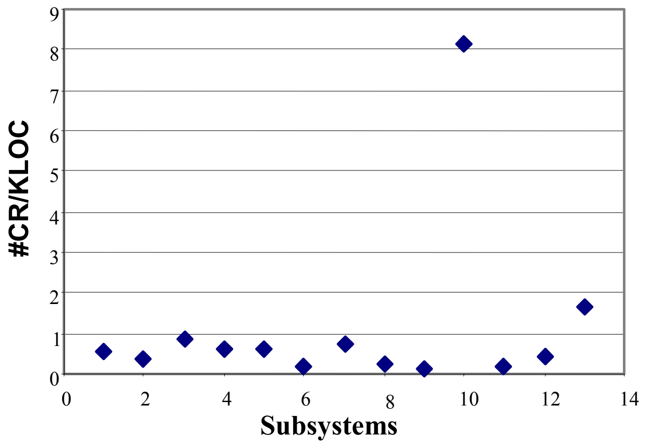


Figure 8 #CRs/KLOC for all 13 subsystems

6. Discussion

We comment each of the five hypotheses below.

H01: Most CRs are issued in order to optimize, and modify the product or documentation, rather than changing the requirements, but the difference is not significant. We need further analyses of the CRs to conclude whether the organization could save some effort by better quality assurance of the solutions.

H02: Although CRs issued to change, enhance or remove functionality account for the largest single group of CRs, quality attributes' related CRs are the largest as a group. This result highlights that that these attributes are optimized over time, and these improvements will have great impact of the evolution of the products.

H03: The results show that the project organization initiates most CRs for enhancing or optimizing the product.

H04: The results show that most CRs are accepted, especially those that request modification of a requirement or documentation, or removal of a requirement. New requirements need resources for implementation, and modification of previous solutions may be considered as too time-consuming compared to the perceived benefits, and therefore are more rejected. There was no significant difference in acceptance rates of the functional vs. quality attributes CRs. One interesting result is that the acceptance rate has been increasing in releases, and this may impact the precision of plans. The organization has already realized that planning precision has been decreasing. In [11] we described that incremental development opens for (more) changes in requirements, and this study demonstrates this.

H05: We could not observe any significant difference between reused, and non-reused components in number of CRs per KLOC. We have shown in [12] that reused components (as blocks) are more stable in terms of volume of code modified between releases, and more reliable in the terms of the number of Trouble Reports per KLOC. Together, these results quantify the benefits of reuse.

The study raises some interesting questions as well: Does the organization take the perhaps costly decision to baseline requirements too early, while the product still undergoes dramatic evolution? Could the number of changes be predicted for future releases using #CRs/KLOC from earlier releases?

Lastly, We have identified the following validity threats:

Construct validity: Most data categories are taken from the literature, and represent well-known study concepts. We used maintenance categories for all changes during development *after* requirement baseline. Previous works study changes post-delivery in the maintenance phase.

Internal validity: The biggest threat is that we ignore many CRs with no subsystems given in **H5**.

External validity: The study object is a large telecom system during three years of development. The results should be relevant and valid for similar systems and organizations, but not e.g. for web-based systems with very high change rates.

Conclusion validity: In **H5**, we have too little data caused by coarse-granular subsystems, . Otherwise, the data material is sufficient to draw valid conclusions.

7. Conclusion and future work

We defined 5 research questions in Section 2.3, and related them to the research hypotheses in Table 2. The results of the analyses are used to answer these:

RQ1 & RQ2 (origin): Most changes originate from the project organization in order to improve quality, and enhance functionality. The share of the first group is higher. The practice indicates iterative realization, and improvement of quality attributes, but functionality is also improved in a lower degree.

RQ3 (impact): CRs are not supplied with the actual cost of implementing the changes, only an estimate. However, we found that most CRs are accepted, and the acceptance rate can have impact on the project plans in terms of decreasing planning precision.

RQ4 (phase): Most CRs are issued pre-delivery, and especially in the short time right after requirement baseline. CRs are issued both before and after implementation of requirements.

The study gives insight into evolution as shown in Figure 1: Quality attributes and functionality are iteratively improved between releases reflecting in the number of CRs to modify solutions, in addition to corrective maintenance. Each release also undergoes changes mostly in form of new or modified requirements that are adaptive, or functional. It also shows that evolution of earlier releases has stopped as expected. The organization should notice the increased number and acceptance rate of CRs, which require extra resources to handle, and implement these.

The study's contribution is in the empirical evaluation of the intention (categories and goals), origin of changes, and of the distribution between functional and non-functional (quality) requirements/attributes in a large-scale project over time. It also extends the concept of software change to the development and evolution phases when the system is developed incrementally and iteratively.

We also have data on the original requirements in each release, and plan to analyze these to increase our understanding on software change as reflected in requirement evolution between releases.

Acknowledgements

The work was done in the context of INCO (INcremental and COmponent-based Software Development), a Norwegian R&D project in 2001-2004 [15], and as part of the first author's PhD study. H. Schwarz and O.M. Killi gathered the first set of CRs during their work on their joint master thesis in spring 2003. We thank them for the effort. We also thank Ericsson in Grimstad for the opportunity to perform this study.

References

[1] Algestam, H., Offesson, M., Lundberg, L.: Using Components to Increase Maintainability in a Large Telecommunication System. *Proc. 9th International Asia-Pacific Software Engineering Conference* (APSEC'02), 2002, pp. 65-73.

[2] Baldassarre, M.T., Bianchi, A., Caivano, D., Visaggio, C.A., Stefanizzi, M.: Towards a Maintenance Process that Reduces Software Quality Degradation Thanks to Full Reuse. *Proc. 8th IEEE Workshop on Empirical Studies of Software Maintenance* (WESS'02), 2002, 5 p.

[3] Basili, V.R: Viewing Maintenance as Reuse-Oriented Software Development. *IEEE Software*, 7(1): 19-25, Jan. 1990.

[4] Bennett, K.H., Rajlich, V.: Software Maintenance and Evolution: a Roadmap. In *ICSE'2000* - Future of Software Engineering, Limerick, 2000, pp. 73-87.

[5] Damian, D., Chisan, J., Vaidyanathasamy, L., Pal, Y.: An Industrial Case Study of the Impact of Requirements Engineering on Downstream Development. *Proc. IEEE International Symposium on Empirical Software Engineering* (ISESE'03), 2003, pp. 40-49.

[6] Jørgensen, M.: The Quality of Questionnaire Based Software Maintenance Studies, *ACM SIGSOFT* - Software Engineering Notes, 1995, 20(1): 71-73.

[7] Lehman, M.M.: Laws of Software Evolution Revisited. In Carlo Montangero (Ed.): Proc. European Workshop on Software Process Technology (EWSPT96), *Springer LNCS 1149,* 1996, pp. 108-124.

[8] Lientz, B.P., Swanson, E.B., Tompkins, G.E.: Characteristics of Application Software Maintenance. *Communications of the ACM*, 21(6): 466-471, June 1978.

[9] Malaiya, Y., Denton, J.: Requirements Volatility and Defect Density. *Proc. 10th IEEE International Symposium on Software Reliability Engineering* (ISSRE'99), 1999, pp. 285-294.

[10] Mockus, A., Votta, L.G.: Identifying Reasons for Software Changes Using Historical Databases. *Proc. IEEE Int. Conference on Software Maintenance* (ICSM'00), 2000, pp. 120-130.

[11] Mohagheghi, P., Conradi, R.: Using Empirical Studies to Assess Software Development Approaches and Measurement Programs. *Proc. 2nd Workshop in Workshop Series on Empirical Software Engineering* (WSESE'03), 2003, pp. 65-76.

[12] Mohagheghi P., Conradi R., Killi, O.M., Schwarz, H.: An Empirical Study of Software Reuse vs. Defect-Density and Stability, Accepted for ICSE'2004, 10 p.

[13] Rational Inc. www.rational.com

[14] Schach, S.R., Jin, B., Yu, L., Heller, G.Z., Offutt, J.: Determining the Distribution of Maintenance Categories: Survey versus Measurement. *Empirical Software Engineering: An International Journal*, 8(4): 351-365, Dec. 2003.

[15] INCO project: http://www.ifi.uio.no/~isu/INCO/

[16] Sommerville, I.: *Software Engineering*. 6th Ed., Addison-Wesley, 2001.

[17] Stark, G., Skillicorn, A., Ameele, R.: An Examination of the Effects of Requirement Changes on Software Releases. *CROSSTALK - The Journal of Defence Software Engineering,* Dec. 1998, pp. 11-16.

[18] Zowghi, D., Nurmuliani, N.: A Study of the Impact of Requirements Volatility on Software Project Performance. *Proc. 9th International Asia-Pacific Software Engineering Conference* (APSEC'02), 2002, pp. 3-11.

P11. Exploring Industrial Data Repositories: Where Software Development Approaches Meet

Parastoo Mohagheghi, Reidar Conradi
Department of Computer and Information Science, NTNU, NO-7491 Trondheim, Norway
parastoo@idi.ntnu.no, conradi@idi.ntnu.no

Abstract

Lots of data are gathered during the lifetime of a product or project in different data repositories that may be part of a measurement program or not. Analyzing this data is useful in exploring relations, verifying hypotheses or theories, and in evaluating and improving companies' data collection systems. The paper presents a method for exploring industrial data repositories in empirical research and describes experiences from three cases of exploring data repositories of a large-scale telecom system: A study of defect reports, a study of change requests, and a study of effort. The system in study is developed incrementally, software is reused in a product line approach, and the architecture is component-based. One main challenge is the integration of the results of studies with one another and with theory. We discuss that the challenges of integration especially arise when development approaches meet one another, while metrics and measurement programs do not. In order to develop advanced theories on the relations between development approaches and their impacts, measurement programs should be updated to collect some basic data that meets all the development approaches. A set of metrics for incremental, reuse-, and component-based development is identified.

Keywords: Data repositories, data mining, metrics, component-based development, incremental development, reuse.

1. Introduction

Exploring industrial data repositories for valuable information has been performed for many decades and the fields of *Data Mining* and *Exploratory Data Analysis* (EDA) have grown to become own branches of computer science. With the growing rate of empirical studies in software engineering and the gained approval of such studies for assessing development approaches and verifying theories, exploring data collected in industrial data repositories is more often performed, standing alongside other empirical methods. The goals of such studies can be exploratory (finding relations or distributions), confirmatory (verifying relations or theories), or used in triangulation for putting different sources of information against each other. Data repositories are also used in searching for design patterns, user interaction patterns, or reengineering legacy systems. For companies, the studies are useful to give insight into their collected data and to assess internal measurement programs and data collection systems. The focus of this paper is on data that can be used to assess quality of software or software development processes.

We present three empirical studies of exploring data repositories of a large telecom system developed by an Ericsson organization in Grimstad-Norway. These repositories contained defect reports, change requests, and effort reports for several releases. We also used data from the configuration management system on software size. Data for 3 years of development is collected in 2003 and 2004. The goals of the studies were to: a) quantitatively assess hypotheses related to reuse and quality metrics such as defect-density and stability of software components, b) explore the origin of software changes, and c) adapt an estimation method for incremental development of software. We describe steps in exploring data repositories, the role of literature search in the process, and the importance of relating hypotheses to one another and to a theory or model. We describe the challenges of integrating the results of these studies. The first challenge is the physical challenge since data is stored in different data repositories and in multiple formats. The second challenge is related to the conceptual integration of results for comparing and combining these in order to build theories. We discuss that problems in combining results especially arise when development approaches meet one another, while metrics are not defined to do so. In this case, incremental, use-case driven, reuse, product line, and component-based development approaches are used in parallel. We propose therefore to define metrics in a way that we can collect data to assess each approach, the combinations of these, and their impacts on one another.

The remainder of this paper is organized as follows. Section 2 discusses research methods, the role of exploring industrial data repositories in empirical research, and steps in such a study. Section 3 presents the studies performed in Ericsson, while Section 4 summarizes the research challenges. Section 5 presents metrics for a combination of development approaches. The paper is concluded in Section 6.

2. Exploring industrial data repositories in empirical research

2.1. Research classifications

Cooper et al. classify research design using 8 descriptors. One of the descriptors is the degree to which the research question has been crystallized, which divides research into *exploratory* and *formal* research [Coop01]. The objective of an exploratory study is to develop research questions or hypotheses and is loosely structured. The goal of a formal research is to test the hypotheses or answer the research questions.

Empirical research is research based on the scientific paradigm of observation, reflection, and experimentation. Empirical studies may be exploratory or formal as any other research. Empirical studies vary in scope, degree of control that the researcher has, and the risk associated with such studies. Wohlin et al. classify empirical strategies in three categories [Wohl00]: *surveys*, *case studies,* and *experiments*. Yin extends research strategies to five, adding *archival analysis* and *history* to research strategies [Yin02]. He does not provide further description of these strategies, except for defining archival analysis most suitable for exploratory studies, while history analysis is proposed for explanatory studies (answering how and why questions). Zelkowitz et al. classify validation methods as *observational*, *historical*, and *controlled*, which can be referred as research methods as well [Zelk98]. Wohlin et al. also divide empirical research into being *quantitative* (quantifying a relation) or *qualitative* (handling other data than numbers; i.e. texts, pictures, interview results, etc). A theory or even a hypothesis should be studied by a combination of methods. For example, the Conjecture 9 in [Endr04] says, "learning is best accelerated by a combination of controlled experiments and case studies".

Coop et al. define data mining as "the process of discovering knowledge from databases stored in data marts or data warehouses [Coop01]. The purpose is to identify valid, novel, useful, and ultimately understandable patterns in data. It is a step in the evolution from business data to information". They add, "data mining tools perform exploratory and confirmatory statistical analyses to discover and validate relationships". When data is stored in repositories with little or no facilities for mining with data mining tools, other research methods should be applied.

2.2. Role of exploring industrial data repositories in empirical research

With *industrial data repositories*, we mean contents of defect reporting systems, source control systems, or any other data repository containing information on a software product or a software project. This is data that is gathered during the lifetime of a product or project and may be part of a measurement program or not. Some of this data is stored in databases that have facilities for search or mining, while others are not.

Zelkowitz et al. define examining data from completed projects as a type of *historical study* [Zelk98]. Using Yin's terminology, it is classified either as archival analysis or history. We mean that this is a quantitative technique where the results should be combined with other studies of both types in order to understand the practice or to develop theories.

As the fields of Software Process Improvement (SPI) and empirical research have matured, these communities have increasingly focused on gathering data consciously, according to defined goals. This is best reflected in the Goal-Question-Metric (GQM) paradigm developed first by Basili [Basi94]. It states that data collection should proceed in a top-down rather than a bottom-up fashion. However, some reasons why bottom-up studies are useful are:

1. There is a gap between the state of the art (best theories) and the state of the practice (current practices). Therefore, most data gathered in companies' repositories are not collected following the GQM paradigm.
2. Many projects have been running for a while without having improvement programs and may later want to start one. The projects want to assess the usefulness of the data that is already collected and to relate data to goals (reverse GQM).
3. Even if a company has a measurement program with defined goals and metrics, these programs need improvements from bottom-up studies.

Exploring industrial data repositories can be part of an exploratory (identifying relations or trends in data) or formal (confirmatory; validate theories on other data that the theories were built on) empirical research; e.g. in order to study new tools, techniques or development approaches. It may be used in *triangulation* as well; i.e. setting different sources of information against each other.

Exploring industrial data repositories may be relatively cheap to perform since data is already collected. It has no risks for the company for interfering with on-going activities. Sometimes extra effort is needed to process the data and insert it in a powerful database. An important aspect is the ethical one; i.e. having the permission to perform such studies in companies and publish the results. The limitations are that the quality of the gathered data is sometimes questionable, data needs cleaning or normalization and other types of preparation before it may be used, and the hypotheses are limited to the available data. Limitations have impact on validity of the results. For example:

- Missing data can reduce the power of statistical tests in hypotheses testing.
- Generalization of results from single studies needs a clear definition of population. Some researchers mean that generalization based on single studies is possible if the context is well packaged and the case is carefully selected [Flyv91].

2.3. Steps in exploring industrial data repositories

Figure 1 shows the main steps in our research. A description of each step is given below.

The *theoretical phase* of the study starts either with a defined hypothesis or theory to assess, or some research or management question to answer. We emphasize the role of literature research or other secondary data analysis in the process. With such a study, possible results will be integrated into the total body of knowledge; i.e. not stay stand-alone and without any connection to a model or theory.

The *preparation phase* consists of a pre-study of data and definition of hypotheses or theory for the context (the particular product, project, and environment). The researcher must decide whether to use the entire data or a sample of it. After the data set is selected, it should be explored visually or numerically for trends or patterns. EDA techniques are also used in the exploring. Most EDA techniques are graphical such as plotting the raw data, with the means and standard deviations etc. Together with the pre-study of data, tools and statistical techniques for the analysis should be selected. Results of the preparation phase may invoke further need for literature search or refinement of research questions.

The *execution phase* consists of steps of a data mining process as described in [Coop01]. The data is formally sampled if necessary and fully explored. Data may need modification, e.g. clustering, data reduction, or transformation. Cooper et al. call the next step for modeling, which uses modeling techniques in data mining (neural networks, decision trees etc.). In the last step of the execution phase, hypotheses or theories should be assessed or research questions should be answered. Finally the results and the context are packaged and reported in the *conclusion phase*.

Very much like GQM, there is a hierarchy of goals, questions and metrics in Figure 1. But there is also a feedback loop between preparation and theoretical phases, due to the impact of the bottom-up approach. Questions may be redefined or hypotheses may be dropped if we do not data to assess them. However, there is no control of treatments, although the study may be applied to contemporary events as well.

There are several interesting examples of successful use of industrial databases for developing theories; e.g. Lehman developed the laws of software evolution by studying release-based evolution of a limited number of systems [Lehm96].

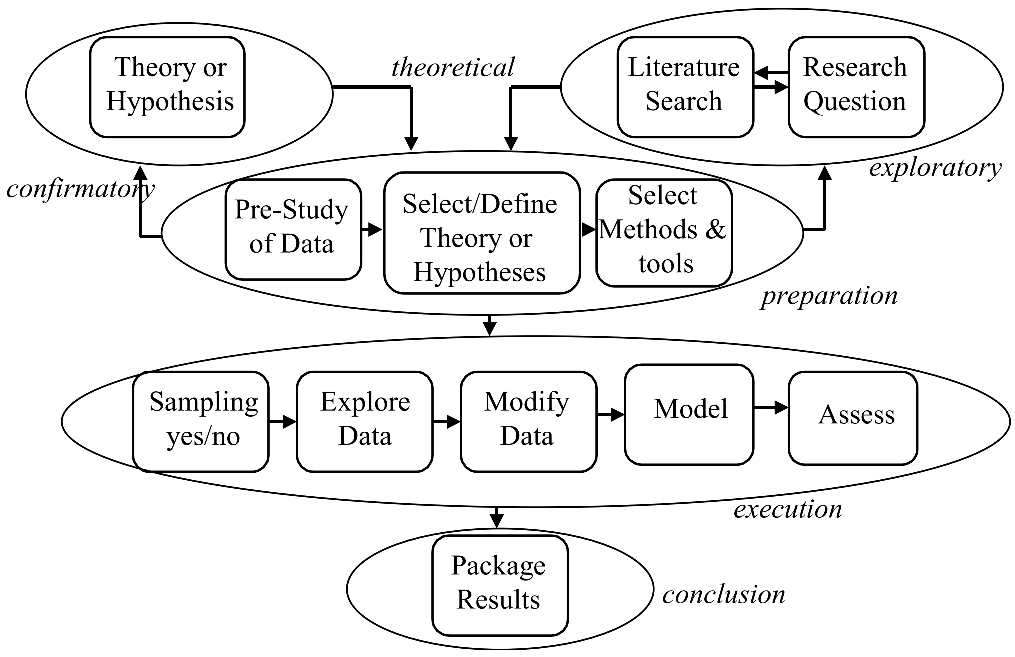


Figure 1 Steps in the process of exploring industrial data repositories in empirical research

3. Empirical studies in Ericsson

3.1. The context

Ericsson has developed several releases of two large-scale telecom systems using component-based development and a product line approach based on reusing software architecture and software components. Systems are developed incrementally and new features are added to each release of them.

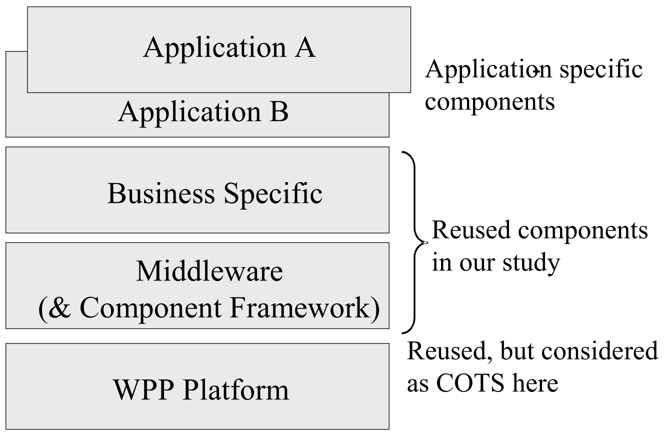

Application A
Application B
Application specific components
Business Specific
Middleware (& Component Framework)
Reused components in our study
Reused, but considered as COTS here
WPP Platform

Figure 2 High-level architecture of systems A & B

The high-level software architecture is shown in Figure 2. The first system (system *A*) was originally developed to provide packet data capability to the GSM (Global

System for Mobile communication) cellular network. A later recognition of common requirements with the forthcoming W-CDMA system (Wide-band Code Division Multiple Access) lead to reverse engineering of the original software architecture to identify reusable parts across the two systems. The two systems *A* and *B* in Figure 2 share the system platform, which is considered here as a Commercial-Off-The-Shelf (COTS) component developed by another Ericsson organization. Components in the middleware and business specific layers are shared between the systems and are hereby called for *reused components* (reused in two distinct products and organizations and not only across releases). Components in the application-specific layer are specific to applications and are called for *non-reused components*. All components in the middleware, business specific, and application-specific layers are built in-house.

The term *component* is used on two levels: for *subsystems* at the highest level of granularity and for *blocks*. The system is decomposed in a number of subsystems. Each subsystem is a collection of blocks and blocks are decomposed in a number of units, while each unit is a collection of software source code modules. Subsystems and blocks have interfaces defined in IDL (Interface Definition Language) and communication between blocks inside a subsystem or between subsystems happens through these interfaces. Communication within a block or unit is more informal and may happen without going through an external interface.

The systems' GUIs are programmed in Java, while business functionality is programmed in Erlang and C. Erlang is a functional language for programming concurrent, real-time, distributed, and fault-tolerant systems. The size of systems measured in equivalent C code is more that one million lines of non-commented source code. The development process is an adaptation of the Rational Unified Process (RUP) [RUP]. RUP is an incremental, use-case driven, and UML-based approach.

We collected and analyzed data gathered in the defect reporting, configuration management, change management, and effort reporting systems for three years of software development. Some results are described in [Moha04a] [Moha04b]. In [Moha03], we discuss how the results can be used to assess development approaches and measurement programs. We give a brief overview of three studies here. The external validity of all studies is threatened by the fact that the entire data set is taken from only one company. The results may be generalized to other systems within the same company or in similar domains.

3.2. Study of defect-density and stability of software components in the context of reuse

In order to quantitatively assess the impact of reuse on software quality, we decided to analyze data that is collected in the defect reporting and the configuration management systems. The defect reporting system included over 13 000 defect reports (corrective maintenance activity) for several releases of the systems. For three releases of system *A*, we had data on the components' size in Lines of Code (LOC) from the configuration management system.

Theory and preparation: Study of defects is usually reported connected to the subject of *reliability* (the ability of a system to provide services as defined), which is thoroughly studied in literature. However, reliability of component-based systems is a new field with few reported empirical studies. Based on the literature search and a pre-study of the available data, we found two groups of research goals:

1. Earlier theories or observations such as correlation between size of a component and its defect-density or the number of defects. Some studies report such a correlation while others not.
2. Studying relations between reuse and software quality metrics. Some results are reported from case studies in industry or university experiments.

We defined 4 hypotheses for quantitative assessment. We decided to assess whether reused components have less defect-density than non-reused ones have and are more stable (less modified between releases). We also decided to assess whether there is a relation between component size, and number of defects or defect-density, for all components and reused vs. non-reused ones (combining group 1 and 2). We chose Microsoft Excel and Minitab for performing statistical analysis.

Execution and results: We did not take a sample but used the whole dataset of some releases. All data on defects and components' size were inserted in a Microsoft SQL database using a C# program. Data for two releases of system *A* were used to assess hypotheses. Our results showed that size did not correlate with defect-density. Only for non-reused components, size correlated with the number of defects. Reused components had significantly less defect-density than non-reused ones and were less modified between releases. We concluded that reused components are designed more thoroughly and are changed with more care. One confounding factor is the type of functionality since non-reused components have more external interfaces than the reused ones have.

Contributions and experiences: Besides answering the research questions, the study was also useful for assessing the defect reporting system. The templates for reporting a defect had changed several times, introducing inconsistencies. Many Trouble reports had missing fields that reduced the internal validity of the results.

Research challenges: We met several challenges in the study:

1. The granularity of component definition: Some defect reports have registered only the subsystem name, while others have registered block name, unit name, or software module name. The main reason is that the origin of fault was not known when the defect was reported and the defect reports are not updated later for this information. We assessed our hypotheses both with subsystems and blocks with similar results. However, the number of subsystems was too low (9-10) for statistical tests.
2. The concept of reuse: Reuse may happen in the releases of the same product, or in multiple products and across organizations. Some mean that the first type cannot be classified as reuse. We defined a reused component to be a component that is used in more than one product.
3. Incremental and component-based development: Ideally hypotheses on defect-density should be assessed for both pre-release and post-release defects. As mentioned by Fenton [Fent00a], the results may differ and those modules that have most pre-release faults, may have least post-release faults. But this turned out to be difficult, if not impossible with the current data of several reasons: Only the whole system is labeled with a release date and not components, the development of a new release is usually running in parallel with testing of the previous one, a component is usually involved in several use cases and is

therefore updated and tested by several teams etc. Thus, relating defects to component releases or life-cycle phases was difficult.

3.3. Study of software change

We performed an exploratory study of the contents of the change management system. The database consisted of 160 Change Requests or CRs of 4 releases of system *A*. CRs are issued to add, delete, or modify a requirement after requirement baseline, or to add or modify a solution or documentation. The quality attributes related to software change are *stability, evolvability* or *maintainability* (or need for such).

Exploring the database: The variables that we had data on were size of components in LOC, type of components (reused or non-reused), and CRs in different releases. CRs are written in FrameMaker and Word using templates that have changed a few times and contain information on reason for the request, consequences, affected components, estimated effort etc.

Hypotheses selection based on literature and data: We found studies on distribution of maintenance activities and one study on the improvement of maintainability using a component-based architecture. Studies on requirement engineering have assumed that most changes are due to external factors (changing environment or customer needs). We found no study that on the origin of changes in more details. We decided to assess the distribution of change requests in the categories used in other studies (perfective, adaptive, preventive), over functional vs. non-functional reasons, phase (pre-or post delivery, before or after implementation), and to compare change-proneness in the number of CRs/size for reused vs. non-reused components.

Selecting and normalizing data: Data from CRs were inserted in a Microsoft SQL database using a C# program and partly manually. We noticed the same problems as described in Section 3.2 with missing data.

Contributions of the study: Our study showed that most CRs are initiated by the organization itself in order to improve a quality attribute (perfective and non-functional). The shares of adaptive/preventive changes are lower, but still not as low as reported in some previous studies. The study helped therefore to understand the origin of changes. We did not identify any significant difference between the change-proneness of reused and non-reused components. Most changes only affect one or two subsystems (high-level components). The study also showed that the percentage of accepted CRs is increasing over releases, which could be subject of further study. Performing such a study early would be useful to improve the CR reporting system. On some occasions, e.g. caused by coarse-granular components, we have too little data, which impacts conclusion validity. Missing data in some CRs is the biggest threat to internal validity.

Research challenges: We met again the challenge of the granularity of component definition: Change-proneness and the impact of CRs on sub-components could not be assessed since CRs only have information on affected subsystems and not blocks. We used the delivery data of the whole system for differing pre- and post-release CRs.

3.4. Study of effort

We have collected and partly analyzed data on the effort spent in 2 releases of system *A*. The goal of this study is to calibrate an estimation method based on use cases. This study is still going on, but it gave us insight on how effort is spent in different activities in several releases.

Selecting and normalizing data: Effort is registered using development phases such as analysis, coding, unit testing etc. for each member of a team. Teams are organized in different ways; i.e. around use cases, non-functional requirements such as performance, features that cross use cases, or 'just-in-time' for an extra task such as reengineering or re-factoring a solution or a component. There are also teams for handling methods and tools, configuration management, and system test. We received some effort data in printed form and some in Excel sheets. We had to parse the data, make consistent categories, re-group the data, insert it into new Excel sheets, and summarize it.

Experiences: There are inconsistencies in categories used in different releases and the effort reporting system has changed in the middle of one release.

Research challenges: We met the following challenges:

1. Organizational: We had data on effort spent by each team, but teams did not record their tasks detailed enough to divide the total effort between use cases, features, or non-functional requirements. Teams are also organized in different ways, making it difficult to map teams to requirements.
2. Use-case driven approach and component-based development: Ivar Jacobson, one of the pioneers of UML, the Unified Process (UP), and use cases writes that "a component realizes pieces of many use cases and a use case is usually realized by code in many components" [Jaco03]. These two decomposition effects are known as *tangling* and *scattering* [Tarr99]. Although these effects are well known and discussed both by Jacobson and others (recently especially by the Aspect Oriented Programming community), the impacts on metrics programs and effort reporting systems are not discussed. When effort is recorded per use case, it is spread over components and vice versa.
3. Use case driven and product line development: Requirements are first defined in features that are characteristics for product line development and later mapped to use cases. Tangling and scattering effects are observed here as well.

4. Discussion of research challenges

We faced two major challenges in comparing and combining results of the studies, which are discussed in other work as well (although with other labels), but not properly solved yet. We refer to them as the *challenges of integration* in two dimensions:

Physical integration refers to integration of databases. The research method may be shared, but the techniques used for exploration of data are very context dependent. In our examples, data on defects and CRs are stored in separate data repositories without having a common interface or analysis tool. One attempt to answer the challenge of physical integration is described in [Kitc01]. The authors' measurement model consists of three layers: The generic domain, the development model domain, and the project domain. The first two domains define the *metadata* for data sets. In this study, we achieved physical integration by inserting all data extracted in the three studies in a SQL database.

Conceptual integration refers to integrating the results of separate studies and integration of results into theories; either existing or new ones. This is not specific to this type of research and empirical studies generally suffer from lack of theories that bind several observations to one another. We observe that the conceptual challenges listed in Sections 3.2, 3.3, and 3.4 are mostly introduced in the intersection between development approaches:

- The granularity problem arises when the old decomposition system in industry meets the component-based development approach and when data is not collected consistently. For example, we could only compare change-proneness and defect-proneness of components in the highest level (subsystems) and did not have data on change-proneness of blocks.
- The reuse definition problem arises with the introduction of product line development without having consensus on definitions.
- Incremental and component-based development: metrics are either defined for the one or other approach.
- Use-case driven approach, product line development, and component-based development: effort reporting system is neither suitable for finding effort per use case or feature, nor per component.

We suggest two steps for solving these challenges and also integrating the results; both physically and conceptually:

1. Using a common database for data collection with facilities for search and data mining.
2. Defining metrics that are adapted for the combination of development approaches.

Some commercial metrics tools are available, but we have not studied them thoroughly enough to answer whether these are suitable for our purpose. The second step is the subject of the next section.

5. Metrics for incremental, reuse, and component-based development

Fenton et al. write: “Most objectives can be met with a very simple set of metrics, many of which should be in any case be available as part of a good configuration management system. This includes notably: information about faults, failures and changes discovered at different life-cycle phases; traceability of these to specific system ‘modules’ at an appropriate level of granularity; and ‘census’ information about such modules (size, effort to code/test)” [Fent00b]. We can’t agree more, but also add that metrics should be adapted for a mixture of development approaches.

We use experiences in the three above examples and other studies we have performed in Ericsson to propose improvements and identify metrics as described:

1. Decide the granularity of ‘modules’ or ‘components’ and use it consistently in metrics. Don’t define some metrics with one component granularity and others with another, unless it is clear how to combine or compare such metrics.
2. The following data should be gathered for components:
    2.1. Size (in Lines of Code if developed in-house or if source code is available, or in other proper metrics) at the end of each release,
    2.2. Size of modified code between releases,

   2.3. Faults (or defects), with information on life-cycle phase, release and product identity,
   2.4. Effort spent on each component in each release,
   2.5. Trace to requirement or use case (this is also useful for documentation and debugging) that could be updated when the component is taken in use,
   2.6. Type: new, reused-as-is or modified,
   2.7. Change requests in each release,
   2.8. Date of delivery in a release that can be set by a configuration management system and be easily used later to decide whether a fault is detected pre-or post-release, or whether a change request is issued pre- or post-delivery.
3. The following data should be gathered for increments or releases:
   3.1. Total size of the release,
   3.2. Size of new and modified code,
   3.3. Requirements or use cases implemented,
   3.4. Effort spent in the release.
4. Effort should be recorded both per component and per use case or feature.

The list shows that it doesn't help to define a set of metrics for a development approach without considering the impact of other approaches. Having this data available would make it possible to assess software quality in different dimensions and answer questions such as: Are defect-density and change-proneness of components correlated? Can we estimate effort based on the number or complexity of use cases, or changes in components? Which components change most between releases? What is the impact of reuse, component-based, incremental development, or a combination of these on needed effort? Hence, we could build theories that combine development approaches.

6. Conclusions and future work

We presented three empirical studies performed by exploring industrial data repositories. We could verify hypotheses on the benefits of reuse, explore the origin the changes for future studies, and study effort distribution and adapt an estimation method, *empirically* and *quantitatively*. As our examples show, quantitative techniques may be used in different types of research. In many cases, exploring industrial data repositories is the only possible way to assess a theory in the real world.

While some concrete results are already published, this paper has the following contributions:

1. Promote the discussion on exploring industrial data repositories as an empirical research method, its advantages and limitations, and presenting a simple method to do so. The method described in Section 2.3 combines the theoretical and preparation phases defined by us, with steps of a data-mining process as defined in [Coop01].
2. Getting insight into the challenges of defining and collecting metrics when development approaches are used in parallel.
3. Identifying a basic set of metrics for incremental, component-based, and reuse-based development.

The set of metrics proposed in Section 5 does not contain any new metrics, but emphasizes that metrics should be adapted for a combination of development approaches. This basic set should be collected before we can build advanced theories on the relations between development approaches.

We plan to work further on the physical and conceptual challenges meeting measurement programs, with focus on evolution of component-based systems in the upcoming SEVO project (Software Evolution in Component-Based Software Engineering) [SEVO04].

7. Acknowledgements

The work is done in the context of the INCO project (INcremental and COmponent-based Software Development [INCO01]), a Norwegian R&D project in 2001-2004 and as part of the first author's PhD study. We thank Ericsson in Grimstad for the opportunity to perform the studies.

8. References

[Basi94] Basili, V.R., Calidiera, G., Rombach, H.D., "Goal Question Metric Paradigm", In: Marciniak, J.J. (ed.): *Encyclopaedia of Software Engineering*. New York Wiley 1994, pp. 528-532.

[Coop01] Cooper, D.R., Schindler, P.S., *Business Research Methods*, McGraw-Hill International edition, seventh edition, 2001.

[Fent00a] Fenton, N.E., Ohlsson, N., "Quantitative Analysis of Faults and Failures in a Complex Software System", *IEEE Trans. Software Engineering*, 26(8), 2000, pp. 797-814.

[Fent00b] Fenton, N.E., Neil, M., "Software Metrics: Roadmap", *Proc. of the Conference on the Future of Software Engineering*, June 04-11, 2000, Limerick, Ireland, pp. 357-370.

[Flyv91] Flyvbjerg, B., *Rationalitet og Magt I- det konkretes videnskab*, Akademisk Forlag, Odense, Denmark, 1991.

[Jaco03] Jacobson, I., "Use Cases and Aspects- Working Seamlessly Together", *Journal of Object Technology*, 2(4): 7-28, July-August 2003, online at: http://www.jot.fm

[INCO01] The INCO Project: http://www.ifi.uio.no/~isu/INCO/

[Jørg04] Jørgensen, M., Sjøberg, D., "Generalization and Theory Building in Software Engineering Research", Accepted in the *8th International Conference on Empirical Assessment in Software Engineering (EASE2004)*, 24-25 May 2004, Edinburgh, Scotland.

[Lehm96] Lehman, M.M., "Laws of Software Evolution Revisited", In Carlo Montangero (Ed.), *Proc. European Workshop on Software Process Technology* (EWSPT96), Nancy, France, 9-11 Oct. 1996, *Springer LNCS 1149*, pp. 108-124.

[Kitc01] Kitchenham, B.A., Hughes, R.T., Linkman, S.G., "Modeling Software Measurement Data", *IEEE Trans. Software Engineering*, 27(9): 788-804, September 2001.

[Moha03] Mohagheghi, P., Conradi, R., "Using Empirical Studies to Assess Software Development Approaches and Measurement Programs", *Proc. 2nd Workshop in Workshop Series on Empirical Software Engineering - The Future of Empirical Studies in Software Engineering (WSESE'03),* Rome, 29 Sept. 2003, pp. 65-76.

[Moha04a] Mohagheghi, P., Conradi, R., Killi, O.M., Schwarz, H., "An Empirical Study of Software Reuse vs. Defect-Density and Stability", Proc. of the 26th International Conference on Software Engineering (ICSE'04), *IEEE Computer Society* Order Number P2163, pp.282-292.

[Moha04b] Mohagheghi, P., Conradi, R., "An Empirical Study of Software Change: Origin, Acceptance Rate, and Functionality vs. Quality Attributes", Accepted in the *2004 ACM- IEEE International Symposium on Empirical Software Engineering (ISESE'04)*, 10 p.

[Tarr99] Tarr, P., Ossher, H., Harrison, W., Sutton, S., "N Degrees of Separation: Multi-Dimensional Separation of Concerns", *Proc. of ICSE* 1999, pp.107-119, 1999.

[RUP] www.rational.com

[SEVO04] The SEVO project: http://www.idi.ntnu.no/grupper/su/sevo.html

[Zelk98] Zelkowitz, M.V., Wallace, D.R., "Experimental Models for Validating Technology", *IEEE Computer*, Vol. 16, pp. 191-222.

[Wohl00] Wohlin, C., Runeseon, P., M. Höst, Ohlsson, M.C., Regnell, B., Wesslen, A., *Experimentation in Software Engineering*, Kluwer Academic Publications, 2000.

[Yin03] Yin, R.K., "Case Study Research, Design and Methods", *Sage Publications*, 2002.

P12. A Study of Effort Breakdown Profile in Incremental Large-Scale Software Development

Parastoo Mohagheghi, Reidar Conradi
Norwegian University of Science and Technology

Abstract

Software projects often exceed their budgets, schedules or usually both. Some reasons are too optimistic estimations, poor knowledge about how to break down effort to different activities in a top-down estimation or how to estimate the total effort based on estimating some activities. Effort breakdown profiles are therefore important to study and such profiles should be updated for major changes in development approaches or tools. There is also a need for empirical assessment of profiles in organizations.

We gathered data on effort estimations and the actual effort spent in two releases of a large telecom software system that is developed incrementally. The data on effort spent in different activities show that only half the effort is spent before system test on specification, analysis, design, coding and unit testing. The other half is spent on system test, project management, software processes and Configuration Management (CM). The contributions of the study are: 1) presenting an effort breakdown profile showing the share of activities such as software process adapting, CM and system test for an incrementally developed large-scale system with recent technologies, and 2) suggesting that incremental development will increase the share of system testing and CM. When a system is developed incrementally, software developed in different increments should be integrated, and regression testing and other techniques such as inspections should secure quality.

Keywords. Effort estimation, effort break down, incremental development, software process

Introduction

We may have a look at some software estimation methods as an introduction to the motivation behind this study. Estimation methods are roughly divided in two groups: *top-down* and *bottom-up*. In a top-down method, total effort or elapsed time is estimated based on some properties of the project as a whole and later is distributed over project activities. Examples of top-down estimation methods are COCOMO 2.0 [3] and regression analysis using historical databases. The bottom-up approach involves breaking down the actual project into activities, estimating these and summing up to arrive at the total required effort or time [7]. Magne Jørgensen suggests that expert estimations are more accurate when performed bottom-up, unless the estimators have experience from or access to very similar projects [4].

Although expert estimation is probably the most widely used method for estimation of software projects, the properties of such estimation methods are not well known [4]. Even software companies have few explicit guidelines to help their experts in estimating software projects in a given organizational context. In Norway, the Simula Research Laboratory has started the BEST project (Better Estimation of Software Tasks) to stimulate research on software estimation. Results of a recent survey on the state of estimation among 52 project managers in 18 Norwegian companies published in the BEST website [2] shows that average effort overruns are 41%, that expert estimation is the dominating estimation method and the software estimation performance has not changed much the last 10-20 years [6].

One way to increase the accuracy of estimations is to improve our understanding of the share of different activities in software development or so-called *effort breakdown profiles*. An example of a breakdown as "the industry average profile for a project that uses traditional methods" is suggested by Charles R. Symons to be: Analysis 22%, Design 15%, Coding and Unit Test 46%, System Test 12% and Implementation 5% [1][7]. As Symons mentions, the profile varies depending on several factors. For example, if a project uses a powerful CASE tool to generate code, the share of Coding will decline, and the share of Analysis and Design will increase. The profiles must therefore be calibrated for different organizations, development methods or tools. Effort breakdown profiles are important when estimating in a bottom-up style, when breaking down the total effort between activities in a top-down method, or for evaluating and calibrating estimation methods. Our study to calibrate a top-down estimation method for an industrial project showed that the profile is important in calibrating. The original method was tested on projects with a profile similar to Symons, while projects in our study spent a lot of effort on activities that were either not predicted in the estimation method (such as CM) or consumed much more effort than predicted.

Software Processes and Effort

There are two new factors in large-scale system development that need more attention when discussing effort. The first factor is that software companies are increasingly developing software using systematic software processes that should be developed or adapted and be maintained. Some known examples are the Rational Unified Process (RUP) or eXtreme Programming (XP). Introducing a software process needs training as well, and the cost or effort of introducing or maintaining software processes should be explicit in the total profile; i.e. not be buried down in other activities. The second factor is that large-scale systems are developed incrementally. Although there is some evidence that incremental development reduces the risks for schedule overruns [5], there are no empirical studies on the relation between incremental development, and effort. We may assume that:

- Integration effort increases due to several increments that must be integrated.
- There is an increasing need for CM systems and processes to handle iterations, releases or upgrades of different releases. Mark Vigder in his position paper on

[1] These activities are not defined in more details. It is reasonable to think that Specification is included in Analysis, and Implementation covers also deployment and installation.

maintainability of component-based systems suggests that we need flexible CM to ease adding, removing, replacing and upgrading components [8]. This is true also for incremental development of software.

- System Test effort increases due to regression testing to assure that new functionality complies with old one.
- Effort spent on quality assurance techniques such as inspections may increase to assure compliance with old deliveries and consistency among those. These techniques also need adaptation to incremental development.

COCOMO 2.0 assumes an incremental model for development but the impact on effort is unclear. COCOMO 2.0 includes also a factor for economy or diseconomy of scale when the system size grows. Applying CASE tools or other tools that facilitate software development and testing are some reasons for the economy of scale. Growing communication overhead and increased dependencies are some reasons for the diseconomy of scale. Benediktsson et al. analyzed a COCOMO-style effort framework to explore the relation between effort and the number of increments [1]. In their model, effort will decrease with allowing sufficiently high number of increments (around 20) when the diseconomy of scale is large. However, their calculation only includes the diseconomy of scale factor and not increased effort due to other factors in incremental development.

Some Historical Data

We have analyzed data on effort spent in developing two releases of a large-scale telecom software system. The software process is an adaptation of RUP. Each release is developed in 5-7 iterations of 2-3 months duration, and the development environment uses CM tools and routines for integration and testing of new increments and new releases. The system is modeled in UML and coded mostly manually in multiple programming languages. The system size is in equivalent C code is calculated to be more than one million non-commented source lines of code. Data on effort spent in different activities as reported by all staff in an effort-recording system is collected and summed up in the following categories as shown in Table 1:

- Development before System Test: Specification, Analysis and Design, Coding, Module Test, Use Case Test, trouble fixing, reviews and inspections.
- System Test: All testing done in simulated and real environment in the company, but excluding final node test in customers' site.
- Project Management: Administration and project meetings.
- Software Process: Adapting and maintaining RUP and related tools.
- CM: Release management, build, and patching.
- Other: Travels, documentation and unspecified.

Table 1 Percentages of effort spent in different activities

	Development before System Test	System Test	Project Management	Software Process	CM	Other
Rel. 1	49	25	10	2	11	3
Rel. 2	55	18	11	5	7	4

Note that Release 2 is not fully tested yet and the share of system test will slightly increase for this release.

The company in the study used an inside-out estimation method; i.e. it estimated the effort needed for Development before System Test and multiplied it by an overhead factor to cover the rest. The overhead factor varied between 1.0 and 2.5 in different estimations. Comparing estimations with the actual data suggests that expert estimations were too optimistic (almost 50% less than the actual effort used in Development before System Test). Data in the study shows that managing teams, processes and deliveries counts for 23% of the total effort in both releases, with roughly half on Project Management, and half on Software Process and CM. Besides, System Test takes 20-25% of effort. In fact, System Test can run as long as the project schedule allows (remember Parkinson's law: Work expands to fill what time available), but empirical data shows the above share. Other observations are:

- Effort spent in Development before System Test must be multiplied approximately by 2 to give the total effort. The empirical data allows finding an overhead factor that may be used for future estimations.
- Symons predicts a reduction in effort when methods or tools are used for the second time, and COCOMO's precedentness factor has a similar effect. We observe a reduction in CM effort, while a slight increase in Software Process, which we relate it to more extensive work with software processes in the second release.

Conclusions

An effort profile for incremental development of a large telecom system is presented, but we have no data from similar studies to compare this with in this domain or other dimains. It is reasonable to think that with incremental development more effort will be needed in putting pieces together, reflected in more CM and system testing. One way to handle this is to add a percentage to effort in some activities. We wonder whether the profile has characteristics that may be generalized to other projects.

Such historical data may be useful for researchers to calibrate estimation methods and study relations between development approaches and effort. The distribution of effort over activities may vary with the type of systems and organizations. However, the distribution seems to be relatively stable for releases of a single system and may be generalized to similar systems in the same organization and is therefore worth to study for practitioners. They may also use the results of such studies as rule-of-thumbs rules for estimating total effort based on some activities or to distribute the estimated effort by some top-down method over activities.

References

[1] Benediktsson, O., Dalcher, D.: Developing a new Understanding of Effort Estimation in Incremental Software Development Projects. *Proc. Intl. Conf. Software & Systems Engineering and their Applications* (ICSSEA'03), December 2-4, 2003, Paris, France. Volume 3, Session 13, ISSN 1637-5033, 10 p.

[2] The Best project: http://www.simula.no/~simula/se/bestweb/index.htm

[3] Boehm, B., Clark, B., Horowitz, E., Westland, C., Madachy, R., Selby, R.: Cost Models for Future Software Life Cycle Processes: COCOMO 2.0. *USC center for software engineering*, 1995, online at: http://sunset.usc.edu/research/COCOMOII/

[4] Jørgensen, M.: Top-down and Bottom-up expert Estimation of Software Development Effort. *Information and Software Technology*, vol.46 (2004), pp. 3-16.

[5] Moløkken, K., Lien, A.C., Jørgensen, M., Tanilkan, S.S., Gallis, H., Hove, S.E.: Does Use of Development Model Affect Estimation Accuracy and Bias? Proc. Product Focused Software Process Improvement: 5th International Conference, PROFES 2004, Kansai Science City, Japan, April 5-8, 2004. *Springer-Verlag*, ISBN: 3-540-21421-6. pp. 17-29.

[6] Moløkken, K., Jørgensen, M., Tanilkan, S.S., Gallis, H., Lien, A.C., Hove, S.E.: A Survey on Software Estimation in Norwegian Industry". Accepted for Metrics 2004.

[7] Symons, P.R., *Software Sizing and Estimating MK II FPA (Function Point Analysis)*, John Wiley & Sons, 1991.

[8] Vigder, M.: Building Maintainable Component-Based Systems. *Proc. 1999 International Workshop on Component-Based Software Engineering*, pp. 17-18 May 1999. http://www.sei.cmu.edu/cbs/icse99/papers/38/38.pdf

.

P13. Use Case Points for Effort Estimation – Adaptation for Incremental Large-Scale Development and Reuse Using Historical Data

Parastoo Mohagheghi[1], Bente Anda[2], Reidar Conradi[1]
[1]*Department of Computer and Information Science, NTNU, NO-7491 Trondheim, Norway*
[2]*Simula Research Laboratory, P.O.Box 134, NO-1325 Lysaker, Norway*
parastoo@idi.ntnu.no, bentea@simula.no, conradi@idi.ntnu.no

Abstract

In Incremental development, each release of a system is built on a previous release and design modifications are made along with adding new capabilities. This paper describes an empirical study, where an effort estimation method based on use cases, the Use Case Points (UCP) method, is extended for incremental development with reuse of software from a previous release and is calibrated for a large industrial telecom system using historical data. The original method assumes that use cases are developed from scratch and typically have few transactions. Use cases in this study as defined in use case specifications, are complex, contain several main and alternative flows and are typically modified between releases. The UCP method was adapted using data from one release and the estimated result counted approximately for all the activities before system test. The method was tested on the successive release and produced an estimation that was 17% lower than the actual effort. Results of the study show that although use cases vary in complexity in different projects, the UCP estimation method can be calibrated for a given context and produce relative accurate estimations.

1. Introduction

Effort Estimation is a challenge every software project face. The quality of estimation will impact costs, expectations on schedule, and expectations on functionality and quality. While expert estimations are widely used, they are difficult to analyze and the estimation quality depends on the experience of experts. Consequently, rule-based methods should be used in addition to expert estimates in order to improve estimates. Since most software is developed incrementally, estimation methods should be updated for iterative enhancement of systems. Evolutionary project management and iteration planning needs an estimation method that can estimate the effort based on evolutionary changes in requirements. There is also necessary to verify whether a proposed estimation method scales up for large system development.

This paper presents a top-down effort estimation method based on use cases, called the Use Case Points (UCP) method. The method was earlier used in some industrial projects as well as in some student projects with success, although it is still not widely used. The goal of this study was to evaluate whether the method scales up for large systems with incremental development.

We broke each Use Case Specification (UCS) down in several simple ones to compensate for the size and complexity of the existing use cases, and calculated the unadjusted use case points for complete UCSs and for modified steps in each to account for incremental development. We also calculated effort needed to build on a previous release of a system by applying a formula from COCOMO 2.0 for reuse of software. The adapted UCP method was developed using data from one release and produced good estimates for the successive release. We also found that our projects spent more effort on system test and Configuration Management (CM) than earlier studies, which impacts the estimation method in the sense that it is reasonable to estimate effort for development before system test, as the practice is in the organization.

This paper is organized as follows. Section 2 presents some state-of-the-art of estimation methods, the UCP method and challenges in estimating incrementally developed projects. Section 3 introduces the context. The research questions are formulated in Section 4. Section 5 presents how the UCP is adapted to the context and Section 6 gives the estimation results. The results are further discussed in Section 7. Section 8 summarizes the observations and answers the research questions. The paper is concluded in Section 9.

2. Estimation Methods

2.1. A Brief Overview of Estimation Methods

Software estimation methods are roughly divided into *expert estimations* based on previous experience, *analog-based* estimations (comparing a project to a previous one being more formal than expert estimates), and formal *cost models* that estimate effort or duration using properties such as size and various cost drivers. Each of these can be performed *top-down* or *bottom-up*. In a top-down method, the total effort or elapsed time is estimated based on some properties of the project as a whole and is later distributed over project activities. The bottom-up approach involves breaking down the actual project into activities, estimating these and summing up to arrive at the total required effort or duration [Symons91]. There are variants as well, for example to estimate effort for some activities and estimating the total effort based on these core activities; i.e. the *inside-out* method.

Project success or failure is often viewed in terms of adhering to a budget and to deliver on time. Good estimation is therefore important for a project to be considered as successful [Verner03]. We focus on three estimation methods in this paper: a) expert estimates that are relevant for the case study, b) COCOMO 2.0 that we partly use in our method, and c) the UCP method that is adapted for the study and is presented in the next section.

Although expert estimation is probably the most widely used method for estimation of software projects, the properties of such estimation methods are not well known [Jørgensen04]. Results of a recent survey on the state of estimation among 52 project managers in 18 Norwegian companies shows that expert estimation is the dominating estimation method and average effort overruns are 41% [BEST04]. It is therefore recommended to balance expert-based and model-based estimations.

COCOMO (the Constructive Cost Model) is a well-known estimation method developed originally by Barry Boehm in 1970s [Boehm95]. The 1981 COCOMO and the 1987 Ada COCOMO update have been extended in COCOMO 2.0 for several factors. These include a non-linear model for developing with reuse, non-sequential and rapid development, and using Function Points (FP) and Object Points (OP) in addition to Source Lines of Code (SLOC) for software sizing. COCOMO takes software size and a set of factors as input, and estimates effort in person months. The basic equation in COCOMO is:

$$E = A * (Size)^{B} \qquad \textit{EQ.1}$$

E is the estimated effort in Person-Months, *A* is a calibration coefficient, and *B* counts for economy or diseconomy of scale. Economy of scale is observed if effort does not increase as fast as the size (i.e. $B<1.0$), because of using CASE tools or project-specific tools. Diseconomy of scale is observed because of growing communication overhead and dependencies when the size increases. COCOMO 2.0 suggests a diseconomy of scale by assuming $B>1.0$. COCOMO 2.0 also includes various cost drivers that fall out of the scope of this paper.

Because of difficulties in estimating SLOC, FP or OP, and because modern systems are often developed in UML and with use cases, estimation methods based on use cases are proposed.

All estimation methods are imprecise, because the assumptions are imprecise. Jørgensen et al. [Jørgensen03] suggest that large software projects are typically under-estimated, while small projects are over-estimated.

2.2. The Use Case Points Estimation Method

A use case model defines the functional scope of the system to be developed. Attributes of a use case model may therefore serve as measures of the size and complexity of the functionality of a system. In 1993, Karner introduced an estimation method that derives estimation parameters from a use case model, called the *Use Case Points (UCP)* estimation method [Karner93]. The method is an extension of the *Function Points Analysis* and *MK II Function Points Analysis* [Symons91]. The UCP method has been evaluated in several industrial software development projects (small projects comparing to this study) and student projects. There have been promising results [Arnold98][Anda01][Anda02], being more accurate than expert estimates in industrial trials.

We give a brief introduction of the six-step-UCP method in Table 1. Steps 2,4, and 6 are further explained below. In Table 1, *WF* stands for Weight Factor, *UAW* is the Unadjusted Actor Weights, *UUCW* is the Unadjusted Use Case Weights, *UUCP* is the Unadjusted Use Case Points, *UCP* is the adjusted Use Case Point, *PH* is Person-Hours, and *E* is the estimated effort in PH.

Step 2. Karner proposed not counting so-called including and extending use cases, but the reason is unclear. Ribu presents an industrial case, where use cases were classified based on the extent of code reuse: a simple use case has extensive reuse of code, while a complex one has no reuse of code [Ribu01].

Step 4. Various factors influencing productivity are associated to weights, and values are assigned to each (0..5). There are 13 *Technical Factors* (e.g. distributed system, reusable code and security) and eight *Environmental Factors* (e.g. Object-Oriented experience and stable requirements). Each factor is given a value, multiplied by its weight (-1..2), and the *TFactor* and *Efactor* are weighted sums. The weights and the formula for technical factors is borrowed form the *Function Points* method proposed by Albrecht [Albrecht79]. Karner, based on some interviews of experienced personnel, proposes the weights for environmental factors. The background of the formula for environmental factors is unknown for us, but it seems to be calculated using some estimation results.

Table 1 The UCP estimation method

Step	Task	Output
1	Classify use case actors: a) Simple, WF = 1. b) Average, WF = 2. c) Complex, WF = 3.	UAW = ∑ (#Actors in each group*WF)
2	Classify use cases: a) Simple (3 or fewer transactions), WF = 5. b) Average (4 to 7 transactions), WF = 10. c) Complex (more than 7 transactions), WF= 15.	UUCW = ∑ (#use cases in each group*WF)
3	Calculate UUCP	UUCP = UAW + UUCW

4	Assign values to the 13 technical, and 8 environmental factors.	TCF=0.6 + (0.01* TFactor),
		EF=1.4 + (-0.03 * EFactor)

5	Calculate UCP.	UCP = UUCP * TCF * EF
6	Calculate effort in PH.	E = UCP*PH/UCP

Step 6. Karner proposed 20 PH/UCP (Person-Hours per UCP) using estimation results of three projects in Objectory, while others have used between 15 and 36 [Ribu01][Anda01]. Karner proposed 20 PH/UCP based on three projects conducted in Objectory. Schneider & Winters refined the original method and proposed 28 PH/UCP if the values for the environmental factors indicate negatives with respect to the experience level of the staff or the stability of the requirements [Schneider98]. The method was extended by Robert Russell to use 36 PH per UCP when the values for these factors indicate a particularly complex project [Russell04]. Previous evaluations of the method have used 20 PH/UCP [Anda01]. Note that the method estimates effort in PH, and not duration of a project.

Table 2 shows examples from [Anda01] where the method is applied to three industrial projects in a company in Norway with 9-16 use cases each. The application domain was banking.

Table 2 Some examples on PH/UCP

Project	UCP	Estimated Effort	Actual Effort	Actual PH/UCP
A	138	2550	3670	26.6
B	155	2730	2860	18.5
C	130	2080	2740	21.1

The UCP method has some clear advantages:

- It gives early estimation top-down. Non-technical estimators usually prefer a top-down estimation strategy [Moløkken02].
- It is suitable when guessing SLOC is difficult, such as in development with COTS (Commercial Off-The-Shelf) software.
- It is independent of the realization technologies, e.g. programming languages.
- Expert estimation processes are non-explicit and are difficult to analyze and give feedback. The UCP method is explicit, allows feedback and adaptation and hence improvement.
- The method eliminates biases in expert estimation.

We also see the disadvantages such as:

- Use cases are not always updated before analysis starts. But if the project decides to use the UCP method, this will promote developing a high quality and stable use case model early.
- The UCP method depends on up-front requirements work for the whole release. Otherwise, the estimation should be repeated for each iteration, which is possible using our adaptation of the method for incremental changes in use cases.
- The method only counts use cases that essentially express functional requirements, not supplementary specifications. The influence of non-functional requirements is reflected in technical factors, which has little influence on the results.
- The method depends on use cases that are well structured and with proper level of details, but not too detailed [Ribu01]. There is no standard way of writing use cases, and practices vary.
- The method is not properly verified.

Two other methods have been proposed for estimation based on use cases [Fetcke98] [Smith91]. These methods respectively make assumptions about the relationship between use cases and function points, and between use cases and SLOC in the final system. There are also commercially available tools for estimation that are based on the UCP method, e.g. Enterprise Architect [Enterprise] and Estimate Easy UC [Estimate].

2.3. Estimation in Incremental Development

Modern software is developed incrementally or evolutionary. Incremental development is usually used for development methods with major up-front specification, while in an *evolutionary approach* product improvements are not preplanned and requirements are gradually discovered. In both approaches, each iteration delivers a working system being an *increment* to the previous delivery or release. Incremental methods such as the Spiral method, the Rational Unified Process (RUP) and the recent agile methods like eXtreme Programming (XP) emphasize user participation, risk-driven development and incremental covering (or discovering) of requirements. RUP is a use-case driven approach that allocates use cases to each iteration. However, in practice some new requirements are defined in new use cases, while other modifications are done by changes in existing use cases.

A challenge in estimation of incrementally developed projects is to count for reuse of software delivered in previous releases. The cost of this reuse is not properly studied. Boehm et al. refer to an earlier study by Parikh and Zvegintzov in 1983 that 47% of the effort in software maintenance involves understanding the software to be modified [Boehm95]. They also write that there are non-linear effects involved in module interface checking, which occurs during the design, code, integration and test of modified code.

Benediktsson et al. analyzed the COCOMO 2.0 model to explore the relation between effort and the number of increments [Benediktsson03]. They extended *EQ.1* for incremental development where they assume an overhead factor between 0.05 and 0.30 for changing code, adding code to a previous release and learning between releases. They calculated effort for incremental development, compared to a waterfall model for different values of *B* in *EQ.1* and different overhead factors. They concluded that when *B* is small (e.g. 1.05), increasing the number of increments has little influence on the effort. However, when *B* increases to 1.20, increasing the number of increments from 2 to 20 reduces the effort by 60%. I.e. incremental development will need less effort than the waterfall model when the diseconomy of scale is significant. Although there is some evidence that incremental development reduces the risks for schedule overruns [Moløkken04], we have not found any empirical studies on the relation between incremental development and effort that can verify or falsify this claim.

3. The Company Context

3.1. Background and Motivation of the Study

The system in this study is a large telecom system developed by Ericsson. It is characterized by large scale, multi-site development, development for reuse since some software components are shared with another product and multi-programming languages (mostly non Object-Oriented programming languages but also minor parts in Java). The size calculated in equivalent C code exceeds 1000 KSLOC (Kilo SLOC). The system is developed incrementally and the software process is an adaptation of RUP. Each release has typically 5-7 iterations and the duration of iterations is 2-3 months. The architecture is component-based with components built in-house. Several Ericsson organizations in different countries (in periods more than 200 developers) have been involved in development, integration and testing of releases.

On the highest level, requirements are defined by use cases and supplementary specifications (for non-functional requirements, e.g. availability, security, and performance).

Expert estimations are used in different phases of every release (before inception and during inception and elaboration phases), in a bottom-up or inside-out style. Expert estimations done by technical staff tend to be over-optimistic and it is difficult to calibrate these. We decided therefore to evaluate whether the UCP method can produce better estimations as a method that may be applied by non-technical staff as well.

3.2. Use Case Specifications

The use case model in our study includes use case diagrams modeled in Rational Rose, showing actors and relations between use cases, while flows are described in textual documents (UCSs). Each UCS includes:

a) One or several main flows: Main flows are complex and have several steps with several transactions in each step. There may be cases when several flows are equally desired. In these cases there are several main flows.
b) One or several alternative flows: Each alternative flow has one or several steps.
c) Some UCSs also have exceptional flows: These describe events that could happen at just any time and terminate a flow. Exceptional flows are described in a table, which gives the event that triggers an exceptional flow, action, and the result.
d) A list of parameters and constraints, such as counters or alarms.

Extending a use case means sometimes that the extended one is a pre-condition for this one and sometimes extra behavior is added. *Including* another use case means that the behavior of the included use case is added to this use case.

Each release may contain new use cases (and UCSs). Usually, behavior of previous use cases is modified or extended, with new or modified steps, flows or parameters. What is new in each use case is marked with bold and blue text in the UCS.

4. Research Questions

We have formulated the following research questions for this study:

RQ1: Does the UCP method scale up for a large industrial project?
RQ2: Is it possible to apply the UCP method to incremental changes in use cases?
RQ3: How to calculate effort needed to reuse software from a previous release?
RQ4. Evaluation of the UCP method: Does the method produce usable results? Does it fit into the industrial settings and the development process? Do the steps of the process make sense?

The UCP method in its original form estimates effort needed to develop use cases from scratch. It is not clear which activities are covered and it is not tested on a large system.

5. Adapting the Use Case Point Estimation Method

We started to count UUCW for release 1 using the method described in Section 2.2. All use cases in this study would be classified as complex. Nevertheless, the total UUCP would be still very low for all 23 use cases (23*15=345 UUCP). Comparing the complexity of these use cases with previous projects convinced us that we have to break use cases down into smaller ones. Since software is built on a previous release, we should also find how to estimate effort for reuse. This section describes our choices to adapt the UCP method and the reason behind each decision. The adaptation rules are summarized in Table 3. Additional information on each step is given below.

Step 1. Actors. An actor may be a human, another system or a protocol. However, the classification has little impact on the final estimation result. Modified actors are counted and MUAW is the Modified UAW.

Step 2. Counting the UUCW and MUUCW (Modified UUCW). We broke each use case down into smaller ones as described in Rules 2.1 to 2.4. Rewriting UCSs is too time-consuming while counting flows and steps is an easy task.

The new use cases should be classified as simple, average or complex. A first attempt to follow the rule described in Section 2.2 resulted in most use cases being classified as simple (66%) and very few as complex. But the complexity of transaction does not justify such distribution.

Table 3 The adapted UCP estimation method

Step	Rule/Task	Output
1	1.1. Classify all actors as Average, WF = 2.	UAW= #Actors*2
	1.2. Count the number of new actors.	MUAW= #New actors*2
2	2.1. Since each step in the main flow contains several transactions, count each step as a single use case. 2.2. Count each alternative flow as a single use case. 2.3. Exceptional flows, parameters, and events are given weight 2. Maximum weighted sum is limited to 15 (a complex use case). 2.4. Included and extended use cases are handled as base use cases. 2.5. Classify use cases as: a) Simple (2 or fewer steps), WF = 5. b) Average (3 to 4 steps). WF = 10, c) Complex (more than 4 steps), WF= 15.	UUCW = ∑ (#use cases in each group*WF) + ∑(Points for exceptional flows and parameters)
	2.6. Count points for modifications in use cases according to rules 2.1-2.5.	MUUCW = ∑ (#New or modified use cases in each group*WF) + ∑(Points for new or modified exceptional flows and parameters)
3	3.1. Calculate UUCP for all software.	UUCP = UAW + UUCW
	3.2. Calculate MUUCP for new software.	MUUCP=MUAW + MUUCW
4	Assume average project.	TCF=EF=1
5	5.1. Calculate UCP.	UCP = UUCP
	5.2. Calculate MUCP.	MUCP= MUUCP
6	6.1. Calculate effort for reuse of software.	RE=(UCP-MUCP) *0.55*PH/UCP
	6.2. Calculate effort for new development.	ME= MUCP*PH/UCP
	6.3. Calculate total effort.	E=ME+RE

An example of a use case called for *Connect* is given in Figure 1. In Figure 1, M1 is described as one step, but it includes verifying that the received message is according to the accepted protocols. M2 refers to an included use cases, while M3 has 4 steps, where none of these is a single transaction and includes another use case as well. Therefore, we chose to classify the use cases according to the Rule 2.5. M1 and M2 would be classified as simple, while M3 would be an average use case.

The UUCW calculated above is for use cases developed from scratch. The use cases in each release are typically modified ones. For modified use cases, the same rules are applied, but for modified steps. The method is similar to the example given in Section 2.2- Step 2, where a simple rule-of-thumb was used (extensive reuse gives simple use case etc). For example, two steps in M3 in Figure 1 are new or modified. These will be counted as a new simple use case. Thus, the use case is 5/27=19% modified.

Steps 4 and 5. TF and EF. Assigning values to technical and environmental factors are usually done by project experts or project leaders, based on their judgment and without any reference [Anda01][Ribu01]. The authors of these papers conclude that the technical factors can be omitted (or set to 1) without large consequences for the estimate. The environmental factors may have a large impact on the estimate, but these are also subjective, and the formula should be validated. We decided to simplify the method by assuming an average project, which gives TCF and EF approximately equal to 1.

Step 6. As discussed in Section 2.3, there is an overhead for changing software of the previous release. The difference in functionality between two releases is large. The model proposed in [Benediktsson03] suggests reduction in effort due to incremental development only when the number of iterations is sufficiently high. There are no generally accepted rules for the overhead factor. We decided to use the reuse model proposed in COCOMO 2.0 as a first trial. COCOMO 2.0 has an equation for calculating effort for modifying reused software. It calculates the equivalent new software ESLOC (Equivalent SLOC) as:

$$ESLOC = ASLOC*AF \qquad \textit{EQ.2}$$

$$AF = 0.01*(AA+SU+0.4*DM+0.3*CM+0.3*IM) \qquad \textit{EQ.3}$$

The abbreviations in *EQ.2* and *EQ.3* stand for:

ASLOC = Adapted SLOC,
AF = Adaptation Factor,
AA = Assessment and Assimilation increment,
SU = Software understanding increment,
DM = percentage of design modification,
CM = percentage of code modification,
IM = percentage of the original integration effort required to integrate the reused software.

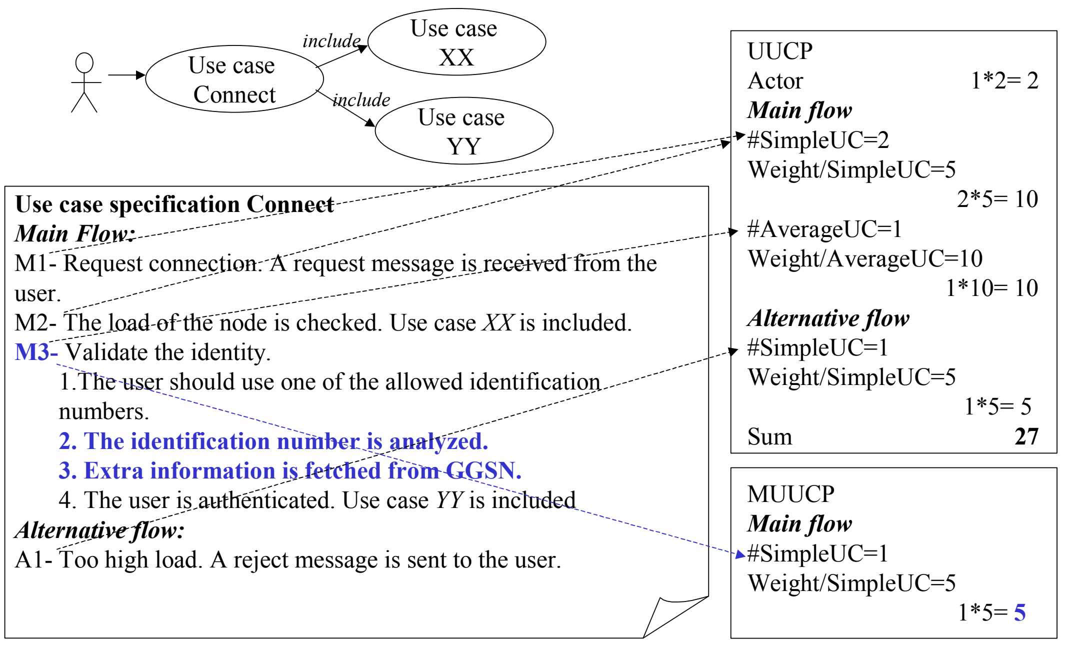
Thus, if software is reused without modification, DM, CM, and IM are zero, but there is cost related to assessment (AA) and understanding of reused software (SU). The cost will increase with the modification degree. DM, CM and IM vary from 0 to maximum 100. Note that AF can become larger than 1; i.e. reuse may cost more than developing from scratch if the cost of assessment or understanding is high, or if the reused software is highly modified. For our model, in the simplest form we propose:

- AA = 0, we assume no search, test, and evaluation cost since reused software is developed in-house,
- SU = 30 for moderate understandable software,
- Mean values for DM, CM and IM may be set to 25, which is the mean for changes in the use cases in the two releases. I.e. we assume that the fraction of

design and code modification and integration effort is equal to the fraction of modification in use cases.

Thus, AF will be 0.55. We have not found any empirical studies that contain such a factor.

In this project, we decided to compensate for not counting the environmental factors and for the large number of complex use cases by using the maximum recommended number of person hours pr use case point, which is 36.



include
include
Use case Connect
Use case XX
Use case YY
Use case specification Connect
Main Flow:
M1- Request connection. A request message is received from the user.
M2- The load of the node is checked. Use case XX is included.
M3- Validate the identity.
1.The user should use one of the allowed identification numbers.
2. The identification number is analyzed.
3. Extra information is fetched from GGSN.
4. The user is authenticated. Use case YY is included
Alternative flow:
A1- Too high load. A reject message is sent to the user.
UUCP
Actor 1*2= 2
Main flow
#SimpleUC=2
Weight/SimpleUC=5
2*5= 10
#AverageUC=1
Weight/AverageUC=10
1*10= 10
Alternative flow
#SimpleUC=1
Weight/SimpleUC=5
1*5= 5
Sum 27
MUUCP
Main flow
#SimpleUC=1
Weight/SimpleUC=5
1*5= 5

Figure 1 Example of counting UUCP and MUUCP for a use case

6. Estimation Results

The method was adapted for use cases of one release and later used it in on use cases from the successive release. Of 23 original use cases in release 1, seven use cases were not modified, one use case was new, while 15 use cases were modified. Release 2 had 21 use cases: two use cases were not modified, one use case was new, while 20 were modified. Note that 3 use cases are missing in release 2 (the sum should be 24). Two use cases are merged in other use cases in release 2, while one use case is removed from our analysis since development was done by another organization and we do not have data on actual effort for this use case.

Table 4 shows the results of breaking use cases into smaller ones (288 use cases in release 1, and 254 use cases in release 2). Columns in Table 4 present the number of use cases in each class (Simple, Average, and Complex) and also modified ones. The distribution has changed towards more average use cases after restructuring. According to Cockburn most well-written use cases have between 3 and 8 steps [Cockburn00], consequently most use cases will be of medium complexity, some are simple, and a few are complex. Our results only verify this for release 2.

Table 4 No. of use cases in each class

Rel.	Simple UC	Average UC	Complex UC	Modified Simple UC	Modified Average UC	Modified Complex UC
1	170	83	35	57	18	2
2	95	100	59	81	16	11

We inserted the number of steps, actors, and exceptions and parameters for all use cases in spread sheets in Excel, counted the UUCP and MUUCP, and estimated the effort following the rules in Table 3. The estimation results with 36 PH/UCP were almost half the effort spent in the releases for all activities. Therefore we compared our releases with the examples discussed before in other aspects. In projects *A* and *B* in Table 2, estimates have been compared with the total effort after the construction of the use case model. The UCP method, however, does not specify exactly which phases of a development project are estimated by the method. These projects' effort distribution is very different from our case, as shown in Tables 5 and 6. The *Other* column in Table 5 covers deployment and documentation, while in Table 6 it covers configuration management, software process adaptation, documentation and travels.

Table 5 Percentages of actual effort spent in different phases in example projects

Project	Development before System Test	System Test	Other	Project Management
A	80%	2%	5%	13%
B	63%	7%	3%	27%

Table 6 Percentages of actual effort spent in different phases in two releases

Rel.	Development before System Test	System Test	Other	Project Management
1	49%	25%	15%	10%
2	55%	18%	15%	11%

These profiles will vary depending on tools, environment and technologies. In our case, development before system test (also including use case testing) only counts for half the actual effort. The estimation method in the company estimates effort needed for development before system test and multiplies this by a factor (between 1.0 and 2.5) to cover all the activities. We concluded that the 36 PH/UCP covers for development before system test. Based on data presented in Table 6, it should be multiplied approximately by 2 to estimate the total effort.

For confidentiality reasons, we cannot give the exact figures for estimations. However, our estimations were 20% lower for release 1 and 17% lower for release 2 than the actual effort with the assumptions described before. The expert estimations for release 2 were 35% lower than the actual effort and thus the method has lower relative error than expert estimations.

7. Discussion of the Results

The results show that the adapted UCP method produced reasonable estimates with the following assumptions:

- We broke each use case down to several smaller ones, justified by the complexity of use cases.
- Classification of use cases is different from Table 1, justified by the complexity of steps.
- We assumed the technical and environmental factors to be 1 for an average project. These factors are highly subjective.
- The Adaptation Factor for reused software is 0.55.
- The results with 36 PH/UCP estimate the effort for specification, design, coding, module test and use case test.

We have done several assumptions and the method is only as good as the assumptions are. The method was first tried on release 1, but it even gave better results for release 2. Each estimate should also come with a range, starting with a wider range for early estimations. Use cases are updated in the early design stage, which gives a range $0.67E$ to $1.5E$ (E is the estimated effort) according to COCOMO 2.0 [Boehm95]. Thus, 20% underestimation is acceptable, but there are factors in our model that could be optimized to provide more accurate estimates. These are essentially two factors: PH/UCP and AF.

The impact of the reused software is large on the total effort. In addition to factors described in Section 2.3, several other factors may also be influential:

- We have performed a study of Change Requests that cover changes in requirements or artifacts in each iteration and between releases [Mohagheghi04]. The results show that most change requests are initiated in order to improve quality requirements (non-functional requirements), which are of high importance but are not reflected in the use cases. Improving quality requirements is by modifying software that is already implemented.
- The above study shows that functionality is also improved between releases by initiating change requests.
- Some effort is spent on modifying software for bugs (corrective maintenance).

We could propose a higher AF to compensate for bug fixing of previous releases and improvements that are not specified in use cases. We can also explain the high value of PH/UCP by:

- Complexity of the system,
- Diseconomy of scale,
- Importance of quality requirements as described above,
- Effort spent on Change Requests,
- Increased effort spent on configuration management and regression testing due to incremental development, cf. the profile in Table 6.

The study has several factors that improve the validity: The data on the spent effort is reliable, we did the estimation without involving the project members, and we have had access to all the use cases. The following validity threats are identified (no threats are identified for internal validity):

- Conclusion validity: The method is tested on one release, in addition to the release used for adaptation. Future updates may be necessary.
- Construct validity: A single case study is not sufficient for calibrating all the parameters that may influence the results.
- External validity: Generalization of the concrete results is not possible without testing the method on other data.

8. Summary

Already when the UCP method was introduced to the project leaders to get their permission for the study, it was considered interesting. A project leader used it in addition to expert estimates by considering the amount of changes in use cases comparing to the previous release. We answer the research questions as:

RQ1: Does the UCP method scale up for a large industrial project? It did when we broke down the use cases as reflected in Rules 2.1-2.5 in Table 3. The method depends on the level of details in use cases and therefore should be adapted to the context by comparing use cases to some examples. One alternative is to include examples of typical use cases in the method, such as defined in [Cockburn00].

RQ2: Is it possible to apply the UCP method to incremental changes in use cases? We did this by counting changes in use cases. The method is straightforward and Rules 1.2, 2.6, 3.2 and 6.2 in Table 3 show how to calculate effort for new development.

RQ3: How to calculate effort needed to reuse software from the previous release? We chose to account for reuse by applying the COCOMO 2.0 formula for reused software, calculating AF and applying it on UUCP for reused steps in use cases. The advantage is that the AF factor may be adapted to the context.

RQ4. Evaluation of the method: The adapted UCP method fitted well into the adapted RUP process and produced reasonable results. The impact of technical and environmental factors may be subject to future studies, for example by defining some profiles.

We also observe the impact of size, complexity of the system and effort spent on configuration management due to incremental development in the high value of PH/UCP. The study also raises some interesting question: Does the value of PH/UCP depend on the effort breakdown profile and should this factor be included in the model? What is the cost of reusing software in incremental development?

9. Conclusions

The UCP method is adapted for a large industrial system with incremental changes in use cases and with reuse of software. Contributions of the study are:

1. Verifying that the method scales up, with our assumptions and by applying the proposed changes.
2. Adapting the UCP method to evolutionary development of software by accounting for reuse of software from a previous release and changes in use cases. We assume that the method is also applicable for reuse of software in a product family approach, or when reusing COTS components.
3. Verifying that the method works well without technical and environmental factors.

The UCP method for estimation can be considered as a relative cheap, repeatable and easy method to apply. It is not dependent on any tools and can promote high quality use cases, which will pay off since use cases are also input to test cases, analysis and documentation.

10. Acknowledgements

The studies are performed in the context of INCO (INcremental and COmponent-based Software Development), a Norwegian R&D project in 2001-2004 [INCO01], and as part of the first author's PhD study. We thank Ericsson for the support.

References

[Albrecht79] Albrecht, A.J.: Measuring Application Development Productivity. *Proc. IBM Application Development Joint SHARE/GUIDE Symposium*, Monterey, CA, 1979, pp. 83-92.

[Anda01] Anda, B., Dreiem, D., Sjøberg, D.I.K., and Jørgensen, M.: Estimating Software Development Effort Based on Use Cases - Experiences from Industry. In M. Gogolla, C. Kobryn (Eds.): UML 2001 - The Unified Modeling Language.Modeling Languages, Concepts, and Tools, 4th International Conference, 2001, LNCS 2185, *Springer-Verlag*, pp. 487-502.

[Anda02] Anda, B.: Comparing Effort Estimates Based on Use Cases with Expert Estimates. *Proc. Empirical Assessment in Software Engineering* (EASE 2002), Keele, UK, April 8-10, 2002, 13p.

[Arnold98] Arnold, P. and Pedross, P.: Software Size Measurement and Productivity Rating in a Large-Scale Software Development Department. Forging New Links. *IEEE Computer Soc*, Los Alamitos, CA, USA, 1998, pp. 490-493.

[Benediktsson03] Benediktsson, O., Dalcher, D.: Developing a new Understanding of Effort Estimation in Incremental Software Development Projects. *Proc. Intl. Conf. Software & Systems Engineering and their Applications* (ICSSEA'03), December 2-4, 2003, Paris, France. Volume 3, Session 13, ISSN 1637-5033, 10 p.

[BEST04] The Best project: http://www.simula.no/~simula/se/bestweb/index.htm

[Boehm95] Boehm, B., Clark, B., Horowitz, E., Westland, C., Madachy, R., Selby, R.: Cost Models for Future Software Life Cycle Processes: COCOMO 2.0. *USC center for software engineering*, 1995. http://sunset.usc.edu/publications/TECHRPTS/1995/index.html

[Cockburn00] Cockburn, A.: *Writing Effective Use Cases*. Addison-Wesley, 2000.

[Enterprise] www.sparksystems.com.au

[Estimate] www.duvessa.com

[Fetcke98] Fetcke. T., Abran, A. and Nguyen, T.-H.: Mapping the OO-Jacobson Approach into Function Point Analysis. International Conference on Technology of Object-Oriented Languages and Systems (TOOLS-23). *IEEE CS Press*, Los Alamitos, CA, USA, pp. 192-202, 1998.

[INCO01] The INCO project: http://www.ifi.uio.no/~isu/INCO/

[Jørgensen03] Jørgensen, M., Moløkken, K.: Situational and Task Characteristics Systematically Associated With Accuracy of Software Development Effort Estimates. *Proc. Information Resources Management Association Conference (IRMA 2003)*, pp. 824-826.

[Jørgensen04] Jørgensen, M.: Top-down and Bottom-up expert Estimation of Software Development Effort. *Information and Software Technology*, vol.46 (2004), pp. 3-16.

[Karner93] Karner, G. Metrics for Objectory. Diploma thesis, University of Linköping, Sweden. No. LiTH-IDA-Ex-9344:21, December 1993.

[Mohagheghi04] Mohagheghi, P., Conradi, R.: An Empirical Study of Software Change: Origin, Impact, and Functional vs. Non-Functional Requirements. Accepted for the ACM-IEEE International Symposium on Empirical Software Engineering (ISESE 2004), 19-20 August 2004, Redondo Beach CA, USA, 10 p.

[Moløkken02] Moløkken, K.: Expert Estimation of Web-Development Effort: Individual Biases and Group Processes. Master's Thesis, University of Oslo, 2002.

[Moløkken04] Moløkken, K., Lien, A.C., Jørgensen, M., Tanilkan, S.S., Gallis, H., Hove, S.E.: Does Use of Development Model Affect Estimation Accuracy and Bias? Proc. Product Focused Software Process Improvement: 5th International Conference, PROFES 2004, Kansai Science City, Japan, April 5-8, 2004. *Springer-Verlag*, ISBN: 3-540-21421-6. pp. 17-29.

[Ribu01] Ribu, K.: Estimating Object-Oriented Software Projects with Use Cases. Master's Thesis, University of Oslo, November 2001.

[Russell04] Rusell, R.: Project Estimation Method. http://www.processwave.net/index.htm, cited July 1, 2004.

[Schneider98] Schneider, G., Winters, J.P.: *Applying Use Cases a Practical Guide*. Addison-Wesley, 1998.

[Smith91] Smith, J.: The Estimation of Effort Based on Use Cases. *Rational Software*, White paper, 1999.

[Symons91] Symons, P.R.: *Software Sizing and Estimating MK II FPA (Function Point Analysis)*. John Wiley & Sons, 1991.

[Verner03] Verner, J.M., Evanco, W.M.: State of the Practice: Effect of Effort Estimation on Project Success. *Proc. of the Intl. Conf. On Software & Systems Engineering and their Applications* (ICSSEA'03), Vol. 3, Session 13, 10 p.